OPERATING SYSTEM

CHAPTER ONE

INTRODUCTION TO OPERATING SYSTEM

SYSTEM SOFTWARE AND APPLICATION SOFTWARE

Those computer software that control and monitor the computer hardware and provide essential functionality to the computer are called system software. Therefore , system software are essential part of a computer , which means a computer can not perform its functions with out the system software. System software are entirely responsible for creating an interface between the computer's user and computer hardware.

System software are the system-side computer programs that are required for the working of the computer itself , although the user of the computer does not need to know about the functionality of system software while using the computer. Examples of system software include : operating systems , device drivers , language processors , etc ...

Some examples of system software include :-

Operating systems :- These are the core software programs that manage a computer's resources , including the CPU , memory and storage devices. Popular operating systems include Windows , macOS , Linux , and Android.

Device Drivers :- These programs enable a computer to communicate with hardware devices such as printers , scanners , and graphic cards. Device drivers act as a translator between the hardware and the operating system , allowing the two to work together seamlessly.

Utility programs :- These programs perform system-level tasks such as managing memory , scheduling tasks , and controlling hardware devices. Examples of utility programs include antivirus software , disk de-fragments , and system back up tools.

What is an Application Software

A computer software which is developed to perform a specific function is known as application software. Application software are also called end user software because they are designed to use by users of the computer. An application software may be a set of computer programs or a single computer program , these software provide the required functionality for a specific purpose. Thus , a computer user uses these software to accomplish a specific task.

The different types of application software can be developed to accomplish different tasks such as word processing , playing multimedia files , database applications , spread sheets , accounting , graphic designing etc ...

Examples of application software include :-

- word processing programs such as Mircosoft Word and Google Docs
- Spreadsheeet programs such as Microsoft Excel and Google sheets
- Database programs such as oracle and Microsoft access

Definition of an operating system

An operating system (os) is a software program that manages and controls the hardware and the software resources of a computer system. It acts as an interface between the user and the computer's hardware and provides services and tools for application programs to interact with the computer.

The operating system is responsible for managing the computer's memory , processing power , input/output devices and other resources. It also provides a platform for running other software applications , manages the file system and provides a user interface for interacting with the computer.

Some common examples of operating systems include :-

Microsoft Windows :- This is the most widely used operating system for personal computers

MacOs :- This is the operating system used on Apple's Mac computers

Linux :- This is an open source operating system that is widely used in servers and other systems.

Android :- This is an operating system designed for mobile devices such as smart phones and tablets.

Types of operating system's

Single-User systems

A single user operating system that is designed to be used by one user at a time. It is typically found on personal computers , work stations and other systems that are used by individual users rather than shared among multiple users. The goals of such systems are maximizing user convenience and responsiveness , instead of maximizing the utilization of the CPU and peripheral devices.

Single user operating systems are designed to provide a simple , user-friendly interface that allows users to perform basic tasks such as file management , document creation and editing and web browsing. They typically provide a graphical user interface (GUI) that allows users to interact with the computer system using icons , menus , and windows.

Single user operating systems are generally easier to use and manage than multi–user operating systems , which are designed to support multiple users simultaneously , they are also less complex , as they do not need to manage access to system resources by multiple users.

Batch systems

In operating systems courses , batch systems refer to a type of operating system that is designed to process a large number of similar jobs or tasks in a batch mode , without any user interaction during the processing of each job.

Batch systems were popular in the early days of computing when computers were large and expensive , and user has to share the system resources. In this environment , users would submit their jobs or tasks to a system operator , who would then queue them up for processing in batch mode.

Batch systems typically use a queue to manage the jobs or tasks that are waiting to be processed. Each job is executed in turn , without any user interaction , and the system may print out a report or summary of the results once the job has finished.

One advantage of batch systems is that they can efficiently process large volumes of similar jobs , such as printing reports or processing payroll , they can also help to maximize the utilization of system resources by allowing multiple jobs to be processed simultaneously.

Multi-programmed systems

In operating systems, a multi–programmed system is a type of operating system that allows multiple programs to run on a computer system simultaneously , with the goal of maximizing the utilization of system resources and improving overall system performance.

In multi programmed system , multiple programs are loaded in to the computer's memory and are executed concurrently , with the operating system controlling the execution of each program. The operating system use techniques such as time-sharing , in which each program is allocated a small slice of time to execute and priority scheduling , in which higher priority programs are given more time to execute.

One advantage of multi-programmed systems is that they can improve overall system performance by allowing multiple programs to run simultaneously and share system resources such as CPU time and memory. This can help to reduce the idle time of the CPU and improve overall system throughput.

Another advantage of multi-programmed systems is that they can improve the response time of the system for interactive applications , by allowing the operating system to switch quickly between different programs as needed.

Time-sharing or Multi-tasking systems

Time sharing or multi tasking systems in operating systems refer to a type of system that allows multiple users or tasks to run concurrently on a single computer system. These systems enable users to interact with the computer in real

time and provide a seamless experience for running multiple applications simultaneously.

In a time-sharing or multi tasking system , the CPU is divided in to multiple time slots or time slices and each user or task is allocated a small slice of CPU time to execute its instructions. The operating system manages the allocation of CPU time , and switches between tasks or users rapidly to give the impression of simultaneous execution.

Parallel or Multiprocessor systems

Parallel or multi processor systems in operating systems refer to a type of system that uses multiple processors or CPUs to perform computations and execute instructions in parallel. These systems enable faster and more efficient processing of complex tasks by distributing the work load across processors.

In a parallel or multi processor system , each processor has its own memory and I/O devices and is capable of executing instructions independently of the other processors. The operating system manages the allocation of tasks to different processors and ensures that the processors are synchronized and communicating effectively.

Real time operating systems

Real – time systems in operating systems refer to a type of system that is designed to respond to events or stimuli in a timely and predictable manner. These systems are used in applications where timing is critical , such as in process control , industrial automation and mission – critical systems.

In real – time system , the operating system must respond to events or stimuli with in a specified time frame , known as deadline. If the system fails to respond with in the deadline , the consequences can be severe , such as equipment damage or loss of life.

Real time systems can be classified in to two types :-

Hardware real time systems :- These systems have strict timing requirements and must meet all deadlines , even in the face of resource contention or other system constraints.

Soft real time systems :- These systems have less strick timing requirements and can tolerate occasional missed deadlines , as long as the system can recover and continue to function properly.

CHAPTER TWO

PROCESS AND PROCESS MANAGEMENT

Early computer systems allowed only one program to be executed at a time. This program had complete control of the system , and had access to all of the system's resources. Current day computer systems allow multiple programs to be loaded in to memory and to be executed concurrently. This evolution required firmer control and more compartmentalization of various programs. These needs resulted in the notion of a process , which is a program in execution. A system therefore consists of a collection of processes , all these processes can potentially execute concurrently , with the CPU (or CPUs) multiplexed among them. By switching the CPU between processes , the operating system can make the computer more productive.

The Process :- Informally , a process is a program in execution. The execution of a process must progress in a sequential fashion. That is at any time , at most one instruction is executed on behalf of the process. We emphasize that a program by itself is not a process , A program is a passive entity , such as the contents of a file store on a disk , where as a process is an active entity , with a program counter specifying the next instruction to execute and a set of associated resources.

Process state

As a process executes , it changes state. The state of a process is defined in part by the current activity of that process. Each process may be one of the following states ….

New State :- The process is being created
Running :- Instructions are being executed
Waiting :- The process is waiting for some event to occur (such as an IO completion or reception of a signal)

Ready :- The process is waiting to be assigned to a processor
Terminated :- The process has finished execution

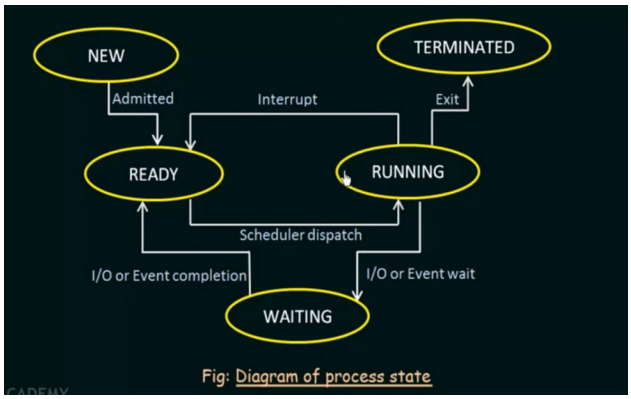


Fig: Diagram of process state

PROCESS CONTROL BLOCK

Each process is represented in the operating system by a process control block (PCB). A process control block is something that we used to represent a particular process in the operating system.

What are the things we have in the process control block which helps us to represent the process in the operating system.

Pointer	Process State
Process Number	
Program Counter	
CPU Registers	
Memory Limits	
Event Information	
List of Open Files	
.	
.	

Figure 2.2: Process control block.

These block tell or represent about the process in an operating system.

Process number (process – id) :- Shows a unique id of a particular process , so every process has to be represented by a unique id.

Process state :- tells us a particular state in which a process is at that particular moment , from the different states that we have , this tells In which of those states the process is in at that particular moment.

Program counter :- It indicates the address of the next instruction that has to be executed for that particular process , so we have lines of instructions that are being executed and at a particular moment what is the address of the next line that has to be executed , that is given by the program counter.

CPU registers :- It tells us the registers that are being used by a particular process. The registers vary in number and type depending on the computer architecture , they include accumulators , index registers , stack pointers and general purpose registers and so on. So these CPU registers will tell us the particular register that are used by a particular process.

CPU Scheduling information :- we know that there are many processes that are waiting to be executed , so scheduling determines which process has to be executed first or it determines the order in which the processes has to be executed.

Accounting information :- Keeps an account of certain things like the resources that are being used by the particular process like CPU , time and memory. It keeps an account of all the information that are being used by a particular process for its execution.

I/O status information :- represents the input/output devices that are being assigned for a particular process. A process during execution may need to use some input/output devices , so which are the input/output devices assigned stored in the input/output status information.

Process Scheduling :-

The objective of multi programming is to have some process running at all times , to maximize CPU utilization. The objective of time sharing is to switch the CPU among processes so frequently that users can interact with each program while it is running. For a uni-processor system , there will never be more than one running process. If there are more processes , the rest will have to wait until the CPU is free and can be rescheduled.

Process scheduler selects an available process (possibly from a set of several available processes) for program execution on the CPU.

SCHEDULING QUEUES

In operating systems, scheduling queues refer to the data structures used to organize and manage the execution of processes or threads. These queues store the processes or threads in different states, allowing the scheduler to determine which one should be executed next.

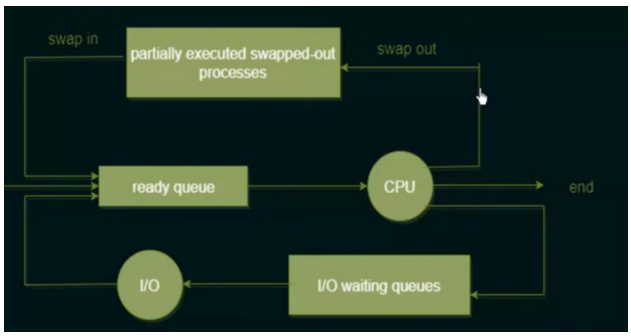
Job Queue :- As a process enter the system they are put in to a job queue , which consists of all processes in the system. So they may not be executing at this time but there are the lists or the set of all processes that we have in the system , so all these processes they are put in this set or this queue known as the **Job Queue** , where they will be waiting In order to go to the ready state.

Ready Queue :- The processes that are residing in main memory and are ready and waiting to execute are kept on a list called the ready queue , in ready queue we have those processes , which are ready and waiting to be executed , they are just waiting to get the CPU , so that they can start executing.

Waiting Queue: This queue stores the processes or threads that are waiting for a specific event or resource to become available before they can proceed with their execution. Examples of events or resources include I/O operations, signals, or synchronization primitives.

Device Queue: These queues are associated with specific devices in the system, such as printers or disks. Processes or threads waiting for access to a particular device are placed in the corresponding device queue until the device becomes available.

Priority Queue: This queue assigns different priorities to processes or threads based on their importance or urgency. The scheduler uses these priorities to determine the order of execution, giving higher priority to critical tasks or time-sensitive operations.



So let's say that from the Job queue , the process came to the ready queue , so let's say a particular process is in the ready queue and it's just waiting to get the CPU for its execution and once it gets the CPU and it begins its execution , but at this stage there are several cases that can occur :-

The first thing that can happen is , it received the CPU and it completed its execution and then the process just ends , that means it goes to the **terminated state.** Another thing that can happen is , it comes from the ready queue and it is assigned to the CPU and it's holding the CPU for it's execution but at the same time , let's say that there is another process that comes with a higher priority , that means this process that comes now is having a higher priority and it has to be executed before this , so the process that was previously executing , it will get swapped out and it goes to a list called the **partially executed swapped out process** , and after that it gets swapped in to the ready queue again , so the execution has not completed fully , it is just partially executed , it still needs a cpu to complete it's executions.

Another thing that can happened again is from the ready queue it is assigned to the CPU , and it begins its execution and at some point of time , it needs some input / output devices , at some point of time the process may need to use some of the input / output devices , so it will go to the input/output waiting queue , where it will wait for the input/output operation to occur , which is required by that particular process. If the input/output devices are available at that time , that operation can immediately occur and it continue its execution but if the input output devices are being held by some other processes and if it has to wait then this process will come to the input/output waiting queue and wait for the availability of the input/output devices and once the input/output devices are assigned to the

process it will execute the input/output operation and it will go back to the ready queue again because it needs the CPU again to resume its execution.

Context Switch

Interrupts cause the operating system to change a CPU from its current task and to run a kernel routine.

When a process are being executed , if an interrupt occurs or a process of higher priority comes then that particular process has to stop its execution and it has to allow the process that causing the interrupt to execute first and after that is completed the previous process can resume its execution.

When processes are being executed , the CPU is being assigned to that particular process , so when an interrupt occurs , the CPU has to be assigned to that process or the thing which is causing the interrupt so that it will execute first and when that is completed , the CPU re-assigned back to the process that was previously being executed.

So when an interrupt occurs , the system needs to save the current context of the process currently running on the CPU , so that it can restore the context when its processing is done , essentially suspending the process and then resuming it.

When an interrupt occurs , the process that was currently running needs to save its context to the CPU , the reason we need it is because when the process resume its execution , it needs to know where it did it stop executing at that particular moment so that when it resumes , it will know from where it has to resume its operation again.

SCHEDULERS

Schedulers in operating systems are responsible for managing the execution of processes and deciding which processes should be allocated system resources such as the CPU and memory.

There are three types of schedulers:

1) Long Term Scheduler (Job Scheduler): This scheduler determines which programs or processes should be admitted to the system for execution. Its main objective is to create a balanced mix of different types of jobs, such as those that require a lot of input/output operations (I/O bound) or those that require a lot of CPU processing (processor bound). The long term scheduler controls the degree of multiprogramming, which refers to the number of processes that are allowed to run concurrently. In some systems, this scheduler may be minimal or absent, especially in time-sharing operating systems.

This scheduler is like a smart gatekeeper who decides which programs or tasks should be allowed to enter the computer for doing their work. It tries to make sure that different types of tasks, like ones that need a lot of reading or writing, or ones that need a lot of thinking, get a fair chance. It also keeps an eye on how many tasks can be done at the same time.

2) Short Term Scheduler (CPU Scheduler): The short term scheduler aims to increase system performance by selecting which process should be executed next from the pool of processes that are ready to run. Its main job is to allocate the CPU to one of the processes in the ready queue. The short term scheduler, also known as the dispatcher, makes frequent decisions about which process to execute next. It operates at a faster rate compared to the long term scheduler.

This scheduler is like a quick decision-maker that picks the next task to be done from a list of tasks waiting for their turn. It's in charge of making sure that each task gets a fair chance to use the computer's brain, which is called the CPU. It's a very fast decision-maker and always wants to keep things moving smoothly.

3) Medium Term Scheduler: The medium term scheduler is responsible for handling the swapping function, which involves moving processes out of main memory (swapping out) to create space for other processes. This occurs when a running process makes an I/O request

and becomes suspended, unable to make progress. To free up memory, the suspended process is moved to secondary storage, which is usually a hard disk. This swapping process reduces the degree of multiprogramming, helping to improve the overall process mix in the system.

This scheduler helps manage the computer's memory, which is like its working space. Sometimes, when a task is waiting for something else, like when it needs to read from a hard disk, it gets put aside temporarily to make room for other tasks. The medium term scheduler helps with this, making sure that the right tasks are moved out of the working space to make room for new tasks. It helps keep things organized and running well.

