22516 Operating System

Unit-IV CPU Scheduling and Algorithms

4.1 Scheduling types - scheduling Objectives, CPU and I/O burst cycles, Pre-emptive, Non- Pre-emptive Scheduling, Scheduling criteria.

Scheduling Types

In computing, scheduling is the method by which work is assigned to resources that complete the work. Here are several types of scheduling in operating systems:

1. **First-Come, First-Served (FCFS) Scheduling**: Jobs are executed on first come, first serve basis. It's simple, easy to understand and implement.
2. **Shortest-Job-Next (SJN) Scheduling**: Also known as Shortest Job First (SJF) or Shortest Process Next (SPN). In this type of scheduling, the process with the smallest execution time is selected for execution next. This minimizes waiting time for a computing process but can cause longer processes to wait indefinitely.
3. **Priority Scheduling**: In priority scheduling, each process is assigned a priority and the process with the highest priority is executed first. If two processes have the same priority then FCFS is used to break the tie.
4. **Round Robin (RR) Scheduling**: Each process in the ready queue is assigned a time quantum (a small unit of time) and execution is allowed for that time period. If execution is not completed, then the process goes back to the ready queue and waits for its next turn.
5. **Multilevel Queue Scheduling**: This scheduler assigns processes to different queues based upon properties of the process, such as memory size, process priority, or process type. Each queue may have its own scheduling algorithms.
6. **Multilevel Feedback Queue Scheduling**: This is a complex scheduler that separates processes according to their needs for the processor. If a process uses too much CPU time, it will be moved to a lower-priority queue.

These different scheduling methods provide trade-offs between various factors including fairness, response time, throughput, and resource utilization. The choice of scheduling algorithm can greatly affect the performance of the system.

Scheduling Objectives

Scheduling is a key task in operating systems to manage the execution of processes. The scheduler is responsible for deciding which processes should run, when they should run, and for how long they should be allowed to run. The main objectives of scheduling include:

1. **Fairness**: Each process should have a fair share of the CPU.
2. **Efficiency**: Keep the CPU and other system resources as busy as possible.
3. **Throughput**: Maximize the number of processes that complete their execution per time unit.
4. **Turnaround Time**: Minimize the time it takes for a process to complete. Turnaround time is the sum of the periods spent waiting to get into memory, waiting in the ready queue, executing on the CPU, and doing I/O.
5. **Waiting Time**: Minimize the time that processes spend in the ready queue.
6. **Response Time**: Minimize the time it takes from when a request was submitted until the first response is produced. This is particularly important in interactive systems.

- Examples of preemptive scheduling algorithms include Round Robin, Priority Scheduling, and Multilevel Feedback Queue Scheduling.

2. **Non-Preemptive Scheduling**:
   - In non-preemptive scheduling, a running process continues to execute on the CPU until it voluntarily releases the CPU or until it completes its execution.
   - The scheduler does not forcibly interrupt a process; instead, it waits for a process to finish its CPU burst and relinquish the CPU voluntarily.
   - Non-preemptive scheduling provides better predictability, as a process can execute without interruption until it finishes its current task.
   - However, it may lead to lower CPU utilization and less responsiveness if a long-running process does not release the CPU.
   - Examples of non-preemptive scheduling algorithms include First-Come, First-Served (FCFS) Scheduling and Shortest-Job-Next (SJN) Scheduling.

The choice between preemptive and non-preemptive scheduling depends on the specific requirements of the system and the applications running on it. Preemptive scheduling is often preferred in modern operating systems as it provides better control over process execution and responsiveness in multitasking environments. Non-preemptive scheduling is still used in some embedded systems and real-time applications where predictability and deterministic behavior are critical.

