Unit-02

Process Management

The concepts of processes and threads are fundamental to understanding how modern operating systems manage and execute tasks. Both processes and threads are units of execution, but they differ in their scope and characteristics.

- ## Process:

➢ **Definition:**

- A process is an independent program in execution. It consists of its own address space, code, data, and system resources.
- Processes are typically isolated from each other, meaning that the memory space of one process is not directly accessible by another process.

➢ **Characteristics:**

- **Isolation:** Processes are independent entities, and they do not share memory space with other processes.
- **Resource Allocation:** Each process has its own set of system resources, including file handles, network connections, and other resources.
- **Communication:** Inter-process communication (IPC) is required for processes to communicate with each other.

➢ **Advantages:**

- **Isolation:** Faults or errors in one process do not directly affect others.
- **Security:** Processes provide a higher level of security, as they cannot interfere with each other's memory.

➢ **Disadvantages:**

- **Overhead:** Creating and managing processes can be resource-intensive.
- **Communication Overhead:** IPC mechanisms can introduce overhead.

❖ Thread:

➢ **Definition:**

- A thread is the smallest unit of execution within a process. Multiple threads within a single process share the same resources, such as memory space and file handles.
- Threads within the same process can communicate more easily than processes.

➢ **Characteristics:**

- **Shared Resources:** Threads within the same process share resources, which makes communication between threads more straightforward.
- **Lightweight:** Threads are lighter-weight compared to processes, as they share resources and do not require separate memory space.

➢ **Advantages:**

- **Efficiency:** Threads are more efficient than processes in terms of resource utilization.
- **Communication:** Threads within the same process can communicate directly using shared memory.

➢ **Disadvantages:**

- **Security:** Since threads within a process share the same memory space, a bug or error in one thread can potentially affect the entire process.

- **Complexity:** Managing threads requires careful synchronization to avoid data corruption or race conditions.

Key Differences:

★ **Isolation:**

– Processes are isolated from each other.

– Threads share the same memory space within a process.

★ **Resource Overhead:**

– Processes have higher resource overhead.

– Threads have lower resource overhead.

★ **Communication:**

– Processes communicate using IPC mechanisms.

– Threads within the same process can communicate through shared memory.

★ **Creation Time:**

– Creating a new process is generally more time-consuming.

– Creating a new thread is quicker.

Operation on Processes

➢ Processes in an operating system perform various operations to execute tasks and interact with the system. Here are different operations associated with processes:

1. **Creation:**

→ Processes are created during system initialization or in response to user requests.

→ The operating system may provide system calls like **fork()** (Unix-like systems) or **CreateProcess()** (Windows) to create new processes.

2. **Termination:**

→ Processes can terminate voluntarily or involuntarily.

→ Voluntary termination occurs when a process completes its execution or explicitly requests termination.

→ Involuntary termination may happen due to errors or external signals.

3. **Scheduling:**

→ The operating system scheduler determines the order in which processes are executed.

→ Scheduling algorithms manage the allocation of CPU time to processes.

4. **Communication:**

→ Processes may need to communicate with each other for coordination or data exchange.

→ Inter-Process Communication (IPC) mechanisms, such as pipes, sockets, and message passing, facilitate communication between processes.

5. **Synchronization:**

→ Processes often need to synchronize their activities to avoid conflicts and data inconsistencies.

→ Semaphores, mutexes, and other synchronization primitives help manage access to shared resources.

6. **Memory Management:**

→ Processes require memory for code, data, and stack.

→ Memory management operations include allocating memory space, loading program code into memory, and deallocating memory when a process terminates.

7. **I/O Operations:**

→ Processes perform input and output operations to interact with the external environment.

→ System calls like **read()** and **write()** are used for I/O operations.

8. **Resource Allocation:**

→ Processes need various system resources, such as file handles, network connections, and CPU time.

→ Resource allocation ensures fair distribution of resources among competing processes.

9. **Error Handling:**

→ Processes must handle errors and exceptions gracefully.

→ Error handling mechanisms, such as signals or exceptions, allow processes to respond to exceptional conditions.

10. **Security:**

→ Processes operate within security boundaries defined by the operating system.

→ Security-related operations include authentication, authorization, and process isolation.

11. **Inter-Process Synchronization:**

→ Processes may synchronize their execution to achieve coordination.

→ Techniques like message passing, semaphores, and locks facilitate synchronization between processes.

12. **Priority Management:**

→ Processes may be assigned priorities to influence the order in which they are scheduled.