In simpler terms, the long term scheduler decides which programs should be allowed to run, the short term scheduler chooses which program should run next on the CPU, and the medium term scheduler handles the swapping of processes between memory and secondary storage. Each scheduler has its own role in managing the execution of processes and optimizing system performance.

Operation On Processes

Processes in an operating system are like tasks or jobs that the computer needs to do. The operating system has to manage the creation and deletion of processes. Here's a simplified explanation of how this works:

Creating a Process: When the operating system creates a new process, it gives it a name and some characteristics. It's like giving a new task a name and some instructions. A process can also create other processes, like a parent giving birth to children. Each new process can create more processes, forming a tree-like structure. When a process is created, the operating system prepares a special block of information called the Process Control Block (PCB), which holds important details about the process. The PCB is then added to a list of ready processes, meaning it's ready to run.

Imagine you have a big task to do, like cleaning your room. But instead of doing it all by yourself, you decide to divide the work among your siblings. You become the parent, and your siblings become the children.

So, you start by assigning different parts of the room to each sibling. Each sibling becomes responsible for their assigned area. Now, here's where it gets interesting. Each sibling can also ask for help from their own friends or siblings. They become parents themselves, and their friends or siblings become their children. This way, the work of cleaning the room is divided and shared among everyone.

In the same way, in a computer operating system, a process (like a program or task) can create child processes under it. The original process is the parent process, and the newly created processes are its children. Just like you assigned tasks to your siblings, the parent process can assign specific tasks or share the workload among its child processes.

And similar to how your siblings can also create their own children to help with the work, child processes can create their own child processes as well. This creates a tree-like structure where the parent process can have children, and those children can have their own children, and so on.

By creating these child processes, the computer can perform multiple tasks simultaneously, making the most efficient use of its resources and getting work done more quickly.

So, process creation is like dividing a big task into smaller tasks, assigning them to different processes, and allowing those processes to create more processes if needed, forming a tree-like structure of tasks being executed by the computer.

When a process creates child processes, there are two possibilities for how they can execute:

The parent and children execute concurrently: This means that the parent process and its child processes can execute at the same time. It's like a parent working on their task while the children

work on their own tasks simultaneously. They can all make progress together, independently, without waiting for each other.

The parent waits for its children to complete: In this case, the parent process pauses its own execution and waits for its child processes to finish their tasks. It's like a parent waiting for their children to complete their work before continuing with their own task. Once all the child processes have finished, the parent process resumes its execution.

To better understand the concept of resources, let's imagine a scenario. Think of the parent process as a baker in a bakery. The baker has all the necessary resources like ingredients, tools, and equipment. When the baker creates child processes, which could be other bakers, the child bakers may receive the same set of resources as the parent baker or a subset of those resources. For example, they might have their own ingredients but share the same equipment.

So, in simpler terms, when a process creates child processes, the parent and children can either work together at the same time, each doing their own tasks, or the parent can pause and wait for the children to finish before continuing with its own task. The resources needed for the tasks can be shared or partially shared between the parent and children processes, depending on how they are created.

TERMINATE A PROCESS

Process termination refers to the event when a process comes to an end and is stopped. There are several aspects to consider when understanding process termination :-

1. **Process Completion:** When a process reaches its final statement, it has finished executing all the tasks or lines of code assigned to it. At this point, the process requests the operating system to delete it using an "exit" system call. This call signals that the process has completed its work and wants to be removed from the system.
2. **Returning a Status Value:** Upon termination, a process may provide a status value or result back to its parent process. This status value is typically an integer that indicates the outcome or relevant information about the process's execution. It helps the parent process understand the result or status of its child process.
3. **Resource Deallocation:** When a process terminates, all the resources it was using are deallocated by the operating system. These resources include physical and virtual memory, open files, and input/output buffers. Deallocation ensures that the resources are freed up and available for other processes to utilize.

Additionally, there are cases where one process can cause the termination of another process through an appropriate system call. However, such termination is usually limited to the parent process terminating its child processes. This mechanism prevents unauthorized termination of unrelated processes and maintains control within the process hierarchy.

The parent process may decide to terminate its child process for various reasons, including:

- **Resource Overuse:** The child process has exceeded its allocated resources, such as memory or CPU usage. This termination prevents resource exhaustion and allows other processes to utilize those resources effectively.
- **Task Completion:** The task assigned to the child process is no longer required or relevant. If the parent determines that the child's task is no longer necessary, it can terminate the child process to free up resources and eliminate unnecessary processing.
- **Parent Termination**: In some cases, when the parent process is exiting or

terminating, the operating system may not allow its child processes to continue running independently. Therefore, the parent process ensures that its children processes also terminate to maintain system integrity and prevent orphaned processes.

Overall, process termination involves the completion of a process's tasks, potential communication of status values, and the deallocation of resources. The termination can be initiated by the process itself upon completion or by the parent process due to resource management or task requirements.

INDEPENDENT AND COOPERATING PROCESS

When multiple processes are running in an operating system, they can be categorized as either independent processes or cooperating processes.

1. **Independent Processes:** Think of independent processes as separate individuals who are working on their own tasks without any interaction or influence from other processes. They don't share any data with other processes. Each process is like a self-contained entity, doing its own thing without being affected by or affecting other processes.
2. **Cooperating Processes:** In contrast, cooperating processes are like a team or group of people working together. They can affect or be affected by other processes in the system. These processes share data with each other, meaning they exchange information and collaborate to accomplish a task or solve a problem. The actions of one cooperating process can impact the behavior or outcome of other cooperating processes.

To put it simply, independent processes are like individuals working independently, while cooperating processes are like team members working together and sharing information.

It's important to note that the distinction between independent and cooperating processes depends on whether they share data with each other. If processes don't share any data, they are independent. However, if they exchange information or work together by sharing data, they are considered cooperating processes.

INTER PROCESS COMMUNICATION

Interprocess Communication (IPC) refers to the mechanisms and techniques used by processes in an operating system to exchange information and coordinate their activities. IPC allows processes to communicate with each other, share data, and synchronize their actions to work together effectively.

Processes in an operating system can run independently or cooperatively. When processes need to collaborate or share information, IPC provides the means for them to do so. It enables processes to send messages, share memory, or use other forms of communication to exchange data and coordinate their activities.

IPC is crucial for various scenarios, such as:

1. **Sharing Data:** Processes may need to exchange data, such as passing messages or sharing files, to cooperate and accomplish tasks collectively. IPC facilitates the transfer of information between processes.
2. **Synchronization:** Processes may need to synchronize their activities to maintain order or prevent conflicts. For example, one process might need to wait for another process to finish its task before proceeding. IPC provides synchronization mechanisms to control the flow and coordination of processes.
3. **Coordination:** Processes may need to coordinate their actions to accomplish complex tasks together. IPC allows processes to communicate and coordinate their activities to achieve a common goal efficiently.

There are different IPC mechanisms and techniques available, including message passing, shared memory, pipes, sockets, and remote procedure calls (RPC). Each mechanism has its advantages, and the choice depends on factors like the nature of the communication, performance requirements, and the level of coordination needed between processes.

Overall, IPC enables processes to communicate, share information, synchronize their actions, and work together to accomplish tasks in an operating system. It is a fundamental aspect of modern operating systems and plays a crucial role in facilitating collaboration and coordination between processes.

Processes often need to communicate with each other. Think of it like people talking to each other to share information or work together. In the computer world, processes also need a way to pass information or work together in a structured manner. This is what we call interprocess communication or IPC.

There are three main issues related to IPC:

1. **Passing Information:** One process needs a way to send information to another process. For example, in a pipeline of commands, the output of one command should go to the next command. So, processes should have a way to exchange information without interrupting each other's work.
2. **Avoiding Conflicts:** When multiple processes are doing critical tasks, we want to make sure they don't interfere with each other. Imagine if two processes both try to use the same memory at the same time. We need mechanisms to prevent such conflicts and ensure that processes work independently and efficiently.
3. **Sequencing Dependencies:** Sometimes, processes have dependencies or specific order requirements. For example, if one process produces data and another process needs to print it, the printing process should wait until the data is ready. We need to establish proper sequencing and synchronization between processes to maintain the correct order of operations.

It's important to note that two of these issues also apply to threads, which are smaller units of execution within a process. Threads can easily share information since they have a common memory space. However, they still face challenges in avoiding conflicts and sequencing dependencies. So, many of the same problems and solutions that apply to processes also apply to threads.

In the upcoming sections, we will delve into these IPC issues and explore different mechanisms to address them.

In simpler terms, interprocess communication is when processes need to talk to each other. They need ways to share information, avoid conflicts, and maintain the right order of operations. These concepts also apply to threads within a process. We will explore these topics further to understand how processes and threads can work together effectively.

THREADS

A thread is a unit of execution within a program or process. It is like a mini-program or a small worker that can perform tasks independently. Threads are a way to divide the work in a program so that multiple things can happen at the same time. Each thread has its own set of instructions (program), a register set (which stores information about the thread's current state), and a stack (a small space to keep track of what the thread is doing).

The advantage of using threads is that they can work together, allowing for parallelism and improved performance. Threads can perform different tasks simultaneously, making programs more efficient and responsive. For example, in a web browser, multiple threads can handle tasks like downloading web pages, rendering graphics, and handling user input, all happening concurrently.

Threads can communicate and synchronize with each other through shared memory or other mechanisms provided by the operating system, such as locks or semaphores. This coordination ensures that threads work together effectively and avoid conflicts when accessing shared resources.

However, it's important to note that threads also introduce challenges. Since threads share the same memory space, they must be careful not to interfere with each other's work or modify shared data simultaneously, which can lead to data corruption or inconsistent results. Proper synchronization mechanisms, such as locks or semaphores, are used to control access to shared resources and prevent race conditions.

In summary, a thread is like a small worker or mini-program within a program or process. It can execute tasks independently and concurrently with other threads, sharing the same memory space and resources. Threads allow for parallelism, improved performance, and coordination among different tasks. However, careful synchronization is necessary to avoid conflicts and ensure data integrity.

THREAD IN SIMPLER TERMS :- Imagine you have a big task to complete, like building a Lego model. Now, instead of doing it all by yourself, you have a team of friends who can help you. Each friend in your team is like a thread.

A thread is like a friend who can work on a specific part of the task independently. They have their own set of instructions (program) and a little space to keep track of what they're doing (stack). They can also share the same Lego pieces and tools (code and data) with other friends in the team.

The good thing about threads is that they can work together at the same time. For example, while one friend is building the base of the Lego model, another friend can work on the roof. They can also talk to each other and help if needed.

Threads are faster to switch between than separate tasks, like if you were building the Lego model alone and had to keep changing between different tasks. With threads, they can quickly switch between each other and keep the progress going smoothly.

However, because threads work together and share things, they need to be careful not to interfere with each other. They have to communicate well and make sure they're not doing the same thing at the same time. It's like your friends need to talk to each other and make sure they don't accidentally take the same Lego piece or build the same part twice.

So, in simpler terms, a thread is like a friend who helps you with a specific task. They can work together with other friends (threads) at the same time, sharing things and getting things done faster. But they also have to be careful not to interfere with each other and communicate well to avoid any problems.

SINGLE THREADED PROCESS AND MULTI THREADED PROCESS

Single-Threaded Process :- Imagine you have a task to complete, like baking a cake, and you're the only one working on it. In a single-threaded process, it's like you're doing the task all by yourself. You follow a step-by-step process, completing one task at a time before moving on to the next. You can't do multiple things at once. For example, you mix the ingredients, bake the cake, and then frost it. Each step needs to finish before you can move on to the next. It's a linear process where you can only do one thing at a time.

Multi-Threaded Process :- Now, imagine you have some friends helping you with the cake-making task. Each friend has their own specific job, like mixing the ingredients, baking the cake, or frosting it. In a multi-threaded process, it's like you and your friends are working together as a team. Each friend represents a separate thread, and each thread can work on a specific part of the task independently. For example, one friend can mix the ingredients while another friend preheats the oven. Threads can execute tasks concurrently and communicate with each other to coordinate their actions. It's like everyone is doing their part simultaneously, making the cake-

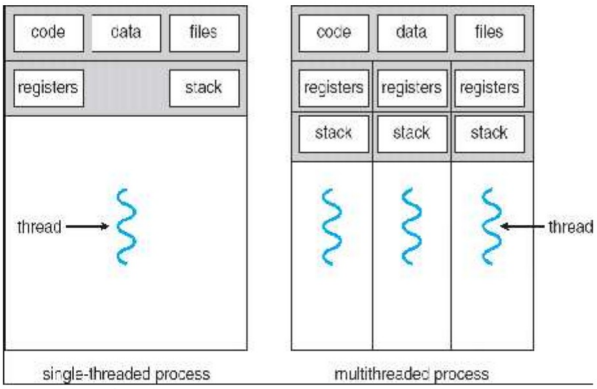


Figure 2.5: Single and Multi threaded process.

TYPES OF THREADS

- **USER LEVEL THREAD**
- **KERNEL LEVEL THREAD**

User-Level Threads: Imagine you and your friends are playing a game, and you're in charge of managing everything yourselves. User-level threads are like playing the game where you and your friends handle all the tasks without any help from adults. You decide who plays what role, how to communicate, and when to take turns. Everything related to the threads is managed within the game itself, without involving any outside authority. User-level threads are fast and flexible because you have control over them, but they may not have access to certain advanced features.

Kernel-Level Threads: Now, let's imagine a different scenario where a teacher oversees the game you and your friends are playing. Kernel-level threads are like playing the game under the supervision of the teacher. The teacher manages the game, ensures everyone follows the rules, and takes care of scheduling and coordination. Kernel-level threads rely on the operating system to handle them. The operating system provides support for managing threads and ensures fairness and stability. Kernel-level threads may be slower to create and manage because they involve the operating system, but they have access to advanced features and are more reliable.

To summarize:

- User-Level Threads: You and your friends handle everything in the game without help from adults. It's fast and flexible, but may not have access to advanced features.
- Kernel-Level Threads: A teacher oversees the game and ensures fairness. It may be slower to manage, but has advanced features and stability.

In simple terms, user-level threads are like playing a game with friends without adult supervision, while kernel-level threads are like playing a game with a teacher overseeing it for fairness and stability.

A SEMAPHORE

Imagine you are in a game where you need to take turns playing with a special toy. To make sure everyone gets a fair chance, there is a special object called a "semaphore" that helps control who can play with the toy at any given time.

The semaphore keeps track of how many people can play with the toy. Let's say the semaphore starts with a certain number, like 5. Each time someone wants to play with the toy, they have to ask the semaphore for permission. If there are still available spots (let's say the semaphore is not zero), they can take a turn and play. But if all the spots are taken (semaphore is zero), they have to wait for their turn until someone finishes playing and gives the spot back.

When someone finishes playing, they let the semaphore know by saying they are done. This increases the number of available spots, and now someone who was waiting can take their turn.

In simpler terms, a semaphore is like a special object that keeps track of how many people can do something at the same time. It helps make sure everyone gets a fair turn and prevents too many people from doing something all at once.

A semaphore is like a special variable that helps manage access to shared resources in a multi-processing environment. There are two common types of semaphores:

Counting Semaphores: Think of a counting semaphore as representing the available resources, like the seating capacity at a restaurant. For example, if a restaurant has a capacity of 50 people, the counting semaphore would start at 50. As people arrive and take up seats, the counting semaphore decreases. When the maximum capacity is reached (semaphore reaches zero), no more people can enter until someone leaves and increases the counting semaphore, making a seat available again.

Binary Semaphores: Binary semaphores are like a simple on/off switch, indicating whether a resource is locked or unlocked. It has only two states: 0 (locked) or 1 (unlocked). It's commonly used for situations where only one process can access a resource at a time.

To access a semaphore, we use two operations:

- **wait():** This operation is used when a process wants to access a resource. It's like a customer arriving at a restaurant and trying to get an open table. If there's an open table (counting semaphore is greater than zero), they can take it and use the resource. But if all tables are occupied (semaphore is zero), they must wait until a table becomes available.

- **signal():** This operation is used when a process is done using a resource. It's like a customer finishing their meal and leaving the table. The signal() operation increments the semaphore, indicating that the resource is available again.

In simpler terms, a semaphore is like a counter that keeps track of available resources. Processes can use the wait() operation to request access to a resource, and if it's available, they can use it. Otherwise, they have to wait. When a process is done using a resource, it signals that it's available again using the signal() operation. Semaphores help ensure that resources are used in a coordinated and controlled manner.

RACE CONDITION

Imagine you have two or more people trying to use the same thing at the same time. They don't have a clear rule or order to follow. This can lead to a race condition.

For example, imagine two friends, Alice and Bob, sharing a toy car. Both of them want to play with it and start pushing it at the same time. But since they aren't coordinating their actions, they may push the car in different directions, causing it to go off course or even crash into something.