Scheduling criteria

Scheduling is a critical function of an operating system that manages the execution of processes. Several criteria are used to compare and evaluate scheduling algorithms:

1. **CPU Utilization**: The goal is to keep the CPU as busy as possible. High CPU utilization is desired to get maximum usage out of the CPU.
2. **Throughput**: This refers to the number of processes that complete their execution per time unit. A higher throughput is generally better as it means more processes are being completed.
3. **Turnaround Time**: This is the amount of time taken to execute a particular process. It is the interval from the time of submission of a process to the time of completion. The goal is to minimize the average turnaround time for a batch of processes.
4. **Waiting Time**: This is the sum of the periods spent waiting in the ready queue. The scheduling algorithm should aim to minimize the average waiting time.
5. **Response Time**: This is the amount of time it takes from when a request was submitted until the first response is produced. For interactive systems, minimizing response time is more important than minimizing waiting or turnaround time.
6. **Fairness**: Ensuring all processes get a fair share of the CPU and no process is starved for too long.
7. **Prioritization**: Some processes may have a higher priority than others, and the scheduler needs to take this into account.
8. **Balancing Resource Utilization**: The scheduler should not only manage the CPU but also take into account the utilization of other resources such as I/O devices and memory.

Each scheduling algorithm may prioritize these criteria differently, and the best choice of scheduling algorithm depends on the specific requirements of the system and its workload.

4. If a new process arrives in the ready queue while a process is executing, and the new process has a shorter remaining time than the currently executing process, the CPU is taken away from the current process (it is preempted), and the new process is scheduled to run.
5. When the running process completes, the scheduler again selects the process with the shortest remaining time.

The benefits of SRTN include:

- It can provide better average turnaround time than SJF, as it allows a shorter process to run immediately rather than waiting for a longer process to complete.

The limitations of SRTN include:

- Like SJF, SRTN suffers from the problem of starvation. Longer processes may wait indefinitely if shorter processes keep arriving.
- Predicting the time remaining of a process can be challenging in practical implementations.
- It can lead to higher overhead due to increased context switching if processes with shorter remaining times frequently arrive.

Because of these limitations, pure SRTN isn't typically used in general-purpose operating systems. Variations or approximations of SRTN may be used instead, which make use of priority scheduling or implement aging to prevent starvation.

Round Robin (RR)

Round Robin (RR) is a preemptive scheduling algorithm used by operating systems for managing process execution. It is one of the simplest, fairest, and easiest to understand scheduling algorithms. Here's how it works:

1. Every process in the system is placed into a queue.
2. The scheduler selects the process at the front of the queue and allocates the CPU to it.
3. The selected process is allowed to run for a fixed amount of time known as a time quantum (or time slice).
4. If the process finishes before the time quantum expires, it is removed from the queue. If it doesn't finish, it is moved to the back of the queue to await its next turn.
5. The scheduler then allocates the CPU to the process that is now at the front of the queue, and the process repeats.

The key characteristics of Round Robin scheduling include:

- It is simple to implement and understand.
- It is fair because every process gets an equal share of the CPU.
- The performance of Round Robin scheduling can be significantly impacted by the length of the time quantum. If the time quantum is too short, the system spends too much time switching between processes. If the time quantum is too long, the system becomes more like a First-Come, First-Served (FCFS) system, and responsiveness can suffer.

Round Robin scheduling is often used in interactive systems where equal share and fairness are important, and it is suitable for time-sharing systems. However, it does not consider process priority, and it may not be efficient for systems with varying process burst times.

Priority scheduling

Priority scheduling is a method of scheduling processes that is based on priority. In this scheduling method, the operating system assigns a priority rank to each process, and the CPU is allocated to the process with the highest priority.
Here's how it works:

Necessary Conditions leading to Deadlocks

For a deadlock to occur, four necessary conditions must hold simultaneously in a system. These are known as Coffman conditions after their authors, E. G. Coffman, Jr., M. J. Elphick, and A. K. Shoshani:

1. **Mutual Exclusion**: At least one resource must be in a non-sharable mode, that is, only one process at a time can use the resource. If another process requests that resource, the requesting process must be delayed until the resource has been released.
2. **Hold and Wait (or Resource Holding)**: A process is currently holding at least one resource and requesting additional resources which are being held by other processes.
3. **No Preemption**: Resources cannot be forcibly removed from the processes holding them until the resources are used to completion. In other words, a resource can only be released by the process that is currently holding it.
4. **Circular Wait**: There exists a set {P0, P1, …, Pn} of waiting processes such that P0 is waiting for a resource held by P1, P1 is waiting for a resource held by P2, and so on, Pn is waiting for a resource held by P0.