→ Priority management operations involve adjusting the priority of processes based on their characteristics or system conditions.

13. **Context Switching:**

→ When the operating system switches between executing processes, it performs a context switch.

→ Context switching involves saving the state of the current process and loading the state of the next process to be executed.

Interprocess Communication

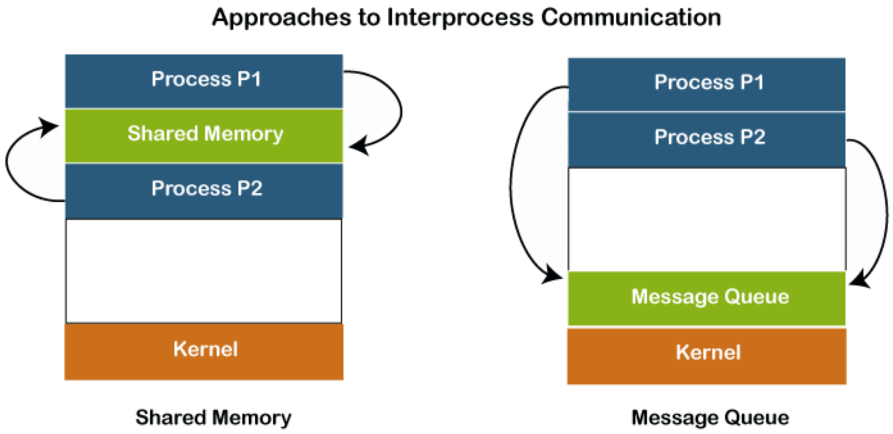
➢ Inter-Process Communication (IPC) is a set of mechanisms that allow different processes to communicate and synchronize with each other. IPC is essential for coordinating activities, sharing data, and enabling collaboration between independent processes running on the same or different machines. Various IPC mechanisms exist, and the choice depends on the specific requirements of the application.

Independent process:

- A process is independent if it can’t affect or be affected by another process.

Co-operating Process:

- A process is co-operating if it can affects other or be affected by the other process.
- Any process that shares data with other process is called co-operating process.



Process States

- In an operating system, a process goes through different states during its lifecycle. The process states represent the various stages a process goes through from creation to termination. The typical process states are as follows:

1. **New:**

→ The process is being created or initialized.

→ In this state, the operating system is allocating resources and setting up initial values for the process.

2. **Ready:**

→ The process is loaded into the main memory and is waiting to be assigned to a processor.

→ It is in a queue of processes that are ready to run but are waiting for the CPU.

3. **Running:**

→ The processor is actively executing the process's instructions.

→ In a multiprogramming system, multiple processes may be in the running state, with the operating system using a scheduling algorithm to switch between them.

4. **Blocked (Waiting):**

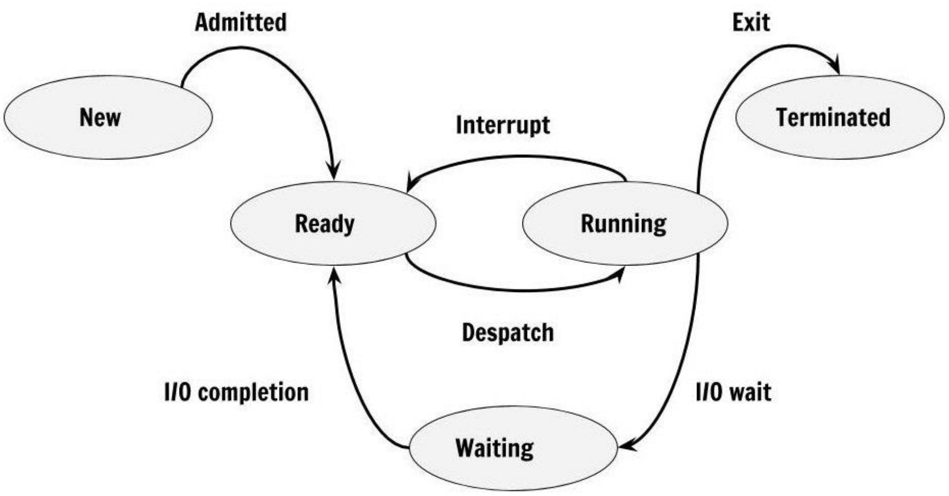
→ The process is waiting for a particular event or condition before it can proceed.

→ Common reasons for a process to be blocked include waiting for I/O operations, user input, or the completion of another process.

5. **Terminated (Exit):**

→ The process has finished execution, either because it completed its task or due to an error.