In a computer system, a race condition is similar. It happens when multiple processes or parts of a program are trying to access and modify the same resource or data simultaneously, without proper coordination. The timing and sequence of these actions become unpredictable, just like in the case of Alice and Bob with the toy car.

When a race condition occurs, the outcome of the program can become incorrect or unexpected. It can lead to errors, crashes, or data corruption because the processes or parts of the program interfere with each other's actions.

To avoid race conditions, programmers use techniques like synchronization and locking mechanisms. These techniques ensure that only one process or part of the program can access the shared resource at a time, preventing conflicts and maintaining proper order.

In simpler terms, a race condition happens when multiple processes or parts of a program try to use the same thing at the same time without following proper rules. Just like in a real race, without coordination, the outcome can be chaotic and unpredictable. To prevent race conditions, programmers use techniques to ensure that processes take turns using shared resources, leading to more reliable and correct program execution.

CRITICAL REGION

Imagine you and your friend both want to play with the same toy. To avoid fights or accidents, you need to take turns and not play with the toy

at the same time. This is similar to what we call a critical region.

In a computer system, a critical region refers to a part of a program where multiple processes or parts of the program may access shared resources, like memory or files. To prevent problems, we want to ensure that only one process can use the shared resource at a time.

To avoid issues like race conditions, we need a way to provide mutual exclusion, which means making sure that only one process can use the shared resource while others are excluded. It's like having a rule that only one person can play with the toy at a time.

The part of the program where the shared resource is accessed is called the critical region or critical section. Think of it as the special area where you have to take turns to use the toy. If we can ensure that no two processes are inside their critical regions simultaneously, we can prevent race conditions.

To make sure processes cooperate correctly and efficiently using shared resources, we need to follow four conditions:

1. No two processes can be in their critical regions at the same time. It's like only one person playing with the toy at a time.
2. We can't assume anything about the speed or number of processors in the computer system.

   When it says, "We can't assume anything about the speed or number of processors in the computer system," it means that we should not rely on specific assumptions about how fast the processors in the system are or how many processors there are.

   - In a computer system, the number of processors (also known as CPUs) can vary, and their speeds can differ as well. Some systems may have multiple processors, while others may have a single processor. Additionally, the speed of processors can vary depending on the hardware configuration.
   - To design a robust and efficient solution, it's important to create algorithms or mechanisms that work reliably, regardless of the number of processors or their speeds. We should not make assumptions about the underlying hardware's specific characteristics to ensure compatibility and effectiveness across different systems.
   - By not assuming anything about the speed or number of processors, we create solutions that are flexible and can work optimally in various computer environments. It allows the operating system or program to adapt to different hardware configurations without relying on fixed assumptions.

3. A process that is not in its critical region should not block or prevent other processes from doing their work.
4. No process should have to wait indefinitely to enter its critical region. We want everyone to get a fair chance to play with the toy.

By following these conditions, we create a system where processes take turns using shared resources, like playing with the toy, ensuring that everyone gets a chance without conflicts or waiting forever.

So, in simpler terms, a critical region is like a special area where processes or parts of a program need to take turns using shared resources. We want to avoid conflicts and make sure everyone gets a fair chance, just like sharing a toy with a friend.

CPU SCHEDULER

In an operating system, scheduling algorithms help manage the allocation of resources, such as the CPU (Central Processing Unit), to different processes or tasks that are running on a computer. Preemptive and non-preemptive scheduling are two types of algorithms used for this purpose.

Non-preemptive Scheduling: Non-preemptive scheduling is like taking turns. Imagine you and your friends are playing a game, and you decide that each person will take a turn to play for a certain amount of time before the next person gets a chance. In non-preemptive scheduling, a process is given control of the CPU and is allowed to run until it completes or voluntarily gives up the CPU.

For example, if you're watching a video on your computer, the non-preemptive scheduling algorithm will let the video play until it finishes or until you decide to pause or stop it. Other processes will have to wait for the video to finish before they can use the CPU.

Preemptive Scheduling: Preemptive scheduling is like interrupting or cutting in line. Imagine you and your friends are waiting in line for a ride, and suddenly someone jumps in front of you without waiting for their turn. Preemptive scheduling allows a process to be interrupted and temporarily paused so that another process can use the CPU.

For example, if you're working on a document and suddenly a high-priority task comes up, the preemptive scheduling algorithm will pause your work and allow the high-priority task to use the CPU. After the high-priority task is done, your work will resume. This way, time-critical or important tasks can be handled promptly.

SCHEDULING CRITERIA

When it comes to choosing a CPU scheduling algorithm, we consider different criteria to compare them. These criteria help us determine which algorithm is the best fit for a particular situation. Here are the criteria:

CPU Utilization: We want to keep the CPU busy and productive as much as possible. CPU utilization refers to how much of the CPU's time is being used for executing processes. We aim for high CPU utilization, ideally close to 100 percent, to ensure efficient utilization of system resources.

Throughput: Throughput measures the number of processes completed per unit of time. It tells us how much work is being done by the CPU. For example, if the CPU can complete 10 processes in a second, the throughput is 10 processes per second. Higher throughput indicates a higher rate of work being accomplished.

Turnaround Time: Turnaround time focuses on the perspective of an individual process. It is the time taken for a process to complete from the moment it is submitted. Turnaround time includes waiting to enter memory, waiting in the ready queue, executing on the CPU, and performing I/O operations. We aim to minimize turnaround time to ensure processes are completed quickly.

Waiting Time: Waiting time refers to the time spent by a process waiting in the ready queue, waiting for its turn to be executed by the CPU. It does not include the time spent executing or performing I/O operations. Minimizing waiting time helps in improving overall process efficiency.

Response Time: In interactive systems, response time is crucial. It measures the time it takes from submitting a request until the first response is produced. For example, in a web application, response time is the time it takes for the system to start showing results to the user. Lower response time leads to better user experience and interactivity.

The goal is to maximize CPU utilization and throughput while minimizing turnaround time, waiting time, and response time. By optimizing these criteria, we can enhance the efficiency and performance of the system.

In simpler terms, the criteria for comparing scheduling algorithms include keeping the CPU busy, completing tasks efficiently, minimizing the time processes take to finish, reducing waiting time, and providing fast response to user requests. The aim is to make the best use of the CPU's time and resources.

SCHEDULING ALGORITHMS

1) FIRST COME FIRST SERVED

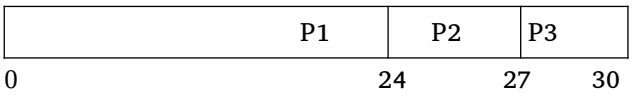
First-Come, First-Served (FCFS) Scheduling is a simple scheduling algorithm used in operating systems. It works on the principle of serving processes in the order they arrive.

In FCFS scheduling, the process that arrives first is given the CPU first and is allowed to run until it completes or voluntarily gives up the CPU. The next process in the queue is then selected and given the CPU, and this process continues until all the processes have been executed.

FCFS scheduling is a non-preemptive scheduling algorithm. Once a process starts running, it is not interrupted until it finishes or voluntarily releases the CPU. This means that if a process with a longer execution time arrives before a process with a shorter execution time, the longer process will continue to run, and the shorter process will have to wait.

Process	**Burst Time**
P1	24
P2	3
P3	3

If the process arrive in the order p1 , p2 , p3 and are served in FCFS order , we get the result shown in the following Gantt chart.



The following table shows the arrival time , start time , waiting time , finish time and Turnaround time of the given processes.

Process	Burst Time	Arrival Time	Start Time	Waiting Time	Finish Time	Turn around Time
p1	24	0	0	0	24	24
p2	3	0	24	24	27	27
p3	3	0	27	27	30	30

The average waiting time is (0 + 24 + 27) / 3 = 17 milliseconds

The average turnaround time = (24 + 27 + 30) /3 = 27milliseconds

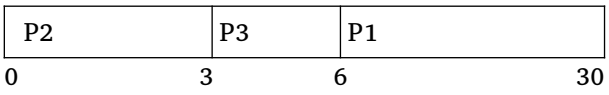
In simpler terms, turnaround time represents the entire duration that a process experiences from its arrival in the system until its completion. It includes the time spent waiting for execution in the ready queue, as well as the time spent actively running on the CPU.

Turnaround Time = Completion Time - Arrival Time

The components of the formula are defined as follows:

- Completion Time: The time at which a process finishes execution.
- Arrival Time: The time at which a process arrives in the system.

Example 2 :- If the processes arrive in the order P2 , P3 , P1 however , the results will be shown in the following Gantt chart



The following table shows the Arrival time , start time , waiting time , finish time and Turn around time of the given processes.

Process	Burst Time	Arrival Time	Start Time	Waiting Time	Finish Time	Turn around Time
p2	3	0	0	0	3	3
p3	3	0	3	3	6	6
p1	24	0	6	6	30	30

The average waiting time = (6 + 0 + 3) / 3 = 3 milliseconds

The average turnaround time = (3 + 6 + 30) /3 = 13 milliseconds

This reduction is substantial. Thus, the average waiting time under a FCFS policy is generally not

minimal, and may vary substantially if the process CPU-burst times vary greatly.

2) SHORTEST JOB SCHEDULING

The shortest job scheduling algorithm is a type of scheduling algorithm that selects the process with the smallest execution time from the list of available processes. This type of scheduling is very intuitive and is often used in batch systems for minimizing job flow time. However, this algorithm does not account for process priority, which can result in priority inversion or indefinite postponement (also known as starvation).

The shortest job scheduling algorithm can be categorized as:

1. Shortest Job Next (SJN) or Shortest Job First (SJF): In SJN or SJF, jobs with the shortest burst time are scheduled first.
2. Shortest Remaining Time First (SRTF): SRTF is the preemptive counterpart of SJF, where the process with the smallest remaining burst time gets scheduled.

The SJF algorithm can be either preemptive or non-preemptive.

In the context of CPU scheduling, the term "preemptive" means that the operating system can take control of the CPU away from a process before it completes execution. This is typically done if another, higher priority process becomes ready to execute or in a round-robin scheduling scenario where each process is given a fair share of the CPU time (quantum).

On the other hand, "non-preemptive" scheduling means that once a process starts execution, it continues until it completes, yields the CPU voluntarily, or blocks for some reason (like I/O operations).

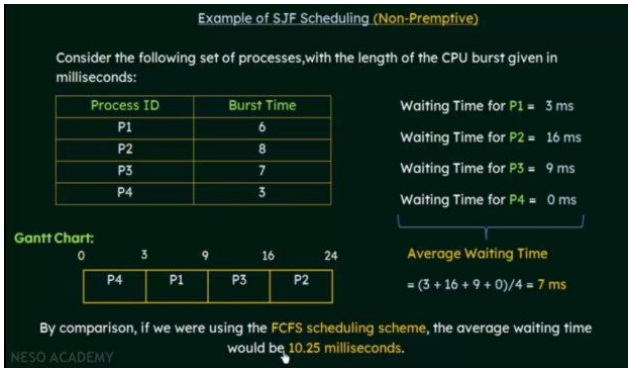
Shortest Job scheduling algorithms can be either preemptive or non-preemptive:

Shortest Job Next (SJN), also known as Shortest Job First (SJF), is a **non-preemptive scheduling** algorithm. Once a job starts execution, it runs to completion. The OS schedules the process with the smallest total execution time first.

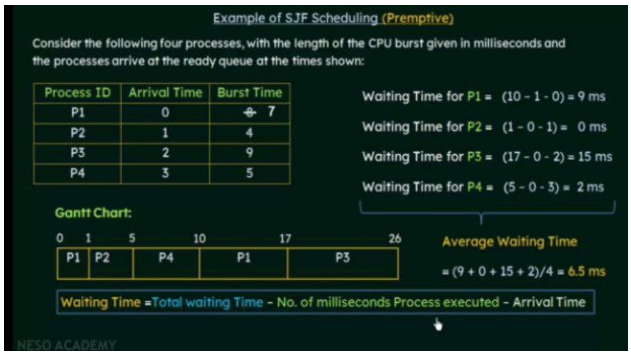
Shortest Remaining Time First (SRTF) is a **preemptive version** of SJF. In SRTF, the OS always ensures that the process with the smallest remaining execution time is the one running. If a new process arrives with a shorter remaining time than the currently executing process, the current process is suspended (preempted), and the CPU is given to the new process.

For example, if process P1 is currently executing with an estimated remaining time of 5 units and a new process P2 arrives with an estimated total execution time of 3 units, the operating system will suspend P1 and start executing P2 because it has a shorter remaining time. This is what we mean by preemptive scheduling in the context of the shortest job scheduling algorithm.

NON-PREEMPTIVE EXAMPLE :-



PREEMPTIVE EXAMPLE :-



3) PRIORITY SCHEDULING ALGORITHM

Priority Scheduling is a method of scheduling processes that is based on priority. In this scheduling method, the scheduler selects the tasks to work as per the priority, which is different from other types of scheduling, for example, simple round robin which treats all processes equally.

Each process is assigned a priority. The process with the highest priority is to be executed first and so on. Processes with the same priority are executed on a first-come, first-served (FCFS) or round-robin basis. Priority can be decided based on memory requirements, time requirements, or any other resource requirement.

There are two types of priority scheduling:

1. Preemptive Priority Scheduling: In Preemptive Priority Scheduling, if a new process arrives with a higher priority than the current executing process, then the current process is put back into the ready queue and the new process is scheduled on the CPU. That means that if a high priority process frequently arrives in the system, the low priority process may starve and never get the CPU for execution.

2. Non-Preemptive Priority Scheduling: In Non-Preemptive Priority Scheduling, a process that is currently running on the CPU will not be removed from the CPU, irrespective of the priority of other processes in the ready queue. Once a process is scheduled, it runs to completion.

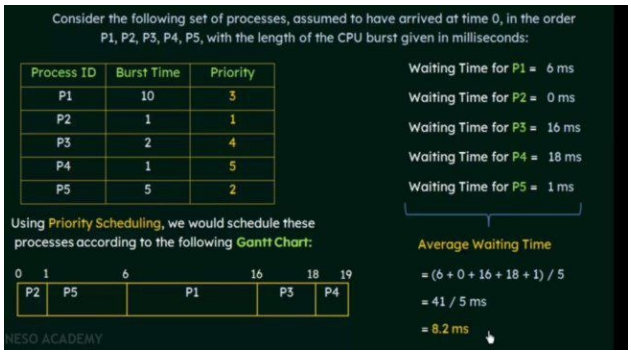
In computer systems, processes are often assigned priorities to determine their order of execution. Priorities are usually represented by a range of numbers, such as 0 to 7 or 0 to 4095. However, there is no universally agreed-upon standard for whether a lower or higher number represents a higher priority.

Different operating systems or scheduling algorithms may adopt different conventions for interpreting priority numbers. Some systems use low numbers (e.g., 0) to indicate high priority, while others use low numbers to represent low priority. This discrepancy can lead to confusion when discussing priorities across different systems or contexts.

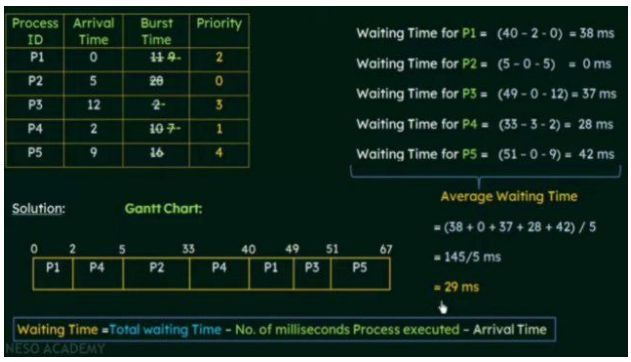
To avoid ambiguity we assumed that low numbers represent high priority. This means that in the context our discussion, a process with a lower priority number would be considered as having a higher priority for execution.

By establishing this convention, we can ensure consistency within its own framework. However, it's important to note that other sources or systems might adopt the opposite convention, where high numbers represent high priority. Therefore, when working with different systems or referring to other materials, it's crucial to understand and follow the specific convention they employ for interpreting priority numbers.

NON-PREEMPTIVE EXAMPLE



PREEMPTIVE EXAMPLE



PROBLEM WITH PRIORITY SCHEDULING

When using priority scheduling algorithms, a significant problem that can occur is indefinite blocking or starvation. This happens when a low-priority process is continuously unable to access the CPU because higher-priority processes keep taking up the available resources. In other words, low-priority processes can be neglected and wait indefinitely for their turn to use the CPU.

To address this issue, a solution called aging is introduced. Aging is a technique where the priority of processes that have been waiting in the system for a long time gradually increases over time.