All these four conditions must hold for a deadlock to occur. If we can prevent at least one of the conditions from holding, we can prevent deadlocks. The system can adopt a specific strategy for ensuring that at least one of these conditions does not hold, thus making deadlocks impossible.

Deadlock Handling - Preventions and avoidance.

Handling deadlocks involves employing a strategy that ensures at least one of the four necessary conditions for deadlocks cannot occur. This can be done through three approaches: deadlock prevention, deadlock avoidance, and deadlock detection.

1. **Deadlock Prevention**: In this approach, the system actively prevents the occurrence of one or more of the four necessary conditions. This can be done in one of two ways:
    - **Negate Mutual Exclusion**: Make resources sharable where possible. For some resources, such as a printer or a CD drive, this is not possible as they are intrinsically non-sharable.
    - **Negate Hold and Wait**: Require processes to request all the resources they will need upfront before execution begins. This can be inefficient as a resource may be held idle for a long time. Alternatively, allow a process to request resources only when it has none; thus, it must release all its held resources before requesting more.
    - **Negate No Preemption**: If a process that is holding some resources requests another resource that cannot be immediately allocated, then all resources currently being held are released. Preempted resources are added to the list of resources for which the process is waiting. The process will be restarted only when it can regain its old resources and obtain the new ones it is requesting.
    - **Negate Circular Wait**: Impose a total ordering of all resource types and require that each process requests resources in an increasing order of enumeration.
2. **Deadlock Avoidance**: The system requires additional information in advance about which resources a process will need during its lifetime. With this information, the system can decide for each resource request whether the process should wait or can safely be allocated the resource. The system can then make a safe decision that avoids entering an unsafe (potentially deadlocked) state.
    - The most famous deadlock avoidance algorithm is the Banker's Algorithm, which is used when a process requests resources. The algorithm checks if granting the request will mean that the system remains in a safe state, then the request is granted, else the process must wait.

7. **Deadlock Prevention**: Ensure that the system will never enter a deadlock state, where a group of processes are each waiting for resources held by others in the group, causing all of them to wait indefinitely.
8. **Protection**: Ensure that the activities of one process do not adversely affect other processes.
9. **Resource Utilization**: Ensure efficient use of system resources, including the CPU, memory, disk I/O, etc.
10. **Predictability**: The system's behavior should be predictable and not drastically vary the outcomes under similar conditions.

These objectives are often contradictory (improving one can worsen another), and thus, there is usually a need for a trade-off among them. The choice of specific objectives, or the weight given to each, will depend on the specific needs of the system and its users.

CPU and I/O burst cycles

In the life of a process, there are typically two types of bursts or cycles - CPU bursts and I/O bursts.

1. **CPU Burst**: CPU bursts refer to the time periods during which a process uses the CPU (for example, for doing calculations). The length of these bursts is determined by the nature of the process - some processes have longer CPU bursts than others. A CPU-intensive task, like mathematical computation, would have longer CPU bursts compared to a task that is more I/O-focused.
2. **I/O Burst**: I/O bursts refer to the time periods during which a process uses I/O services, such as reading from a disk or writing to a network socket. During an I/O burst, the process waits for the I/O operation to complete and doesn't need the CPU.

The pattern of CPU and I/O bursts is characteristic for each process. Some processes might be CPU-bound (long CPU bursts, few I/O bursts), others might be I/O-bound (many short CPU bursts, long I/O bursts). This pattern of bursts is also taken into account by the CPU scheduler when deciding the execution order of processes.

A typical process execution scenario starts with a CPU burst. That is followed by an I/O burst, during which the CPU is idle (unless there are other ready processes that can use it). This is followed by another CPU burst, then another I/O burst, and so on, until the process is complete. The pattern might be described as CPU burst – I/O wait – CPU burst – I/O wait – CPU burst, and so on.