Here's how aging works in simpler terms:

1. **Waiting Processes** :- When a process is ready to run but is waiting for the CPU, it is considered blocked. Low-priority processes may experience indefinite blocking because higher-priority processes keep getting the CPU.
2. **Gradual Priority Increase :-** With the aging technique, the priority of a waiting process is slowly increased over time. Let's say we have a range of priorities from 0 (low) to 127 (high). We can increment the priority of a waiting process by 1 every 15 minutes.
3. **Aging Effect :-** As time passes, the priority of the waiting process gradually rises. Even a process with an initial priority of 0, which is the lowest priority, would eventually have the highest priority in the system. This means that after a certain period of waiting, the low-priority process will finally get its chance to be executed.
4. **Timeframe:** In this example, it would take no more than 32 hours for a process with an initial priority of 0 to age to a priority 127 process. So, even if a process starts with the lowest priority, it will eventually be given the highest priority after a certain waiting period.

By gradually increasing the priority of waiting processes, the aging technique ensures that low-priority processes are not indefinitely blocked. It provides a fair opportunity for all processes to execute, regardless of their initial priority.

In simpler terms, indefinite blocking occurs when low-priority processes keep waiting for the CPU because higher-priority processes are always given priority. The solution of aging tackles this problem by gradually increasing the priority of waiting processes over time. This ensures that even processes with initially low priority eventually get a chance to execute.

3) ROUND ROBIN ALGORITHM

Round Robin (RR) is a pre-emptive scheduling algorithm that is designed for time-sharing systems. The name "round robin" comes from the principle known as round-robin scheduling, where each task is given an equal and fixed amount of time (also known as quantum) in turn.

In simpler terms, imagine you are at a dinner table with friends and a dish comes out. You each take turns serving yourselves a small bit of the food at a time, and the dish goes around and around the table until everyone is satisfied or the dish is empty. That's the round robin principle - everyone gets a little bit at a time in turns, until all is done.

Here's how the Round Robin scheduling algorithm works:

1. Each process in the system is placed into the ready queue.
2. The first process is removed from the ready queue and is set to run for a specific amount of time, defined by the time quantum.
3. If the process finishes before or when the time quantum expires, it is removed from the CPU and the next process in the ready queue is set to run.
4. If the time quantum expires before the process finishes, the process is moved to

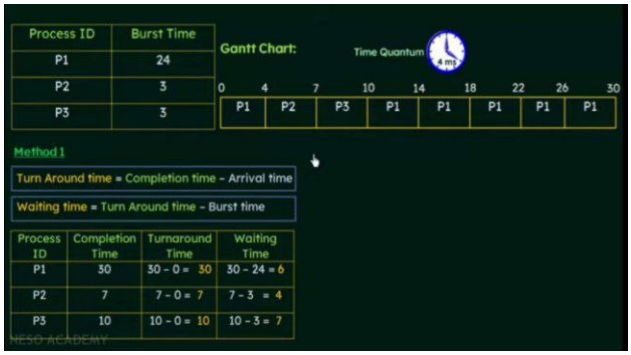
the end of the ready queue, and the next process in the queue is set to run.

5. This cycle repeats until all processes have finished.

One of the major challenges in round robin scheduling is determining the length of the time quantum. If the time quantum is too large, the benefits of the round robin principle (fairness, responsiveness) might be lost. If the time quantum is too small, the system could spend too much time switching between tasks, which decreases overall system efficiency. It's important to strike a balance based on the specific needs of the system.

One advantage of round robin scheduling is its simplicity and fairness - every process gets an equal share of the CPU. However, it doesn't always provide the best average waiting time or turnaround time, especially if the processes vary significantly in their CPU requirements.

EXAMPLE 1 :-



EXAMPLE 2 :-

DEADLOCK

In a computer system, multiple processes may compete for a limited number of resources. These resources can include things like memory space, CPU cycles, files, printers, or other input/output devices. When a process needs a resource, it requests it. If the resource is available, the process can use it. However, if the resource is not available at that moment, the process enters a waiting state until the resource becomes available.

Now, deadlock occurs when multiple processes are waiting for resources that are held by other waiting processes, and as a result, none of the processes can progress. It's like a traffic jam where all the cars are waiting for each other to move, but no one can move because they are all blocked.

To better understand deadlock, let's imagine a scenario with three tape drives and three processes. Each process initially holds one tape drive. Now, if each process requests an additional tape drive and those drives are already held by other processes, we have a deadlock situation. Each process is waiting for a resource (an additional tape drive) that can only be released by one of the other waiting processes. None of the processes can move forward because they are all stuck waiting for something that won't happen.

In general, deadlock occurs when a set of processes are stuck in a state where each process is waiting for an event that can only be caused by another process in the set. The events we are concerned about here are resource acquisition and release.

To deal with deadlocks, operating systems use techniques such as resource allocation algorithms and deadlock detection algorithms. These techniques aim to prevent or resolve deadlocks by carefully managing resource allocation and ensuring that processes can make progress.

Understanding deadlock is important in operating systems because it helps system designers and administrators ensure that processes can run smoothly without getting stuck in situations where they cannot proceed due to resource dependencies.

Under the normal model of operation , a process may utilize a resource in only the following sequence :-

- When a process needs to use a resource, it requests it from the operating system. If the requested resource is available, the process can start using it immediately. However, if the resource is currently being used by another process, the requesting process has to wait until it becomes available.
- Once the process acquires the resource, it can utilize it to perform its designated task. For example, if the resource is a printer, the process can start printing documents. During this phase, the process actively interacts with the resource and performs the necessary operations.
- After completing its work with the resource, the process releases it back to the operating system. By releasing the resource, the process indicates that it no longer needs it and allows other processes to potentially acquire and use it.

DEADLOCK CHARACTERIZATION

Necessary conditions for a deadlock to occur:

1. **Mutual Exclusion:** At least one resource in the system must be held in a non-sharable mode. This means that only one process can use the resource at a time. If another process requests that resource while it is already being used, the requesting process has to wait until the resource becomes available. For example, if a printer is being used by one process, another process must wait until the printer is released.
2. **Hold and Wait:** There must be a situation where a process is holding at least one resource and is waiting to acquire additional resources that are currently being held by other processes. In other words, a process that already holds some resources can request additional resources while still retaining the resources it currently has. This condition can lead to a deadlock when multiple processes are waiting for resources that are held by other waiting processes.
3. **No Preemption**: Resources cannot be forcibly taken away (preempted) from the processes that are currently holding them. A resource can only be released voluntarily by the process that is currently using it after it has completed its task. This condition ensures that a process cannot be interrupted or have its resources forcibly taken away by another process, which could potentially lead to inconsistencies or incorrect results.
4. **Circular Wait**: There must exist a circular chain or cycle of processes, each of which is waiting for a resource held by the next process in the chain. In simpler terms, there is a set of processes, {P0, P1, ..., Pn}, where P0 is waiting for a resource held by P1, P1 is waiting for a resource held by P2, and so on, until Pn is waiting for a resource held by P0. This circular dependency among processes creates a deadlock situation where no process can proceed because each process is waiting for a resource held by another process in the cycle.

It's important to note that all four conditions must hold simultaneously for a deadlock to occur. If any one of these conditions is not satisfied, a deadlock cannot arise. Additionally, the circular-wait condition implies the hold-and-wait condition, so these conditions are not completely independent.

Identifying and understanding these necessary conditions for deadlocks helps in developing strategies and techniques to prevent or resolve deadlock situations in operating systems.

RESOURCE ALLOCATION GRAPH

A resource-allocation graph is a directed graph that helps to describe and analyze deadlock situations in a system. It consists of two sets of vertices: one set represents the active processes in the system (P = {P1, P2, ..., Pn}), and the other

set represents the resource types in the system (R = {R1, R2, ..., Rm}).

In the graph, a directed edge from a process Pi to a resource type Rj, denoted as Pi→Rj, indicates that process Pi has requested an instance of resource type Rj and is currently waiting for that resource. Conversely, a directed edge from a resource type Rj to a process Pi, denoted as Rj→Pi, signifies that an instance of resource type Rj has been allocated to process Pi. Request edges point to the resource type nodes, while assignment edges also indicate the specific instance of the resource that has been assigned.

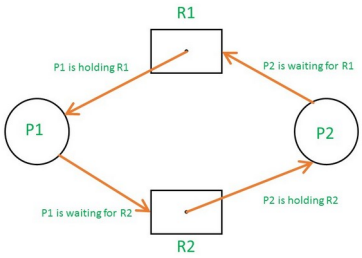
Pictorially, processes are represented as circles, while resource types are represented as squares. Resource instances are depicted as dots within the squares.

The resource-allocation graph helps in identifying deadlock situations based on its structure. Here are some important observations:

1. **No Cycles:** If the resource-allocation graph does not contain any cycles, then there is no deadlock in the system. Each process can proceed with its execution, and resources are being utilized effectively.

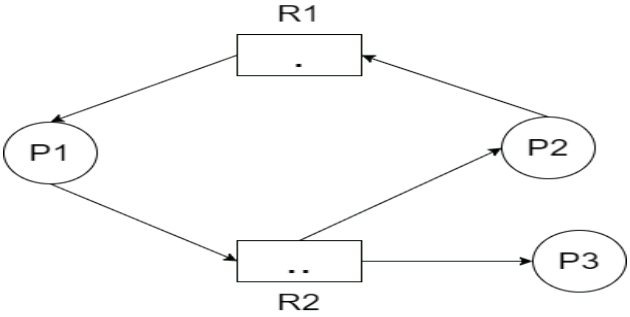
If the graph contains a cycle, it indicates a potential deadlock situation. However, the presence of a cycle is not sufficient to confirm a deadlock. The implications depend on the availability of resource instances.

Single Resource Instance: Imagine you have a toy that only exists in one copy. You can think of it as a unique and special toy that no one else has. In the context of deadlock, a single resource instance means there is only one of that particular resource available in the system. It's like having only one piece of that toy. If there is a cycle involving this type of resource, it means deadlock is certain and the processes involved are stuck and cannot move forward.



SINGLE INSTANCE RESOURCE TYPE WITH DEADLOCK

Multiple Resource Instances: Now imagine you have a toy that many other kids also have. It's like a popular toy that many people can play with at the same time. In the context of deadlock, multiple resource instances mean there are several copies of that resource available in the system. It's like having many pieces of that toy. If there is a cycle involving this type of resource, it suggests the possibility of deadlock, but it's not definite. More investigation is needed to confirm if deadlock has actually happened.



Multi-instance RAG

To summarize:

- Single resource instance: Only one piece or copy of a resource exists. If there is a cycle involving this resource type, deadlock is certain.
- Multiple resource instances: Many pieces or copies of a resource exist. If there is a cycle involving this resource type, it indicates a potential deadlock, but further analysis is needed to confirm if deadlock has occurred.

The resource-allocation graph serves as a useful tool to identify potential deadlock situations by analyzing the presence of cycles and the

availability of resource instances. It helps in understanding the relationship between processes and resources in the system, aiding in deadlock prevention and resolution strategies.

METHODS OF HANDLING A DEAD LOCK

There are three different methods for dealing with the deadlock problem:

Deadlock Prevention: Deadlock prevention aims to ensure that the system never enters a deadlock state. It requires the use of protocols and strategies to eliminate one or more of the necessary conditions for deadlock to occur. By addressing these conditions, the system can prevent deadlocks from happening altogether. This approach typically involves careful resource allocation, avoiding circular wait, and ensuring that resources are not held indefinitely. The goal is to proactively design the system in a way that deadlocks cannot arise.

Deadlock Recovery: Deadlock recovery involves allowing the system to enter a deadlock state and then taking action to recover from it. This approach acknowledges the possibility of deadlock occurrence but provides mechanisms to detect deadlocks and resolve them. Techniques such as resource preemption (taking resources from one process to give to another) or process termination (ending one or more processes involved in the deadlock) can be used to break the deadlock and restore system functionality. Deadlock recovery typically requires additional overhead and may involve sacrificing processes or resources to resolve the deadlock.

Deadlock Ignorance: Some operating systems, including UNIX, adopt the approach of ignoring the problem of deadlocks altogether. They assume that deadlocks are rare events and focus on maximizing system performance and resource utilization.

If a system does not employ deadlock prevention, avoidance, or recovery mechanisms, deadlock situations may occur. In such cases, the system may be in a deadlock state without any means of recognizing or resolving the deadlock. This can lead to system paralysis and a complete halt in progress.

Understanding the different methods for handling deadlocks helps in designing operating systems that strike a balance between preventing deadlocks, recovering from them, or dealing with them when they occur. Each method has its own advantages and trade-offs in terms of system complexity, resource utilization, and overall system performance.

DEADLOCK PREVENTION

As we have seen earlier , for a deadlock to occur , each of the four necessary conditions must hold. By ensuring at least one of these conditions can not hold , we can prevent the occurance of a dead lock , let us elaborate on this approach by examining each of the four necessary conditions separetely.

Mutual Exclusion: Mutually exclusive is a satistical term describing two or more events that can not happen sumultaneously. This condition means that some resources cannot be shared by multiple processes simultaneously. For example, if there is only one printer, only one process can use it at a time. However, sharable resources, such as read-only files, can be used by multiple processes simultaneously without causing deadlocks.

Hold and Wait: To prevent the hold-and-wait condition, we can use different protocols. One approach is to require processes to request and be allocated all the resources they need before they start executing. Another approach allows a process to request resources only when it has none. It can request and use resources but must release them before requesting additional resources. These protocols aim to avoid situations where a process is holding resources and waiting for others, reducing the chance of deadlocks.

No-Preemption: Preemption means taking resources away from processes that are currently holding them. To prevent this condition, a protocol can be used. If a process is waiting for a resource that cannot be immediately allocated, all the resources currently held by that process are

preempted (taken back). The process is then restarted only when it can regain its old resources and acquire the new ones it is requesting.

Circular Wait: This condition can be prevented by imposing a total ordering of resource types and requiring processes to request resources in an increasing order. Each resource type is assigned a unique number, allowing comparison and determination of their order. For example, if we have tape drives, disk drives, and printers, we assign numbers to them (e.g., tape drive = 1, disk drive = 2, printer = 3). Processes must request resources in increasing order of these numbers. This way, circular wait, where processes are waiting for resources in a circular dependency, can be avoided.

Imagine you have different types of toys, such as toy cars, dolls, and building blocks. To prevent a situation where kids are waiting for each other's toys and no one can play, we can assign a number to each type of toy. Let's say toy cars are number 1, dolls are number 2, and building blocks are number 3.

Now, when kids want to play with toys, we ask them to request toys in order of their numbers. This means they can only request toy cars first (number 1), then dolls (number 2), and finally building blocks (number 3). They must follow this order when asking for toys.

By doing this, we make sure that kids are not waiting for toys in a circular way. For example, if a kid is playing with a toy car (number 1) and wants a doll (number 2), they have to wait until the doll is available. Another kid can have the doll, but they cannot ask for a toy car (number 1) because it comes before the doll in the order. This prevents a situation where kids are waiting for each other's toys and no one can play.

By having this resource ordering and making sure kids request toys in the correct order, we can avoid the problem of circular wait and ensure that everyone gets a chance to play with different toys.

By implementing these prevention techniques, we aim to ensure that at least one of the necessary conditions for deadlock does not hold, thus preventing deadlocks from occurring. It involves managing resource allocation, timing of requests, and the order in which resources are requested.

It's important to note that these prevention protocols also have some disadvantages. They can lead to low resource utilization, as some resources may remain allocated but unused for a long time. Additionally, there is a possibility of starvation, where a process with multiple resource requirements may have to wait indefinitely because some of the needed resources are always allocated to other processes.

RECOVERY FROM DEAD LOCK

Process Termination: When a deadlock is detected, there are two ways to break the deadlock. One option is to inform the system operator and let them manually resolve the deadlock. The other option is to let the system automatically recover from the deadlock. One way to do this is by aborting (stopping) one or more processes involved in the deadlock.

- **Abort all deadlocked processes**: In this method, all the processes involved in the deadlock are forcefully stopped. This breaks the deadlock cycle, but it comes with a cost. The processes may have been running for a long time and may have completed some work. Aborting them means discarding their partial results and potentially needing to redo their computations.
- **Abort one process at a time:** Instead of aborting all processes at once, this method aborts one process at a time until the deadlock is resolved. After each process is aborted, a deadlock detection algorithm is invoked to check if the deadlock is still present. This method incurs additional overhead because each process needs to be checked, but it avoids

discarding all the work done by the processes.

Resource Preemption: Another way to resolve deadlocks is by preempting (taking back) resources from processes involved in the deadlock and reallocating them to other processes until the deadlock is broken.

- **Selecting a victim:** To decide which resources and processes to preempt, a strategy is needed. The system must determine which processes can give up their resources without causing further problems.
- **Rollback:** When a resource is preempted from a process, a decision must be made about what to do with that process. It may need to roll back (undo) some of its work to a previous state, as it can no longer continue with the preempted resource.
- **Starvation**: To prevent starvation (a process being constantly preempted and unable to make progress), mechanisms need to be in place to ensure that all processes eventually get their required resources and are not left waiting indefinitely.

These recovery methods aim to break the deadlock and restore system functionality. They involve stopping processes or taking back resources to resolve the circular dependencies causing the deadlock. Both methods have their own considerations in terms of cost, overhead, and potential impact on the progress of processes.

CHAPTER THREE

MEMORY MANAGEMENT

ADDRESS BINDING

Address binding refers to the process of connecting or linking addresses used in a program to specific locations in memory. Address binding is like giving a special home address to things in a computer. It helps the computer know where to find and store information.

Imagine you have a toy box with different toys inside. Each toy has a special place where it belongs. For example, the toy car goes in one corner, the doll goes in another corner, and the building blocks go in a different corner.

Address binding in a computer is similar. When a program runs, it needs to find and store information in the computer's memory. The computer gives each piece of information a special address, just like each toy has a special place in the toy box.

Address binding is important because it helps the computer keep track of where things are stored. When the program needs to find information, it can go directly to the correct address to get it. And when the program wants to save new information, it knows where to store it. Think of it as giving each thing in the computer its own home address, so the computer can find and keep track of everything easily.

In the context of address binding, the term "binding" refers to the process of connecting or associating a specific address with a particular entity in a computer system. It establishes a relationship between a symbolic representation of an address (such as a variable name in a program) and the actual physical or virtual memory location where the data or instructions are stored.

Binding ensures that when a program needs to access information or instructions, it knows exactly where to find them in memory. It "binds" or links the symbolic representation of an address

to the actual location in memory where the data resides.

The purpose of binding is to enable efficient and reliable access to data and instructions during program execution. It helps the computer system accurately map and manage the memory resources required by programs, ensuring that the right data is retrieved from the right place in memory.

In simpler terms, binding is like creating a connection or a link between a name (symbolic representation) and the place (memory address) where something is stored. It helps the computer system know where to find and store information, making it easier to access and manage data and instructions.

When a program is being prepared to run on a computer, its instructions and data need to be assigned specific memory addresses. The process of assigning these addresses is called binding. There are three common stages where binding can take place:

Compile Time: If the computer knows exactly where the program will be stored in memory before it even runs, the compiler (a special program) can generate instructions that directly reference those memory addresses. It's like creating a road map with fixed destinations.

Load Time: If the memory addresses are not known at compile time, the compiler generates instructions that can be adjusted later. These instructions are called relocatable code because they can be moved around. During the loading process, the final binding of memory addresses takes place. It's like waiting until you know the exact location of your destination before getting directions.

Let's simplify it:

When a program is being prepared to run on a computer, it needs to know where to find its instructions and data in memory. Sometimes, the program doesn't know the exact memory addresses in advance, so it uses special instructions that can be adjusted later. We call these instructions "relocatable code."

Imagine you're going on a trip, but you don't know the exact address of your destination yet. You can still pack your bag with things you might need and get ready to go. Similarly, the program prepares its instructions and data without knowing the exact memory addresses.

When it's time to start the program, the computer finds a suitable place in memory to load it. During this loading process, the computer figures out the final memory addresses where the program will be stored. It's like finding the exact address of your destination before you start your journey.

Once the program's instructions and data are loaded into memory and their final memory addresses are determined, the program is ready to run. It can access its instructions and data at the correct memory locations and perform its tasks.

In simple terms, load time binding is like packing your bag without knowing the exact address, but when you're about to start your trip, you find out the exact address and can start moving. Similarly, the program prepares its instructions without knowing the exact memory addresses, but when it's time to run, the computer figures out where to put the program in memory.

Execution Time: In some cases, a program might need to move around in memory while it's running. This could be because the program is too big to fit entirely in one memory segment or because the computer needs to optimize the memory usage. In these situations, binding is delayed until runtime, when the program is actually being executed. It's like deciding where to go while you're on the road.

Let's simplify it

Imagine you're playing with building blocks, and you have a big tower you're building. But as you keep adding more blocks, the tower gets too tall to fit on just one table. So, you decide to move

parts of the tower to other tables to make room for everything.

In a computer program, sometimes the program itself is like that big tower. It's too big to fit entirely in one part of memory. So, while the program is running, it might need to move around in memory to find enough space. It's like the program is deciding where to go while it's already running.

When a program moves around in memory during execution, the binding of addresses is delayed until that moment. The computer figures out the memory addresses for the program while it's running. It's like making decisions about where to put the blocks while you're already playing with them.

This can happen for different reasons. Maybe the program is really large and needs to spread out across different parts of memory. Or maybe the computer wants to optimize how it uses memory to make things faster. In these cases, the program decides where to go in memory as it goes along.

In simple terms, execution time binding is like rearranging your building blocks as you play with them because your tower is getting too tall. Similarly, a program might need to move around in memory while it's running because it's too big or the computer wants to be efficient.

The purpose of these different binding stages is to make sure that the program knows where to find its instructions and data in memory. It allows the computer to efficiently execute the program and manage memory resources effectively.

DYNAMIC LOADING

When we want to use a computer program, we usually load the whole program into memory before we start using it. But what if the program is really big and has many parts that we might not need right away? It would take up a lot of memory space unnecessarily.

To solve this problem, we can use a technique called dynamic loading. With dynamic loading, we don't load the entire program into memory at once. Instead, we keep parts of the program on the disk until they are needed.

Here's how it works:

1. The main part of the program is loaded into memory and starts running.
2. As the program runs, it might come across a part, called a routine, that it needs to use. Before loading the routine, the program checks if it's already in memory.
3. If the routine is not in memory, a special program called a relocatable linking loader is called. This program loads the needed routine into memory from the disk.
4. Once the routine is loaded, the main program can use it as needed.
5. Unused routines stay on the disk and are never loaded into memory, saving memory space.

The advantage of dynamic loading is that we only load the parts of the program that we actually need, when we need them. This makes better use of memory because we don't waste memory space on parts of the program that are not being used.

It's important to note that dynamic loading doesn't require special support from the operating system. It's up to the programmers and users to design their programs in a way that takes advantage of dynamic loading. They need to organize their programs into separate routines and use the relocatable linking loader when necessary.

In simpler terms, dynamic loading is like only taking out the toys or tools you need from a big box instead of taking everything out at once. It saves space and makes things more efficient because you only load what you need when you need it.

DYNAMIC LINKING

When we write computer programs, we often use libraries that have pre-existing code for common tasks. These libraries are like collections of useful

functions that we can use in our programs, just like using tools to build something.

In most operating systems, when we finish writing our program, we combine it with the library code to make a complete program. This is called static linking. It's like putting all the tools we need inside our project, even if other people are using the same tools for their projects.

But dynamic linking works differently. Instead of including the library code in our program, we wait until the program is actually being used to link it with the library code. It's like borrowing tools from a shared toolbox when we need them.

Here's how dynamic linking works :-

1. When we finish writing our program, we tell the computer that we want to use certain functions from the library, like asking for specific tools.
2. When the program is run, the computer loads it into memory, just like opening a book to read.
3. Now, our program needs to use the functions from the library. Instead of having its own copy of the library code, the program uses a reference to find the functions it needs.
4. The computer then finds the actual library code and connects it with our program, like getting the right tools from the toolbox and putting them in our hands.
5. Finally, our program can use the library functions as if they were part of the program itself, just like using the borrowed tools to do our work.

Let's simplify it

Imagine you have a big box full of toys, and your friends also have the same box. Each toy is like a function that helps you do something special, like a magic wand or a super-fast race car.

Usually, when you want to play with a toy, you take it out of your box and keep it with you while you play. That's like static linking, where each program has its own copy of the toy (function) it needs.

But with dynamic linking, it's different. Instead of taking the toy out of the box, you keep it in the box and tell your friends that you want to use a specific toy. When you want to play with it, you borrow it from the box and use it, and then put it back when you're done.

This way, you and your friends can share the toys in the box without needing to have multiple copies. It saves space and makes it easier to find the toy you want. That's how dynamic linking works with programs and libraries. Programs can borrow functions from a shared library when they need them and return them when they're done.

So, dynamic linking is like sharing a big box of toys with your friends. You borrow the toys (functions) you need from the box when you want to play with them and put them back when you're finished. It helps save space and makes it easier for everyone to find and use the toys they need.

The cool thing about dynamic linking is that many programs can share the same library code, like sharing tools with friends. This saves memory because we don't need to copy the entire library for every program. Without dynamic linking, each program would have its own copy of the library code, and that would be wasteful.

Dynamic linking is often used with libraries that have common functions that many programs use. Instead of making a copy of the library for each program, we share it among them. When a program needs a specific function, it gets it from the shared library, just like borrowing a tool from a shared toolbox and returning it when we're done.

OVERLAYS

Imagine you have a very big puzzle with many pieces, but your table is too small to fit all the pieces at once. However, you still want to solve the puzzle and see the complete picture. To tackle this challenge, you divide the puzzle into smaller sections, and you place one section on

the table at a time. As you work on one section and need a different section, you swap the current section with the new one. This way, you can solve the entire puzzle using the limited space on your table.

Now, let's relate this to computer programs. In older computer systems with limited memory, programs could be larger than what could fit in the available memory at once. This is where overlays come into play. Overlays are a technique used to divide a program into smaller logical sections or overlays.

Here's how overlays work in simpler terms:

1. **Dividing the Program:** Just like dividing a puzzle, a program is divided into smaller logical sections based on its functionality. Each section contains a specific part of the program's code.
2. **Swapping Sections:** The computer can only load one section into memory at a time due to limited memory capacity. As the program execution progresses and reaches a point where a different section is needed, the current section is swapped out of memory, and the new section is loaded.
3. **Manual Management:** Overlays require manual programming and management. The programmer needs to determine which sections are needed at different points in the program and handle the swapping of sections in and out of memory.
4. **Limited Memory Utilization:** Overlays are used when the entire program cannot fit into the available memory all at once. By dividing the program into smaller sections and swapping them as needed, memory can be effectively utilized.

So, overlays are like dividing a large puzzle into smaller sections and swapping them on and off the table as you solve it. Similarly, in computer programs, overlays divide a program into smaller sections, load one section at a time into memory, and swap them as the program progresses. It's a way to manage memory limitations and work with programs that are larger than the available memory capacity.

LOGICAL VERSUS PHYSICAL ADDRESS SPACE

Logical Address: Imagine you have a virtual map that helps you find your favorite toys in a large toy store. The map has numbers and symbols that guide you to where each toy is located. This virtual map is like a logical address.

In computer systems, a logical address is a virtual address generated by the CPU while a program is running. It doesn't physically exist in the computer's memory. It's like a reference or code that helps the computer know where to find specific information. The set of all logical addresses generated by a program is called the logical address space.

Physical Address: Now, imagine you're in a real toy store where the toys are physically placed on different shelves. Each toy has its own specific location on a shelf. These physical locations represent the physical addresses.

In computer systems, a physical address is the actual location of data in the computer's memory. It's like the specific shelf and spot where a toy is placed in the toy store. While the program generates logical addresses, the physical addresses are computed by a hardware device called the Memory Management Unit (MMU). The physical address space is the collection of all physical addresses corresponding to the logical addresses.

To summarize :-

- **Logical Address:** It's like a virtual map that helps you find toys in a toy store. It's a reference used by the computer to locate information while a program is running.
- **Physical Address:** It's like the actual location of toys on shelves in a toy store. It represents the physical memory locations where data is stored. The MMU computes the physical address corresponding to a logical address.

In simpler terms, a logical address is like a map guiding the computer to find information, and a physical address is the actual location where the information is stored in the computer's memory.

SWAPPING

When a computer is running many programs at the same time, it needs to manage the memory efficiently. Swapping is a technique used to temporarily move a program out of memory and store it on a special place called a "backing store," like a fast disk. The program can then be brought back into memory later to continue running.

Here's how swapping works:

1. Imagine a computer that can run multiple programs simultaneously, like a round-robin game where everyone gets a turn.
2. Each program gets a limited amount of time called a "quantum" to run on the CPU. When the quantum is up, the program is temporarily moved out of memory.
3. The memory manager takes the program that just finished and swaps it out to the backing store, freeing up memory space.
4. Meanwhile, the CPU scheduler gives a turn to another program that is already in memory.
5. The swapped-out program can stay in the backing store until it's time for its turn again. Then it's swapped back into memory for continued execution.
6. This swapping process continues, with programs taking turns in memory while others are swapped out.

Swapping is also used in priority-based scheduling, where higher-priority programs get priority for CPU time. If a high-priority program needs to run, the memory manager can swap out a lower-priority program to make room for it. When the high-priority program is done, the lower-priority program can be swapped back in.

Swapping requires a special place called a backing store, usually a fast disk, to store the programs temporarily. The computer keeps track of which programs are in memory and which are in the backing store. When the CPU scheduler wants to run a program, it checks if it's in memory. If not, and there's no free memory space, it swaps out a program from memory and swaps in the desired program.

It's important to note that swapping takes some time because programs need to be moved in and out of memory. This switching time is called the context switch time and can affect the overall performance of the system.

Contiguous Allocation

In a computer's main memory, we need to allocate space for both the operating system and user processes. One way to do this is by using contiguous allocation, which means that each process is placed in a single, uninterrupted section of memory.

To manage this allocation, the memory is divided into two partitions: one for the operating system and one for the user processes. In our discussion, let's assume that the operating system resides in the low memory area.

In contiguous allocation, each process is assigned a specific section of memory that is large enough to accommodate its needs. To keep track of where each process is located, we use two special registers: the base register and the limit register.

The base register points to the smallest memory address of a process, while the limit register indicates the size of that process's memory section. Together, these registers define the boundaries of the process in memory.

For example, let's say we have three processes: Process A, Process B, and Process C. Each process is assigned a contiguous section of memory. The base and limit registers for each process specify the starting address and the size of their memory sections.

Using these registers, the computer can easily locate and access the instructions and data for each process when needed.

Contiguous allocation allows multiple processes to coexist in memory at the same time, making efficient use of the available memory space. However, it also means that processes need to be assigned continuous blocks of memory, which can limit the flexibility and efficiency of memory utilization.

Imagine you have a big room called the computer's memory. In this room, you need to keep the operating system and different programs. To do this, you divide the room into two parts: one for the operating system and one for the programs.

Now, let's focus on the part for the programs. Each program needs its own space to stay in the memory. In contiguous allocation, we give each program a special area that is like its own little house. This house is a single piece of space in the memory, without any gaps in between.

To remember where each program's house is, we use two special signs. One sign tells us the starting address of the house, and the other sign tells us how big the house is. With these signs, we can find each program's house easily.

For example, let's say we have three programs: Program A, Program B, and Program C. Each program has its own house in the memory. The signs tell us where each house starts and how much space it takes.

This way, when we need to run a program or do something with it, we know exactly where to find it in the memory.

Contiguous allocation allows many programs to live in the memory together, making good use of the available space. But it also means that programs need to have one continuous block of space, which can sometimes limit how efficiently we can use the memory.

Single Partition Allocation

Imagine that the computer's memory is like a big playground. In this playground, we have two areas: one area for the operating system and one area for the user programs.

The operating system is like the supervisor of the playground, and it wants to make sure that everything is running smoothly. It stays in the low part of the playground, like the ground floor of a building. The user programs, on the other hand, stay in the high part of the playground, like the upper floors of a building.

To protect the operating system from any changes or problems caused by the user programs, we use a special system. We have two special guards called the relocation register and the limit register. The relocation register tells us where the operating system starts in the memory, and the limit register tells us how much space it takes.

Whenever a user program wants to access the memory, we check if the address it wants to use is within the allowed range set by the relocation and limit registers. If it is, we allow the program to access that part of the memory. This way, we make sure that the program can't go beyond its allowed area and cause any trouble for the operating system or other programs.

When a new program is selected to run by the computer, we have a person called the dispatcher who prepares everything. The dispatcher sets the relocation and limit registers to the correct values for that program. This helps to protect the program and keep it within its allocated memory space.

One important thing to note is that this system allows the operating system to change its size dynamically. This means that the size of the operating system can grow or shrink as needed. For example, if a certain part of the operating system is not being used, it can be removed from the memory to make space for other things. This flexibility helps to optimize the memory usage and make the best out of it.

So, in simple terms, we use the relocation and limit registers to make sure that the operating system and user programs stay in their designated areas in the memory. This protects them from causing problems for each other. The system also allows the operating system to change its size dynamically to make the most efficient use of the memory.