Pre-emptive and Non- Pre-emptive Scheduling

Preemptive and non-preemptive (also known as cooperative) scheduling are two different approaches to process scheduling in operating systems:

1. **Preemptive Scheduling**:
    - In preemptive scheduling, the operating system has the authority to interrupt a running process and force it to relinquish the CPU, even if the process has not completed its execution.
    - The scheduler can interrupt a process and switch to another process based on predetermined criteria, such as time quantum expiration or higher-priority processes becoming ready.
    - Preemptive scheduling ensures that no single process can monopolize the CPU for an extended period, improving fairness and responsiveness.
    - Context switching is more frequent in preemptive scheduling, as processes are often switched even before they voluntarily release the CPU.

4.2 Types of Scheduling algorithms - First come first served (FCFS), Sho11est Job First (SJF), Shortest Remaining Time (SRTN), Round Robin (RR) Priority scheduling, multilevel queue scheduling.

Types of Scheduling algorithms

Several types of scheduling algorithms are used in operating systems to manage the execution of processes:

1. **First-Come, First-Served (FCFS)**: Jobs are processed in the order in which they arrive. Once a job begins execution, it runs to completion.
2. **Shortest Job Next (SJN)**: Also known as Shortest Job First (SJF) or Shortest Process Next (SPN), this algorithm selects the job with the smallest total CPU burst time. This approach can lead to process starvation for processes which will require a longer time to complete if short processes are continually added.
3. **Priority Scheduling**: In this type of scheduling, each process is assigned a priority, and priority is used to schedule the processes. If processes have the same priority, a FCFS scheduling is used to break the tie. Priority can be either static (set by the user or system at the creation time) or dynamic (changing based on the behavior of the process).
4. **Round Robin (RR)**: Round Robin scheduling is a preemptive algorithm that is quite similar to FCFS scheduling, but preemption is added to switch between processes. A small unit of time called a time quantum is defined, all processes get to execute for one quantum at a time.
5. **Multilevel Queue Scheduling**: In this scheduling, processes are divided into multiple groups or queues based on process attributes like process type, CPU burst time, process age, priority, etc. Each queue can have its own scheduling algorithm.
6. **Multilevel Feedback Queue Scheduling**: This is a complex version of Multilevel Queue Scheduling. In this scheduling, a process can move between queues. The idea is to separate processes with different CPU-burst characteristics.

The choice of scheduling algorithm can greatly impact the system's performance, and different algorithms are suited to different circumstances. The scheduler in modern operating systems often use a combination of these algorithms.

First come first served (FCFS)

First-Come, First-Served (FCFS) is one of the simplest scheduling algorithms used by operating systems for process scheduling. As the name suggests, in FCFS, the process that arrives first gets executed first.

Here's how it works:

1. The operating system maintains a queue of all the ready processes. The processes are positioned in the queue in the order that they arrive.
2. The process at the head of the queue is selected for execution on the CPU.
3. The process runs until it completes or until it blocks because it needs to wait for an I/O operation to complete.
4. When the running process completes or blocks, the scheduler picks the next process in the queue.

FCFS is simple to understand and implement but it has some significant limitations:

1. **Convoy Effect**: In a mix of short and long processes, if a long process gets the CPU first, then all the other shorter processes have to wait for a long time leading to poor average response time. This scenario is known as the convoy effect.
2. **No Preemption**: In FCFS, once a process starts executing, it runs to completion. There's no way to temporarily stop a running process to let another process run.

3. **Non-Optimal Waiting Time**: FCFS can lead to high average waiting time. For example, if a very long process enters the system just before many short processes, the short processes unfairly wait for a long time.
4. **No Priority**: FCFS doesn't consider the priority of processes. A higher priority process has to wait if a lower priority process arrives earlier.

Despite its simplicity, because of these limitations, FCFS is not often used in modern general-purpose operating systems. However, it might still be suitable for simple or specialized environments where tasks have similar lengths or where preemption and priorities are not a concern.

Shortest Job First (SJF)

Shortest Job First (SJF) is a scheduling algorithm used by operating systems to manage the execution of processes or tasks. As the name suggests, in SJF, the process with the shortest burst time (execution time) is scheduled next.

Here's how it works:

1. The operating system maintains a list or a queue of all the ready processes.
2. The scheduler goes through the list of ready processes and selects the process with the shortest estimated execution time.
3. The selected process is then dispatched to the CPU for execution.
4. When the running process completes or blocks (for example, to wait for an I/O operation), the scheduler again looks for the ready process with the shortest execution time.

SJF can be either non-preemptive (once a process starts executing, it runs to completion) or preemptive (if a new process arrives with a shorter burst time than what's left of the currently executing process, the current process is preempted, and the new process takes over).

The advantages of SJF include:

- It is optimal in terms of minimizing the average waiting time for a set of processes.

However, it has some significant limitations:

- **Starvation**: Processes with longer burst times may end up waiting indefinitely if shorter processes keep coming. This problem is known as starvation, or the convoy effect.
- **Burst Time Prediction**: It's difficult to know the execution time of a process in advance. Real systems don't have perfect knowledge of how long a job or process will last.

Because of these limitations, pure SJF isn't often used in general-purpose operating systems. Instead, variations of SJF, like adaptive or feedback algorithms, might be used, where the scheduler estimates the lengths of the CPU bursts.

Shortest Remaining Time (SRTN)

Shortest Remaining Time Next (SRTN), also known as Shortest Remaining Time First (SRTF), is a preemptive version of the Shortest Job First (SJF) scheduling algorithm.

In SRTN scheduling, the process with the smallest amount of time remaining until completion is selected to execute. Because it is preemptive, if a new process arrives that requires less time to complete than the remaining time of the currently executing process, the currently executing process is preempted, and the new process is scheduled.

Here's how it works:

1. The operating system maintains a list or queue of all ready processes.
2. The scheduler selects the process with the shortest remaining time from the list of ready processes.
3. This process is dispatched to the CPU for execution.

1. Each process is assigned a priority when it enters the system. The assignment could be based on a variety of factors such as the importance of the process, the type or category of the process, the amount of resources the process requires, the user's preferences, etc.
2. The scheduler selects the process with the highest priority from the list of ready processes.
3. The selected process is dispatched to the CPU for execution.
4. If a new process arrives in the ready queue that has a higher priority than the currently executing process, the CPU may be preempted from the current process and given to the new process, depending on whether the system is using preemptive or non-preemptive scheduling.

Priority scheduling has several advantages:

- It allows the operating system to place more important tasks ahead of less important ones, which can be useful in a variety of situations.
- It can be either non-preemptive (once a process starts executing, it runs to completion) or preemptive (if a new process arrives with a higher priority than the currently running process, the current process is preempted and the new process starts executing).

However, priority scheduling also has some drawbacks:

- **Starvation**: Low-priority tasks may end up waiting indefinitely if high-priority tasks keep coming. This problem is known as starvation or indefinite blocking. It can be resolved using techniques such as aging (gradually increasing the priority of processes that wait for a long time).
- **Priority Inversion**: This is a situation where a lower priority process holds a resource required by a higher priority process, resulting in the higher priority process having to wait.
- **Difficulty in Priority Assignment**: It can be challenging to assign the appropriate priorities to each process, especially in complex systems.

Despite these challenges, priority scheduling is widely used in operating systems due to its flexibility and adaptability to a wide range of scenarios.

multilevel queue scheduling

Multilevel Queue Scheduling is a complex scheduling algorithm used by operating systems which separates processes into different groups or queues based on their characteristics, and each queue has its own scheduling algorithm.