Let's simplify it

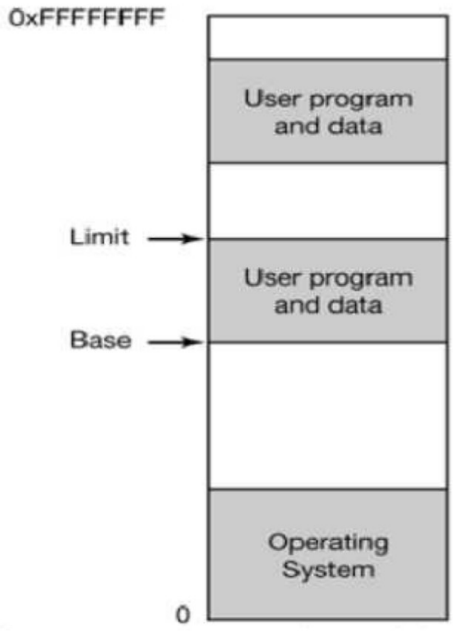
Imagine you're playing in a big playground. There are different areas for different things. One area is for the grown-ups who make sure everything runs smoothly, like the playground supervisors. They stay on the ground floor. Another area is for kids like you, who play on the upper floors.

To keep things organized, we have special guards called relocation and limit registers. These guards help make sure everyone stays in their own play area. When a kid wants to play, we check if they are going to the right part of the playground. The relocation register tells us where the grown-ups' area starts, and the limit register tells us how much space they have.

When a new kid comes to play, we have someone called the dispatcher who sets everything up. They make sure the relocation and limit registers are set correctly for each kid. This helps keep everyone safe and prevents them from going where they shouldn't.

The cool thing is that the grown-ups' area can change its size if needed. If they have extra space that's not being used, they can make it smaller and let the kids use it. This way, everyone gets to play and have fun!

So, in simple words, the relocation and limit registers are like guards in the playground. They make sure the grown-ups and kids stay in their own areas. And the grown-ups can change the size of their area to share it with the kids.



Multiple Partition Allocation

In a computer system, we have a limited amount of memory available to store programs and data. To make the best use of this memory, we divide it into fixed-sized sections called partitions. Each partition can hold one program or process.

When a new process needs memory, the operating system looks for a partition that is big enough to accommodate it. If there is an available partition, the process is loaded into it. When a process finishes or terminates, the partition becomes available again for another process to use.

Initially, all the memory is considered as one large block called a hole. As processes come and go, the memory is divided into different-sized holes. When a process arrives and needs memory, the operating system searches for a hole that is large enough to hold it. If the hole is larger than needed, it is divided into two parts: one part is allocated to the arriving process, and the other part becomes a new hole. When a process finishes, its memory block is released and returned to the set of holes. If the newly released memory is adjacent to other holes, they can be merged to form a larger hole.

When we have different holes (spaces) in the memory, we need to decide which hole to use

for a new program. There are three ways to choose a hole:

First-fit: Imagine you have a set of blocks (holes) in different sizes. With first-fit, you pick the first block that is big enough to fit the program. You don't keep searching for a better fit once you find a suitable block.

Best-fit: In this case, you want to find the smallest block that can accommodate the program. You may need to look through all the blocks to find the smallest one, unless they are arranged from smallest to largest. The goal is to minimize the amount of leftover space (empty space).

Worst-fit: Here, you choose the largest available block, regardless of whether it is bigger than what the program needs. This strategy can create bigger leftover space, which might be useful for future programs.

So, in summary:

- First-fit: Choose the first suitable hole.
- Best-fit: Pick the smallest hole that fits.
- Worst-fit: Select the biggest hole available.

DIFFERENCE BETWEEN SINGLE AND MULTIPLE PARTITION ALLOCATION

Let's imagine we have a room (which represents the computer's memory) where we want to accommodate different toys (which represent different processes or programs running on the computer).

In single partition allocation, the room is divided into two sections: one section for the toys that belong to the room itself (like the room's own toys) and another section for the toys that are brought in by children (like the toys of the user processes). The important thing to note here is that there is only one dedicated space for the children's toys, and they need to share that space.

On the other hand, in multiple partition allocation, the room is divided into multiple sections, each with its own designated space for toys. This means that each child (or process) can have their own separate space to keep their toys. They don't have to share the same space with other children.

So, the main difference between single and multiple partition allocation is the number of dedicated spaces available for the toys (or processes). In single partition allocation, there is only one shared space, while in multiple partition allocation, there are multiple separate spaces.

FRAGMENTATION IN OPERATING SYSTEM

Fragmentation refers to the phenomenon where available memory in a computer system is divided into small, non-contiguous pieces, making it difficult to allocate large enough memory blocks for processes or programs. There are two types of fragmentation: external fragmentation and internal fragmentation.

External fragmentation occurs when the free memory space is broken into small, scattered pieces, even though the total amount of free memory may be sufficient to satisfy a request. It is like having a jigsaw puzzle with many small pieces that don't fit together neatly. External fragmentation makes it challenging to find a large enough continuous block of memory for a process, leading to inefficient memory utilization.

Internal fragmentation, on the other hand, happens within a partition allocated to a process. It occurs when the allocated memory is slightly larger than what the process actually needs, leaving behind unused and wasted memory within that partition. It's like having a container that is slightly bigger than what you need, resulting in some empty space inside.

Fragmentation occurs as processes are loaded and removed from memory, leading to small gaps or wasted space.

To address external fragmentation, one solution is **compaction,** which rearranges the memory contents to place all the free memory together in one large block. This helps reduce the scattered

small holes of free memory, making it easier to find contiguous blocks for new processes.

For internal fragmentation, **allocating slightly** larger memory blocks than requested helps minimize the overhead of managing very small holes of free memory. This means that there may be some unused memory within a partition, but the difference between the allocated memory and the requested memory is kept relatively small.

In simpler terms, fragmentation is like having scattered puzzle pieces or empty spaces in a container. It makes it difficult to find a big enough space for something or results in some wasted space. Compaction helps put the scattered memory pieces together, while allocating slightly more memory than needed minimizes wasted space inside memory partitions.

Solution to Fragmentation

- Compaction (already seen it)
- Paging
- Segmentation

PAGING

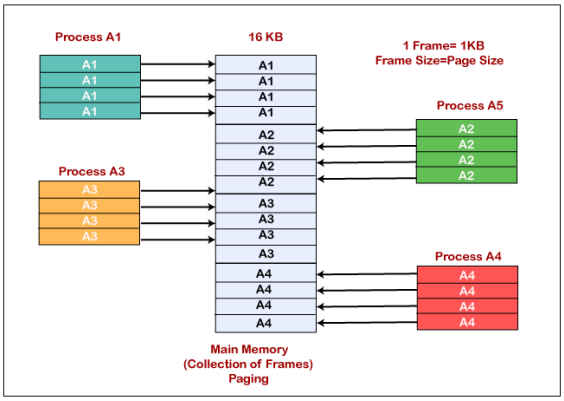
Imagine you have a big box of Lego blocks. The blocks are of the same size, and you also have a special shelf to store them. Now, when you want to build something with the blocks, you don't need to have all the blocks out on the table at once. Instead, you can keep some blocks on the shelf and bring them out as you need them. This way, you can work with a limited number of blocks at a time without cluttering up your workspace.

Paging is a similar concept in a computer. Instead of having all the parts of a program in memory at once, the program's memory is divided into fixed-sized blocks called pages, just like the Lego blocks. The physical memory of the computer is also divided into blocks called frames, which are the same size as the pages.

When a program is running, its pages are loaded into any available frames in the computer's memory. It's like taking out the necessary Lego blocks from the shelf and placing them on the table when you need them.

To keep track of where each page is located in memory, there is a special table called the page table. Think of it as a map that tells you which pages are stored in which frames. When the program needs to access a specific part of its memory, the CPU uses the page number and an offset to look up the page's location in the page table. This tells the computer's memory unit where to find the actual data in physical memory.

So, in simpler terms, paging is like storing Lego blocks on a shelf and bringing them out as needed. It allows programs to use memory that is not necessarily located in one continuous block. Instead, the memory is divided into fixed-sized pages, and the computer keeps track of where each page is stored using a page table.



Imagine you have a big library with lots of books. Each bookshelf in the library can hold a fixed number of books, let's say 8 books per shelf. The library also has a catalog that tells you which book is on which shelf.

Now, when you want to read a book, you don't need to bring the entire library to your reading table. Instead, you can look up the catalog to find the shelf number where the book is located, and then you can take that book from the shelf and read it.

In computer systems, paging works in a similar way. The computer's memory is like the library, and the books are the pages of a program or process. Each page has a fixed size, let's say 4 pages per book. The computer also has a catalog called the page table that tells you which page is stored in which part of memory.

When a program is running, it doesn't need to have all its pages in memory at once. Just like you don't need all the books from the library on your reading table. Instead, the program's pages are loaded into available slots in the computer's memory. Each slot can hold one page, just like a bookshelf can hold one book.

To access a specific part of the program's memory, the CPU uses a logical address. This logical address consists of a page number and an offset. The page number tells you which page the data is on, just like the catalog tells you which shelf the book is on. The offset tells you the specific location within the page, just like the page number on the book tells you the specific chapter or page.

So, in simpler terms, paging is like having a library with lots of books, where each book is divided into fixed-sized pages. The computer's memory is like the bookshelves, and the page table is like the catalog that tells you where each page is stored. When a program needs to access its memory, the CPU uses a logical address, which consists of a page number and an offset, to find the corresponding page in memory.

SEGMENTATION

Imagine you have a big toy box filled with different toys. When you want to play with a specific toy, you don't think about its position in the box as a linear order of toys. Instead, you think of the toys as separate groups or segments based on their type, such as dolls, cars, or building blocks.

In a computer, segmentation is a way of organizing memory that matches this way of thinking. Instead of considering memory as a long sequence of bytes, we divide it into different segments or groups. Each segment can have a specific purpose, like storing instructions or data.

When a program runs, it is divided into segments, just like the toys are divided into groups. Each segment has a segment number that identifies it and an offset that specifies the position within that segment. This combination of segment number and offset is used to access a specific byte in memory.

To keep track of where each segment is in the physical memory, the computer uses a table called a **segment table.** This table contains pairs of information: the base address and the limit of each segment. The base address tells the computer where a segment starts in memory, and the limit specifies the size of the segment.

When a program wants to access a specific byte in memory, it uses the segment number and offset to find the corresponding segment in the segment table. From there, the computer can calculate the physical address of the desired byte by adding the offset to the base address of the segment.

Segmentation allows programs to have different-sized segments and provides a more flexible way of managing memory. It aligns with how users and programmers think about memory, as separate and distinct groups rather than a linear sequence of bytes.

Example :-

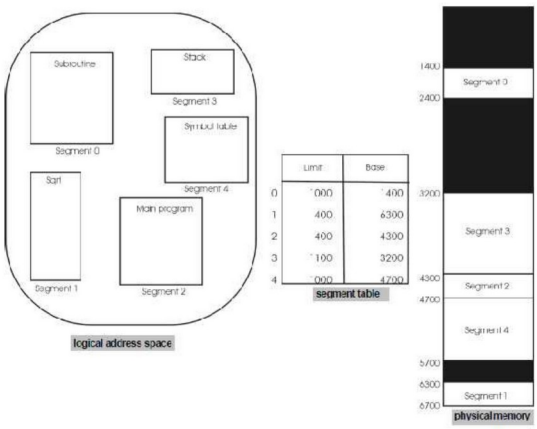


Figure 3.6: Example of segmentation.

For example, let's say you want to find byte number 53 in segment number 2. You look at the segment table, which tells you that segment 2 starts at memory location 4300 (base address) and is 400 bytes long (limit). To find byte 53, you add 53 to the starting location (4300 + 53), and you get the memory location 4353.

Similarly, if you want to find byte number 852 in segment number 3, you look at the segment table and find that segment 3 starts at memory location 3200. You add 852 to 3200, and you get the memory location 4052.

But it's important to be careful! If you try to access a byte that is outside the limit of a segment, like byte 1222 in segment 0, it would cause a problem. The segment table tells you that segment 0 is only 1000 bytes long, so trying to access byte 1222 would be an error, and the computer would notify the operating system.

So, segmentation is like having different boxes for storing information, and each box has a number. To find something inside a box, you need to know the box number and the position of what you're looking for.

In the example provided, the segment number, offset, and base address can be identified as follows :-

- **Segment number:** The segment number is the identifier for a specific segment or box of memory. In the example, segment numbers are mentioned as "segment 2," "segment 3," and "segment 0." Each segment represents a different area of memory.
- **Offset:** The offset refers to the position or location of a specific byte within a segment. It tells us how far into the segment we need to go to find the desired byte. For instance, in the example, byte numbers like "53," "852," and "1222" represent the offset.
- **Base address:** The base address is the starting memory location of a segment. It tells us where the segment begins in physical memory. In the example, the base addresses for segments are mentioned as "location 4300" and "3200."

To summarize:

- Segment number: Identifies the specific segment or box of memory.
- Offset: Represents the position or location of a byte within the segment.
- Base address: Marks the starting memory location of a segment.

Difference between paging and segmentation

Segmentation and paging are two different memory management techniques used in operating systems. Here's a comparison between the two:

Segmentation:

- In segmentation, the logical address space of a process is divided into variable-sized segments, each representing a different part of the program (such as code segment, data segment, stack segment, etc.).
- Each segment has its own starting address and length.
- Segments can vary in size and can be allocated and deallocated dynamically as needed.
- Segmentation provides a flexible memory allocation scheme that allows processes to grow or shrink based on their memory requirements.
- It allows for logical separation of different parts of a program and provides protection between segments.
- Segmentation may lead to external fragmentation, where free memory becomes scattered in small chunks, making it harder to allocate contiguous blocks of memory.

Paging:

- In paging, the logical address space of a process is divided into fixed-sized blocks called pages.
- Physical memory is divided into fixed-sized blocks called frames.
- Pages and frames are of the same size, and each page can be mapped to any available frame in physical memory.

- Paging provides a uniform and fixed-size memory allocation scheme.
- It eliminates external fragmentation because each page can be placed anywhere in physical memory as long as a free frame is available.
- Paging introduces the concept of a page table, which is used to map logical addresses to physical addresses.
- Paging requires hardware support, specifically a memory management unit (MMU) that handles the translation of logical addresses to physical addresses.

In summary, segmentation divides the logical address space into variable-sized segments, while paging divides it into fixed-sized pages. Segmentation allows for flexible memory allocation, while paging provides a more uniform memory allocation scheme. Both techniques have their advantages and trade-offs, and their choice depends on factors such as the system requirements and hardware capabilities.

VIRTUAL MEMORY

Imagine you have a small table in your room where you can do your homework. But sometimes your homework is too big to fit on the table. So what can you do? You can use a special trick called virtual memory.

Virtual memory is like having a magical bookshelf that can hold all your books. When you need to work on a big project, you take out a few pages from the bookshelf and put them on your table. These pages represent the parts of your project that you are currently using.

If you need more pages, but your table is full, you can put some of the pages back on the bookshelf and take out new ones. This way, you always have enough pages on your table to do your work, even if the project is really big.

Virtual memory helps your computer do the same thing. It has a special space on the hard disk called the bookshelf, which can store parts of programs and data that are not currently being used. When the computer needs to work on something, it takes out the necessary parts from the bookshelf and puts them in its memory (which is like your table). If it needs more space, it can swap out some parts to make room for new ones.

By using virtual memory, the computer can run big programs and work on large projects, even if it doesn't have a lot of memory. It can easily switch between different tasks and use its resources more efficiently.

So, virtual memory is like a magical bookshelf that helps the computer handle big tasks by storing parts of them when they are not needed, and bringing them back when they are. It's a clever way to make the most of limited space and keep everything running smoothly.

Here's a simplified explanation of virtual memory:

In a computer, programs and data are stored in memory for the CPU to access and execute. However, the physical memory of a computer is limited, and sometimes programs are too big to fit entirely in memory. Virtual memory solves this problem by using a portion of the hard disk as an extension of the physical memory.

Virtual memory works by dividing programs into smaller chunks called pages. These pages are loaded into the physical memory only when they are needed. When a program tries to access a page that is not currently in the physical memory, a process called demand paging is used. The operating system brings the required page from the hard disk into the physical memory and then allows the program to access it.