Here's how it works:

1. Processes are divided into different queues based on their characteristics. For example, a common division is to separate system processes, interactive processes, and batch processes into different queues. The criteria for division into queues can be the memory size, process priority, process type, or CPU burst time.
2. Each queue has its own scheduling algorithm. For instance, the foreground (interactive) queue might be scheduled using a Round Robin scheduler, while the background (batch) queue might use FCFS scheduling.
3. There's a scheduler for the queues as well, which determines which queue gets the CPU. This could be a simple priority system (the queue with the highest priority is always given the CPU if it has processes ready to run), or it could use time slices to cycle between the queues.

Advantages of Multilevel Queue Scheduling include:

- It is capable of differentiating between different types of processes and can handle a wide range of process types, ensuring that each gets an appropriate share of the CPU.
- It allows for priority scheduling, as more important tasks can be placed in higher-priority queues.

Disadvantages include:

- Inflexibility. Once a process is assigned to a queue, it cannot be moved to another queue.
- It can lead to starvation. If higher-priority queues always have processes to execute, a lower-priority queue may never get the CPU.

Despite these challenges, Multilevel Queue Scheduling can be very effective in situations where processes are easily divided into different classes and have significantly different behavior. It's commonly used in real-world operating systems.

4.3 Deadlock - System Models, Necessary Conditions leading to Deadlocks, Deadlock Handling - Preventions, avoidance.

Deadlock

A deadlock is a state in a computer system where a set of processes is unable to proceed because each is waiting for a resource that is held by another process in the set. Deadlock is a situation of interprocess communication where progress is impossible.

Here's a common example: Consider two processes, A and B, and two resources, 1 and 2. Process A has resource 1 and needs resource 2. At the same time, process B has resource 2 and needs resource 1. Neither process can continue until the other process releases a resource. They are in a deadlock.

There are four necessary conditions for a deadlock to occur, known as Coffman conditions:

1. **Mutual Exclusion**: At least one resource must be held in a non-sharable mode, meaning only one process can use the resource at any given time.
2. **Hold and Wait**: A process holding at least one resource and waiting to acquire additional resources that are currently held by other processes.
3. **No Preemption**: Resources cannot be forcibly taken from a process holding them; resources must be explicitly released by the process holding them.
4. **Circular Wait**: There exists a set of waiting processes such that P1 is waiting for a resource held by P2, P2 is waiting for a resource held by P3, and so on until Pn is waiting for a resource held by P1. This forms a circular chain.

Deadlocks are a problem because they halt the execution of processes, and resolving a deadlock often requires intervention, which can be costly in terms of system performance or even lead to system failure. Operating systems use various techniques to prevent, detect and recover from deadlocks.

Deadlock System Models

A system model is necessary to characterize and analyze deadlocks. Typically, a system is modeled in terms of resources and processes.

Here's a simple model:

Resources: These are the things that the processes in our system will be using and competing for. Examples include printers, memory, CPU cycles, disk drives, databases, and so on.

In the system model, resources are characterized by two properties:

1. Each resource type Rj has Nj instances. Each instance is identical in terms of its functionality.
2. Each resource instance can be either in the available state or allocated to exactly one process.

Processes: These are the activities in our system that use resources and compete with other processes for resources.

In the system model, processes follow this general sequence:

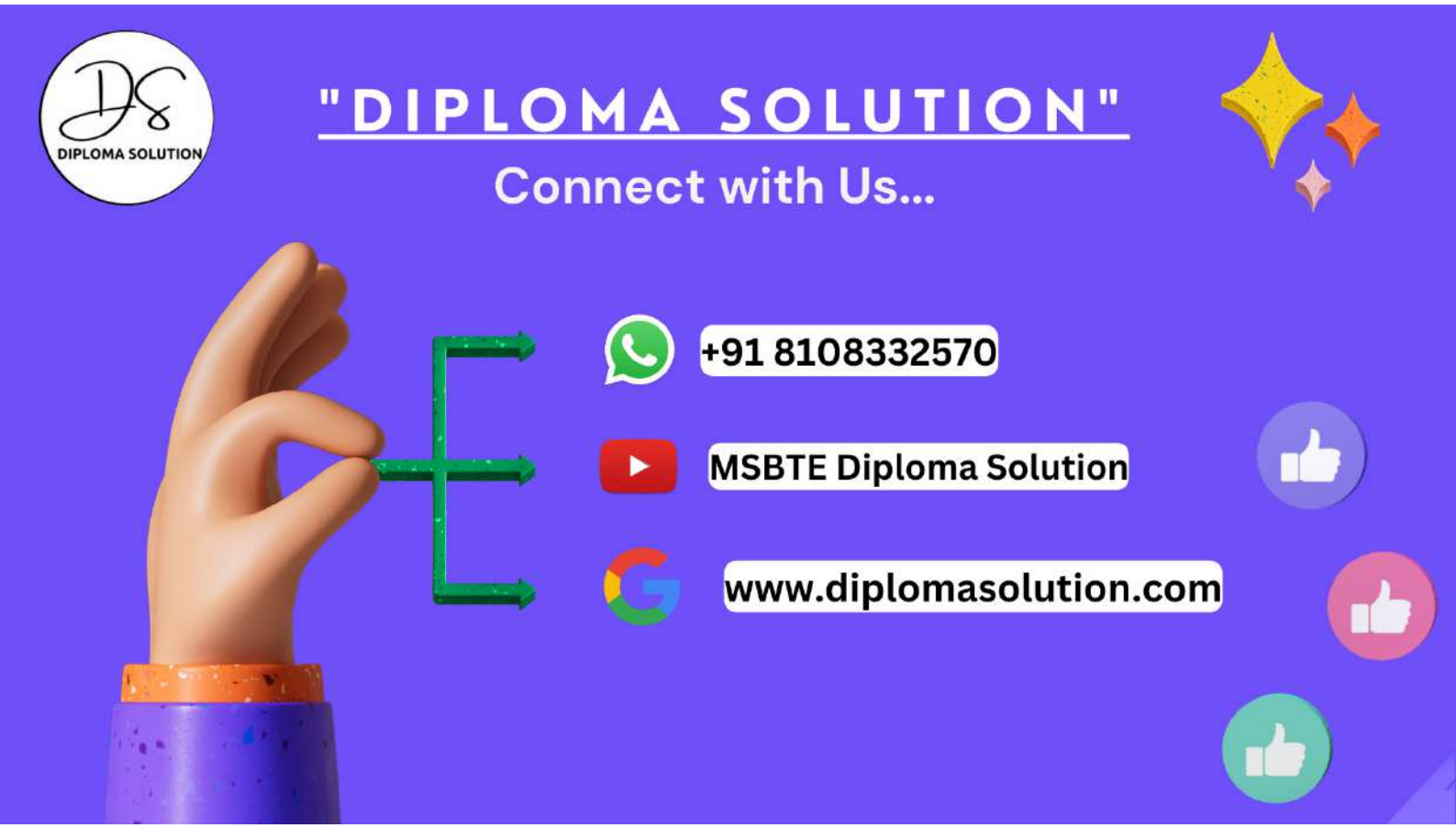
1. Request: The process requests the resources it needs. If the resources are not available at that time, the process must wait until they are.
2. Use: Once the process has all its resources, it can perform its function.
3. Release: When the process is finished with the resources, it releases them.

It's important to note that not all resources are immediately reusable; for instance, a printer cannot be reallocated to another process until the current print job is finished.

This system model can help to understand how resources are requested, used, and released, and it forms the basis for the characterization and analysis of deadlock scenarios.

3. **Deadlock Detection and Recovery**: If deadlocks cannot be prevented or avoided, they can be allowed to occur, detected, and then recovered. A detection algorithm is used to determine whether a deadlock has occurred. If a deadlock is detected, the system must recover from the deadlock, typically by terminating some or all of the processes in the deadlock or by preempting resources from some processes.

Each of these strategies has its own trade-offs in terms of resource utilization and system throughput. The appropriate strategy will depend on the specific details of the system and its workload.


DIPLOMA SOLUTION
"DIPLOMA SOLUTION"
Connect with Us...
+91 8108332570
MSBTE Diploma Solution
www.diplomasolution.com