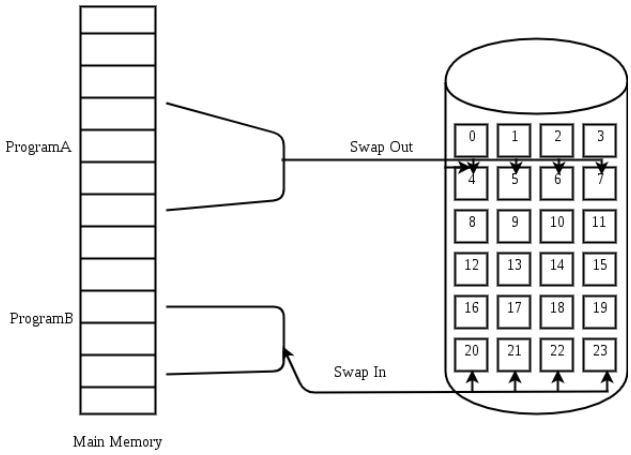
This approach has several advantages:

1. **Larger Programs:** With virtual memory, programs can be larger than the physical memory. They can be divided into pages, and only the necessary pages are loaded into memory as needed.
2. **Efficient Memory Usage**: Virtual memory allows multiple programs to share the

same physical memory. Each program is given a portion of the memory, and as pages are swapped in and out, the physical memory is used more efficiently.

3. **Increased Performance:** Virtual memory reduces the need for constant swapping of entire programs in and out of memory. This improves overall performance and allows more programs to run simultaneously, increasing the utilization of the CPU.

4. **Flexibility:** Certain parts of a program, like error handling routines or rarely used features, don't need to be loaded into memory until they are actually required. This saves memory space and improves efficiency.



In simpler terms, virtual memory is like having a big workspace where you can keep all your books and papers. You can't fit all of them on your desk at once, so you put some on a nearby shelf. When you need a specific book, you take it from the shelf and place it on your desk. If you need another book that is not on your desk, you put one of the books back on the shelf to make room. This way, you can work with many books without cluttering your desk. Virtual memory works in a similar way, allowing computers to handle large programs without running out of memory.

DEMAND PAGING

Imagine you have a big box filled with your toys, but you don't have enough space to take them all out and play with them at once. So instead, you decide to take out only the toys you want to play with right now, and leave the rest in the box.

Demand paging is similar to this idea. When a program is running on a computer, instead of loading the entire program into memory at once, the computer only brings in the parts of the program that are needed at the moment. It's like taking out the specific pages of a book that you want to read, rather than carrying the entire book with you.

By using demand paging, the computer can save a lot of memory space because it doesn't need to load everything at once. It only brings in the pages of the program that the user wants to use. This helps the computer work more efficiently and use its resources wisely.

Page Fault :- Imagine you have a special bookshelf where you keep all your books. When you want to read a book, you take it from the bookshelf and start reading. But what if the book you want is not on the bookshelf? That's when you need to go to a storage room where you keep the rest of your books.

In computer terms, a page fault is like not finding a book you want to read on the bookshelf. It means that the computer doesn't have a particular part of the program you're using in its memory. So, it needs to go to another storage place (like a hard disk) to find that missing part, called a page, and bring it into memory.

While the computer is fetching the missing page, the program might have to wait for a short while. It's like you waiting for someone to bring you the book you wanted to read. Once the missing page is brought into memory, the program can continue running smoothly without any interruptions.

So, a page fault is when the computer needs to go to another place to find a missing part of a

program, and it may cause a brief delay in the program's execution. It's similar to you waiting for a book to be brought to you before you can continue reading.

CHAPTER FOUR

FILE SYSTEMS

Imagine you have a collection of different things, like books, toys, and drawings. To keep them organized, you give each item a name, like "My Favorite Book," "Teddy Bear," or "My Artwork." These names help you and others easily identify and find the items you're looking for.

In computers, a file is similar to one of those named items. It's a way to store information, such as text documents, pictures, or music. Just like you have names for your items, a file also has a name that helps humans recognize and access it.

A file has certain characteristics or attributes that describe it. These attributes include:

- **Name**: The human-readable name that helps identify the file.
- **Identifier**: A unique number or tag assigned to the file by the computer system.
- **Type**: The type of file, such as a document, image, or video.
- **Location**: The specific place where the file is stored on the computer's storage devices, like a hard disk.
- **Size**: The current size of the file, measured in bytes, which determines how much space it occupies.
- **Protection**: Information about who can access the file, read it, modify it, or execute it.
- **Time, date, and user identification**: Details about when the file was created, last modified, and last accessed, which can help with security and tracking usage.

All the information about files is organized and stored in a directory structure, which is like a digital filing cabinet. Each file has its own entry in the directory, containing its name and unique identifier.

So, a file is a named collection of information stored on a computer, and it has attributes that describe its properties, such as its type, size, location, and access permissions. The directory structure keeps track of all the files, just like a filing cabinet helps you keep track of your belongings.

FILE OPERATIONS

Creating a file: When we want to create a file, the computer needs to do two things. First, it looks for a suitable place in its memory to store the file. Then, it adds an entry in a special list called the directory, which keeps track of all the files. This way, the computer knows the file exists and where it is stored.

Writing a file: Imagine you want to write something down in a notebook. To do that on a computer, you use a special command that tells the computer the name of the file you want to write to and the information you want to write. It's like giving instructions to the computer to save your words or data in the file.

Reading a file: Just as you can read from a book or a paper, you can also read from a file on a computer. When you want to read a file, you use a command that tells the computer the name of the file you want to read from and where in the computer's memory you want the information from the file to be stored. The computer then brings that information into the memory for you to access.

Repositioning within a file: Let's say you have a big book with many pages. Sometimes you want to go directly to a specific page without reading everything from the beginning. Similarly, when working with files on a computer, you can use a command to tell the computer to find a particular part of a file. The computer searches the directory for the right file and moves a pointer

called the current-file-position pointer to the specified location within the file. It's like telling the computer to jump to a specific page in the book.

Deleting a file: To delete a file, we search the directory for the named file. Having found the associated directory entry, we release all file space, so that it can be reused by other files, and erase the directory entry.

Truncating a file: Truncating a file is like erasing all the content inside it, but keeping the file's other information the same. It's a bit like having a notebook with things written in it and deciding to clear all the pages while keeping the cover and other details the same.

When you truncate a file, you are essentially removing all the data or text inside the file, making it empty. However, the file's name, location, permissions, and other attributes remain unchanged. It's a way to start fresh with the file, as if it were brand new, but without the need to delete and recreate the file from scratch.

Think of it as taking a notebook and erasing everything you wrote on the pages, but the notebook itself still has the same cover, title, and other information. It's a way to keep the file's structure intact while removing its contents.

These are the basic operations performed on files: creating a file, writing to a file, reading from a file, and repositioning within a file. Each operation has its own purpose and involves different commands or instructions that the computer understands.

When we work with files, there are many operations we can perform, such as reading, writing, creating, and deleting files. To make these operations faster, the operating system keeps track of information about open files in a small table called the **open-file table**. This table helps the system find the necessary details about a file without constantly searching through the directory.

When we want to perform operations on a file, we first need to open it. The **open()** system call is used to open a file. It searches for the file in the directory and copies its information into the open-file table. This way, we can access the file directly without searching for it every time.

Once a file is open, several pieces of information are associated with it :-

- **File pointer:** This keeps track of the current position in the file, so we know where to read from or write to next.
- **File-open count:** This counts how many times the file has been opened by different processes. When this count reaches zero, meaning no process is using the file, it can be removed from the table.
- **Disk location:** This information tells us where the file is stored on the disk. It helps the system quickly access the file's data without repeatedly reading it from the disk.
- **Access rights:** Each process opens a file with specific access rights, determining what operations it can perform on the file. This information is stored in a table for each process, allowing the operating system to allow or deny further I/O requests.

When we are done working with a file, we close it. Closing a file removes its entry from the open-file table, indicating that we no longer need to access it actively.

Think of it like having a library card to borrow books. When you want to read a book, you first show your library card, which has all the necessary information about you and the book you want to read. Once you're done reading the book, you return it, and the library removes the book's details from your library card.

FILE TYPES

When we work with files, we often come across different types of files, such as documents, images, videos, and music. To help us identify the type of a file easily, many systems use a naming convention that includes a file extension.

The file extension is a part of the file name that comes after a period (.) character.

For example, let's say we have a file named "mydocument.docx". Here, "mydocument" is the name of the file, and "docx" is the file extension. The file extension tells us that this file is a document and it is associated with a specific program or file format, in this case, Microsoft Word.

Different file types have different extensions. Here are some common file types and their extensions:

- **Documents**: Files containing text, such as essays, reports, or letters. They often have extensions like .docx (Microsoft Word), .pdf (Portable Document Format), or .txt (Plain Text).
- **Images**: Files that represent pictures or graphics. They can have extensions like .jpg or .jpeg (JPEG image), .png (Portable Network Graphics), or .gif (Graphics Interchange Format).
- **Videos**: Files that contain moving images. They can have extensions like .mp4 (MPEG-4 video), .avi (Audio Video Interleave), or .mov (QuickTime Movie).
- **Music**: Files that store audio recordings or songs. They can have extensions like .mp3 (MPEG Audio Layer 3), .wav (Waveform Audio File Format), or .flac (Free Lossless Audio Codec).

By looking at the file extension, both the user and the operating system can quickly identify the type of the file and determine which program should be used to open or work with it. This makes it easier to organize and manage different types of files.

ACCESS METHODS

When we want to access information stored in a file, there are different methods we can use. Let's look at two common access methods: sequential access and direct access.

Sequential Access: Sequential access is the simplest method of accessing information in a file. With sequential access, data is processed in a specific order, one record after another. It is like reading a book from start to finish, where you read each page in order without skipping any pages.

In sequential access, a read operation reads the next portion of the file in the order it was written. Each time we read, a file pointer is automatically moved to the next position, keeping track of the current location. Similarly, a write operation appends data to the end of the file, extending the file's content.

Sequential access is commonly used by editors, compilers, and other programs that process data in a linear manner. However, it may not be efficient for large files when we need to access specific records randomly.

Direct Access: Direct access, also known as relative access, allows for random access to data within a file. In this method, the file is divided into fixed-length logical records or blocks, and each block is assigned a unique number.

With direct access, we can read or write records in any order, without having to go through the entire file sequentially. For example, we can read block 14, then jump to block 53, and then write to block 7. This random access is possible because of the underlying disk-based storage, which allows accessing any block directly.

To perform direct access, file operations are modified to include the block number as a parameter. Instead of reading or writing the next record, we specify the block number we want to read or write.

It's important to note that not all operating systems support both sequential and direct access methods for files. Some systems only allow sequential access, while others only support direct access. The choice of access method depends on the specific needs of the application and the capabilities of the operating system

DIRECTORY STRUCTURE

Storage Structure :- Imagine you have a big box where you can keep all your toys. This box is like a computer's storage device, called a disk.

Normally, you put all your toys in the box and that's it.

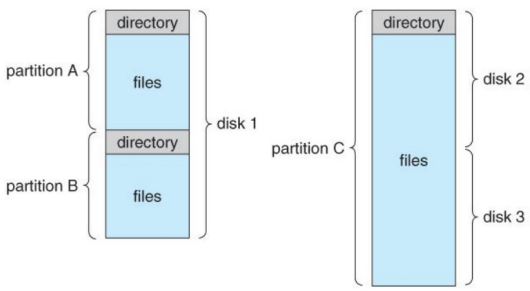
But sometimes, you might want to organize your toys in different ways. You could divide the box into sections and put different types of toys in each section. You could also use some parts of the box for things other than toys, like storing extra clothes or snacks.

In a computer, the box or disk can be used entirely for storing files. However, sometimes it's useful to divide the disk into smaller parts called partitions or slices. Each partition can have its own set of files. This way, you can keep different types of files separate from each other.

Sometimes, you might have more than one disk, and you can combine them to make a bigger storage space called a volume. Each volume is like a virtual box where you can store even more files.

Now, every box or volume needs a special list called a directory. This list keeps track of all the toys or files inside the box or volume. It has information like the names of the files, where they are located, how big they are, and what type they are. This helps you find and organize your files easily.

So, in simpler terms, a disk is like a big box where you keep files. You can divide the box into sections for different types of files. If you have more than one box, you can combine them to make an even bigger storage space. And to keep track of all the files, there is a special list called a directory that tells you information about each file.



DIRECTORY OVERVIEW

In simpler terms, when we talk about a directory structure, we need to think about the things we can do with it. These are the different operations:

1. **Search for a file**: It's like looking for a specific toy in your toy box. We want to find the right place in the directory where a particular file is listed.
2. **Create a file**: Imagine you want to add a new toy to your toy box. Similarly, we can create a new file and include it in the directory.
3. **Delete a file:** Sometimes, you may not want a toy anymore and want to remove it from your toy box. Similarly, we want to be able to remove a file from the directory when we don't need it anymore.
4. **List a directory**: When we want to see all the toys in our toy box, we can make a list of them. Similarly, we want to be able to see a list of all the files in a directory and get information about each file.
5. **Rename a file:** Suppose you want to change the name of a toy because its name doesn't match anymore. Similarly, we can change the name of a file in the directory to reflect its new content or purpose. This may also change its position in the directory.
6. **Traverse the file system**: Imagine you want to explore and look at every toy and box within your toy box collection. Similarly, we might want to access and go through every directory and file in a directory structure, exploring the entire system.

So, these operations involve finding files, adding new files, removing files, listing files, changing file names, and exploring the entire directory structure.

SINGLE LEVEL DIRECTORY

In a computer, a directory is like a folder that helps organize files. It keeps track of all the files on a computer and provides a way to access them easily. Just like how you can have different folders to store your toys, a computer has directories to store files.

Now, let's talk about the single-level directory structure in simpler terms:

Imagine you have a big box where you keep all your toys, and you don't have any other boxes or folders to organize them. All your toys are mixed together in that one box. This is similar to a single-level directory structure.

In a single-level directory, all the files are stored in one directory. It's like putting all your toys in one big box. This makes it easy to understand and manage because there is only one place to look for files.

However, there are some limitations to this approach:

1. **Unique names:** Since all the files are in the same directory, they must have different names. It's like each toy in your box needs to have a different name so that you can identify them easily. If two toys have the same name, it would be confusing.
2. **Difficulty in remembering**: Even if there is only one user and all the files are in the same directory, it can become difficult to remember the names of all the files as the number of files increases. It's like trying to remember the names of all the toys in a big box. As the number of toys (or files) grows, it becomes harder to keep track of everything.

So, in summary, a single-level directory structure is like having all your toys in one big box. It's simple to understand and manage, but it has limitations like the need for unique file names and difficulty in remembering the names of all the files.

TWO LEVEL DIRECTORY

Imagine you have a big box where you keep all your toys, but now you want to share the box with your friend. To avoid confusion, you decide to create a separate section in the box for each of you. This way, your toys are kept separate from your friend's toys. This is similar to a two-level directory structure.

In a two-level directory structure, each user has their own directory, which is like their personal toy section in the box. Each user's directory lists only the files that belong to that user. When a user starts using the computer or logs in, the system checks the master directory to find their specific directory.

Here are some key points about the two-level directory structure:

1. **User's own directory**: Each user gets their own separate space to store their files. It's like having your own personal section in the toy box where you can keep your toys.
2. **Unique file names**: Within each user's directory, file names only need to be unique within that user's directory. It's like each user can have their own toy named "Teddy" because they have their own section in the box.
3. **Master file directory:** There is a master directory that keeps track of each user's directory. It helps the system find the right user's directory when they log in. It's like having a list of names that tells you where each person's toy section is in the box.
4. **System executable files:** Apart from user directories, there is usually a separate directory for system executable files. These are special files that make the computer work.
5. **Access to directories**: The system can decide whether users are allowed to access directories other than their own. If

allowed, there must be a way to specify which directory they want to access. If access is denied, special arrangements must be made for users to run programs located in system directories.

6. **Search path:** A search path is like a list of directories where the system looks for specific programs. Each user can have their own unique search path to find programs.

So, in simpler terms, a two-level directory structure is like having separate toy sections in a shared toy box. Each user has their own section, and their files are listed only in their section. There is a master list that helps the system find each user's section. Users may or may not be able to access other sections, and they can have their own search paths to find programs.

TREE STRUCTURED DIRECTORIES

Imagine you have a big tree in your backyard. At the bottom of the tree, there is a main root from which all the branches grow. Each branch can have smaller branches growing out of it, and those smaller branches can have even more branches. It's like a family tree, where each person has children, and those children have their own children, and so on.

In a computer's directory structure, we can think of it as a tree like this. The tree has a main root directory, just like the base of the tree in your backyard. This root directory is the starting point for the entire file organization.

In this tree structure, users can create their own subdirectories (smaller branches) and organize their files within them. Each subdirectory can contain files or even more subdirectories.

Here are some key points about tree-structured directories :-

1. **Root directory:** The root directory is like the base of the tree. It's the top-level directory from which all other directories and files stem. Think of it as the starting point of the entire file organization.
2. **Paths and organization**: Every file in the system has a unique path name. It's like a specific address that tells you where a file is located within the tree structure. Users can create subdirectories (smaller branches) and organize their files within them to keep things organized.
3. **Current directory:** Each user or process has a concept of a current directory. It's like a starting point for that user or process to search for files. All searches for files happen relative to the current directory.
4. **Absolute and relative pathnames:** Files can be accessed using absolute pathnames, which start from the root of the tree and provide the full path to a file. Alternatively, files can be accessed using relative pathnames, which are relative to the current directory.
5. **Special structure**: Directories are stored in the computer's memory just like any other file, but there is a special bit that identifies them as directories. Additionally, directories have a specific structure that the operating system understands, helping it manage and navigate the tree structure efficiently.

So, in simpler terms, a tree-structured directory is like a big tree where files and subdirectories are organized. The root directory is the starting point, and users can create their own subdirectories to keep their files organized. Each file has a unique address within the tree. Users have a current directory from which they search for files. Directories are stored in a special way, allowing the operating system to understand and manage the tree structure effectively.

ACYCLIC GRAPH DIRECTORIES

Imagine you have a big playground, and there are multiple pathways connecting different areas of the playground. Some pathways lead to the same location. It's like having different routes to reach the same place.

In a computer's directory structure, we can have a similar arrangement using an acyclic graph. Unlike a tree structure where each file has a unique location, in an acyclic graph, directories can have shared subdirectories and files. This means the same file can exist in multiple directories, and any changes made to the file are immediately visible in all the locations.

To implement this sharing, we use a concept called "links." A link is like a pointer or reference to another file or subdirectory. It's a way of saying that this file can be found in multiple places. There are two types of links commonly used:

1. **Hard links:** A hard link involves multiple directory entries that all refer to the same file. It's like having signposts pointing to the same location from different paths. Hard links are only valid within the same file system.
2. **Symbolic links:** A symbolic link is a special file that contains information about where to find the linked file. It's like having a signpost that tells you where to go to find the real location. Symbolic links can be used to link directories and files in different file systems or within the same file system.

In the context of different operating systems:

- **UNIX**: UNIX supports both hard links and symbolic links. Hard links keep track of how many directory entries refer to the same file using a reference count. When a reference is removed, the link count is reduced, and when it reaches zero, the disk space can be reclaimed. Symbolic links provide flexibility by allowing links to be created for directories and files in different file systems.
- **Windows**: Windows primarily supports symbolic links, which are known as shortcuts. Shortcuts are similar to symbolic links and can be used to link directories and files.

However, there are considerations when using links:

- For hard links, a reference count is maintained, and when all the references to a file are removed, the disk space can be reclaimed.
- For symbolic links, there is a question of what to do if the original file is moved or deleted. Options include adjusting the symbolic links to reflect the new location or leaving them "dangling," meaning they are no longer valid when used.

So, in simpler terms, an acyclic-graph directory structure allows files and subdirectories to be shared. It's like having different pathways in a playground that lead to the same location. Links, such as hard links and symbolic links, are used to create these connections. Hard links refer to the same file in multiple locations, while symbolic links provide information on where to find the linked file. Different operating systems have different approaches to links.

FILE SYSTEM MOUNTING

Imagine you have a big toy storage area where you keep all your toys. But sometimes, you have so many toys that you need more than one storage area to fit them all. In that case, you can combine the different storage areas into one big storage space. This way, you can access all your toys from a single place.

In a computer, a file system is like a storage area for files. Sometimes, the files are too big to fit in just one storage area, so we need to combine multiple file systems to create one large storage area.

Mounting a file system is the process of combining these multiple file systems into one big tree structure. It's like connecting the different storage areas to create a unified space for all the files.

Here's how the mounting process works:

1. **Naming and location:** To mount a file system, we need to tell the computer the name of the storage device (like a disk) and the location where we want to attach the file system. The location is typically an empty folder, like an empty toy box waiting to be filled.
2. **Verification:** The computer checks if the storage device contains a valid file system. It reads the structure of the device, making sure it's in the expected format. It's like checking if the toy storage area is suitable for storing toys.
3. **Attaching and accessing:** Once the verification is successful, the computer notes in its system that the file system is mounted at the specified location. This allows the computer to access and navigate the file system as part of its overall file structure. It's like connecting the storage area to the rest of the toy organization, so you can easily find and play with any toy from the unified space.

For example, consider a situation where you have a main storage area on your computer's hard disk, and you also have a separate file system on a floppy disk. Before mounting, these file systems are separate and you can't access the files on the floppy disk easily. But when you mount the file system on the floppy disk, it becomes part of the main file structure, and you can access the files on the floppy disk from specific locations in the main structure.

Similarly, if you have multiple hard disks, you can mount all of them into a single unified structure.

So, in simpler terms, file-system mounting is like combining different storage areas into one big space. It allows the computer to access and organize files from multiple file systems as if they are part of a single structure. It's like connecting different storage areas so you can find and play with your toys from one place.

FILE SHARING

File sharing is when multiple users want to work together or access the same files to achieve a common computing goal. It's like sharing toys with your friends so that everyone can play together and have fun.

MULTIPLE USERS

When an operating system allows multiple users, there are important considerations for file sharing, file naming, and file protection:

1. **Access to other users' files:** The operating system can either allow users to access each other's files by default or require specific permission from the owner of the file. It's like deciding whether you can borrow a friend's toy without asking or if you need their permission first.
2. **Owner and group concept:** To implement sharing and protection, the system keeps track of additional attributes for files and directories. These include the owner (the user who has the most control over the file) and the group (a subset of users who can share access to the file).
3. **File and directory attributes:** The owner and group IDs of a file or directory are stored with other file attributes. When a user wants to perform an operation on a file, their user ID is compared with the owner attribute to determine if they are the owner of the file. Similarly, the group IDs are compared to see which permissions apply. Based on the comparison, the system allows or denies the requested operation.

REMOTE FILE SYSTEMS

1. **Networking and sharing:** With networks, computers can communicate with each other even if they are far apart. This allows sharing of resources, including data in the form of files. It's like being able to share toys with your friends who are in different houses.
2. **FTP (File Transfer Protocol):** Originally, file sharing across systems was done using FTP. It allowed individual files to be transported between systems as needed. FTP can require an account or password for access, or it can be anonymous, not requiring any user name or password.
3. **Distributed file systems:** There are various forms of distributed file systems that enable remote file systems to be connected to a local directory structure. This means you can access files on remote systems as if they are part of your own computer. The actual files are still transported across the network as needed, possibly using FTP as the underlying transport mechanism.
4. **World Wide Web (WWW):** The WWW has made it easy to access files on remote systems without connecting their entire file systems. This is often done using FTP as the underlying file transport mechanism, and it allows you to view and download files from remote systems.

So, in simpler terms, file sharing is when multiple users work together or access the same files. It's like sharing toys with your friends. The operating system determines whether users can access each other's files. There are owners who have more control over the files, and groups of users who can share access. With networking, files can be shared across computers using protocols like FTP. Remote file systems can be connected to a local structure, allowing access to files on remote systems. The World Wide Web also allows you to access files on remote systems using FTP.

When computers want to access files on another computer, they can use a remote file system. It's like asking a friend to fetch a toy from their toy storage area and share it with you. In this case, the computer with the files is called the server, and the computer accessing the files is called the client.

Client-Server Model :-

1. Client and server: The client computer wants to access files, and the server computer has those files. It's like you asking your friend (client) to get a toy from their toy storage (server) for you.
2. User IDs and group IDs: To work properly, the user IDs and group IDs must be consistent across both systems. This is important when multiple computers managed by the same organization share files with a common group of users. It's like making sure you and your friend agree on who the toys belong to.
3. Security concerns: There are security concerns in this model. Servers often restrict access to trusted systems only to prevent impersonation (a computer pretending to be another). They may also restrict remote access to read-only mode. Additionally, servers control which file systems can be mounted remotely, and they usually keep limited, relatively public information within those file systems, protected by regular backups. The NFS (Network File System) is an example of such a system.

Failure Modes :-

1. Local file system failures: Local file systems can fail for various reasons, like disk failure, corruption of directory structure or other disk management information, hardware or cable failures, or human errors. These failures may cause the computer to crash, and human intervention is needed to fix the problems.

2. Remote file system failures: Remote file systems have additional failure modes due to the complexity of network systems. Interruptions in the network between two computers can occur due to hardware failure, poor configuration, or network implementation issues.
3. Handling loss of remote file system: Imagine the client computer is using a remote file system, and suddenly, it becomes unreachable. In this situation, the client computer should not behave as it would if a local file system was lost. Instead, it can either terminate all operations to the lost server or delay operations until the server is reachable again. These behaviors are defined in the remote-file-system protocol. Terminating operations can result in users losing data, so most protocols allow delaying file system operations with the hope that the remote server will become available again.

So, in simpler terms, a client-server model is like asking a friend (client) to fetch a toy from their toy storage (server) and share it with you. In remote file systems, computers access files on other computers. Security measures are in place to protect access, and certain failures can occur in both local and remote file systems. When the remote file system is lost, the client computer can either stop operations or wait for the server to be reachable again.

PROTECTION

When we want to protect files on a computer, we need to control who can do what with those files. This is like having different rules for different toys so that everyone plays with them properly. Access control mechanisms help us control the types of actions that can be performed on files.

Types of Access :-

1. Read: It means looking at the contents of a file, like reading a book.
2. Write: It means making changes or adding new information to a file, like writing in a notebook.
3. Execute: It means running or using a file as a program, like playing a game on a computer.
4. Append: It means adding new information at the end of a file, like adding a new page to a book.
5. Delete: It means removing a file completely, like throwing away a toy.
6. List: It means seeing the name and details of a file, like looking at a list of toys and their descriptions.

Access Control: To control access to files, we usually associate each file with an access-control list (ACL). It's like having a list of people who can play with a specific toy and what they are allowed to do with it.

When someone wants to access a file, the computer checks the access-control list for that file. If the person is listed with the requested access (like read or write), they are allowed to perform that action. If they are not listed or don't have the required access, they are denied access.

One way to simplify access control is by categorizing users into three groups for each file :-

1. **Owner**: The person who created the file. They have the most control and can do all actions on the file.
2. **Group**: A set of people who share the file and need similar access. They have specific permissions, like read and write, but can't delete the file.
3. **Universe**: All other users of the system who don't fall into the owner or group category. They have limited permissions, usually just read access.

For example, let's say Sara is writing a book and hires three graduate students (Jim, Dawn, and Jill) to help. Sara should have full access to the book file, while the students should have read and write access but not the ability to delete the file. Other users in the system should only have read access. To achieve this, a group called "text" is created with the students as members, and the access rights are set accordingly.

To ensure proper control, permissions and access lists need to be managed carefully. In some systems, only authorized individuals, like managers, can create or modify groups.

So, in simpler terms, access control is about setting rules for who can do what with files. We have different types of access like read, write, and execute. Access-control lists help us keep track of who can access files and what they can do with them. By categorizing users into owner, group, and universe, we simplify access control. It's like having different rules for different people to play with different toys.

ALLOCATION METHODS

When we store files on a disk, we need to decide how to allocate space for those files. It's like finding the best places to keep your toys in your toy storage area so that they can be easily accessed. There are different methods for allocating disk space, and we will discuss three major methods: contiguous, linked, and indexed.

1) Contiguous Allocation :-

1. **Contiguous allocation:** This method requires that each file is stored in a continuous block of space on the disk. It's like putting all the parts of a puzzle together in one place.
2. **Performance benefits:** Contiguous allocation provides fast performance because reading consecutive blocks of the same file doesn't require moving the disk heads much. It's like flipping through pages of a book without needing to jump around.
3. **Storage allocation issues:** Similar to allocating blocks of memory, allocating contiguous disk space involves considerations like first fit, best fit, and fragmentation problems. However, with disks, moving the disk heads to different locations takes time, so it may be more beneficial to keep files contiguous whenever possible.
4. **Compacting the disk:** Even file systems that don't store files contiguously by default can benefit from utilities that compact the disk. These utilities rearrange the files and make them contiguous, improving performance.
5. **Problems with file growth:** There can be issues when files grow or when the exact size of a file is unknown at creation time:
    - Overestimating the file's final size leads to external fragmentation and wastes disk space.
    - Underestimating the size may require moving the file or aborting a process if the file outgrows its allocated space.
    - Slow growth over time with a predetermined initial allocation may result in a lot of unusable space before the file fills it.
6. **Extents:** Another variation of contiguous allocation is allocating file space in large chunks called extents. If a file outgrows its original extent, an additional extent is allocated. An extent can be as large as a complete track or even a cylinder, aligned on appropriate boundaries.

So, in simpler terms, allocation methods are ways to decide where to store files on a disk. Contiguous allocation means keeping the parts of a file together. It helps with fast access, but problems can arise when files grow or their sizes are unknown. Compacting the disk can improve performance. Extents are large chunks of space

used for allocation. It's like finding the best spots to store your toys together or in big chunks.

2) LINKED ALLOCATION

When we store files on a disk, we need to decide how to allocate space for those files. We have discussed contiguous allocation, and now let's simplify the concept of linked allocation.

Linked Allocation:

1. Linked allocation: In linked allocation, each file is like a chain made up of blocks on the disk. The blocks can be scattered anywhere on the disk, and the directory contains pointers to the first and last blocks of the file.
2. Example: Imagine a file with five blocks. The first block might be at block 9, then it continues to block 16, then block 1, then block 10, and finally block 25. It's like connecting different puzzle pieces together to form a complete picture.
3. Benefits of linked allocation: Linked allocation solves the problems of external fragmentation and pre-known file sizes. Files can grow dynamically at any time without issues. It's like being able to add more puzzle pieces to a picture whenever needed.
4. Sequential access efficiency: Linked allocation is efficient for sequential access, where you read the blocks in order. It's like reading a storybook from the beginning to the end. However, random access, where you jump to different parts of the file, is less efficient as you need to start from the beginning of the list for each new location.
5. Clusters and internal fragmentation: To reduce the space wasted by pointers, clusters of blocks can be allocated together. This helps to manage internal fragmentation, where some space within a block is unused. It's like grouping several puzzle pieces together to form larger clusters.
6. Reliability and File Allocation Table (FAT): A problem with linked allocation is the risk of losing or damaging a pointer. Doubly linked lists provide some protection, but they require additional overhead and wasted space. In the DOS operating system, a variation called the File Allocation Table (FAT) is used. It stores all the links in a separate table at the beginning of the disk. This allows the FAT table to be cached in memory, improving random access speeds.

So, in simpler terms, linked allocation is like connecting blocks of a file together using pointers. It doesn't suffer from external fragmentation and allows files to grow dynamically. Sequential access works well, but random access can be slower. Clusters of blocks can be used to reduce wasted space. The File Allocation Table (FAT) is a variation of linked allocation used in DOS, which improves random access speed. It's like connecting puzzle pieces to create a picture or using a table to keep track of links in a storybook.

INDEXED ALLOCATION

We have discussed contiguous and linked allocation methods for storing files on a disk. Now let's simplify the concept of indexed allocation.

Indexed Allocation:

1. Indexed allocation: Indexed allocation combines all the indexes needed to access each file into a common block specific to that file. Instead of scattering the indexes all over the disk or using a FAT table, each file has its own index block, which is like an array of disk-block addresses.
2. Wasted disk space: Indexed allocation requires allocating an entire index block for each file, regardless of the number of data blocks the file contains. This leads to

some disk space being wasted compared to linked lists or FAT tables. It's like having a separate sheet of paper for each chapter in a book, even if some chapters are shorter.

3. Different approaches: There are different approaches to implementing the index block:
    - Linked scheme: In the linked scheme, an index block is one disk block that can be read and written in a single disk operation. The first index block contains some header information and the addresses of the first N data blocks. If needed, it may also include a pointer to additional linked index blocks.
    - Multi-level index: In the multi-level index scheme, the first index block contains pointers to secondary index blocks. These secondary index blocks, in turn, contain pointers to the actual data blocks. It's like having a table of contents pointing to different sections, and each section has its own page numbers.
    - Combined scheme: The combined scheme, used in UNIX inodes, stores the first few data block pointers directly in the inode itself. Then, singly, doubly, and triply indirect pointers are used to access more data blocks as needed. It's like having quick access to a few specific pages in a book and using bookmarks to reach other pages.

In simpler terms, indexed allocation uses a separate index block for each file, making it easier to find specific data blocks. However, it requires some disk space to be wasted. Different approaches, like the linked scheme or multi-level index, are used to organize the index blocks. It's like having separate sheets for each chapter in a book or a table of contents with pointers to different sections. The combined scheme used in UNIX inodes allows for quick access to some data blocks and uses indirect pointers for accessing more blocks. It's like having bookmarks to reach specific pages and using references for other pages.