→ Resources allocated to the process are released, and the process is removed from the system.



Process Synchronization: critical section problems and solutions

- Process synchronization is crucial in concurrent programming to ensure that multiple processes or threads execute in a coordinated manner without interfering with each other's data. The critical section problem is a classic synchronization issue that arises when multiple processes or threads share a common resource and need to access it simultaneously. The objective is to prevent interference and data corruption caused by concurrent access to the shared resource. Here's an overview of the critical section problem and some common solutions:

Critical Section Problem:

1. **Critical Section:**

→ The critical section is a portion of code that accesses shared resources and must be executed atomically.

2. **Requirements:**

→ Mutual Exclusion: Only one process can be in the critical section at a time.

→ Progress: If no process is in the critical section and some processes wish to enter, only those processes not in the remainder section can participate in deciding which will enter next.

→ Bounded Waiting: There exists a bound on the number of times other processes are allowed to enter the critical section after a process has made a request to enter.

Solutions to the Critical Section Problem:

1. **Mutual Exclusion with Locks:**

- **Semaphore/Mutex:**

→ A semaphore or mutex is used to achieve mutual exclusion.

→ Processes acquire the semaphore before entering the critical section and release it afterward.

→ This ensures that only one process at a time can hold the semaphore and, therefore, enter the critical section.

2. **Bakery Algorithm:**

→ Each process is assigned a unique number.

→ Processes enter the critical section based on their assigned number.

→ The process with the lowest number enters first in case of a tie.

3. **Peterson's Algorithm:**

→ Designed for two processes.
→ Uses shared variables and flags to implement a turn-taking mechanism.
→ Ensures that only one process can be in the critical section at a time.

4. **Test-and-Set Instruction:**

→ A hardware-based solution using a special atomic instruction.
→ The "test-and-set" instruction atomically sets a variable to a new value and returns its previous value.
→ This can be used to implement locks for mutual exclusion.

5. **Semaphore-based Solutions:**

→ Generalization of the mutex concept.
→ A semaphore maintains a count and supports operations like **wait** (decrement count) and **signal** (increment count).
→ A binary semaphore can be used as a mutex.

6. **Monitor:**

→ A higher-level synchronization construct.
→ A monitor is an abstract data type that encapsulates shared resources and procedures to operate on them.
→ Provides mutual exclusion automatically.

7. **Readers-Writers Problem:**

→ Extends the critical section problem to distinguish between readers (processes that only read shared data) and writers (processes that modify shared data).
→ Different solutions exist for favoring readers or writers based on application requirements.

Peterson's Solution

Overview:

→ Named after Gary Peterson, this algorithm provides a solution to the critical section problem for two processes.
→ It ensures mutual exclusion, meaning that only one process can be in the critical section at a time.

Key Features:

→ Uses shared variables, **flag** and **turn**, to coordinate access to the critical section.
→ Processes set their flag to indicate their interest and set **turn** to indicate whose turn it is.
→ The process wanting to enter the critical section waits until it's its turn and the other process is not interested.

Limitations:

→ Primarily designed for a two-process scenario.

Mutex Locks:

Overview:

→ A mutex (short for mutual exclusion) is a synchronization primitive used to protect critical sections.
→ It allows only one thread or process to access a specific resource at a time.

Key Features:

- → A thread or process must acquire the mutex before entering the critical section and release it afterward.
- → Provides a simple and effective way to achieve mutual exclusion.
- → Often implemented using hardware instructions like **test-and-set** or atomic compare-and-swap.

Use Cases:

- → Critical section protection in multithreaded or multiprocess programs.

Semaphores:

Overview:

- → Semaphores are synchronization objects that can be used to control access to a shared resource by multiple processes in a concurrent system.

Key Features:

- → Semaphores have an associated non-negative integer value.
- → Operations include **wait** (decrement) and **signal** (increment).
- → Useful for both mutual exclusion and signaling between processes.

Types:

- → **Binary Semaphore:** Acts like a mutex with values 0 and 1.
- → **Counting Semaphore:** Can have values greater than 1.

Use Cases:

- → Coordination and synchronization in concurrent programs.
- → Implementing solutions to classic synchronization problems.

Monitors:

Overview:

- → A high-level synchronization construct that combines data (shared resource) and procedures (methods) that operate on the data.

Key Features:

- → Encapsulates shared resources and their associated procedures.
- → Provides mutual exclusion implicitly.
- → Ensures that only one process can execute a method at a time.

Use Cases:

- → Higher-level abstraction for synchronization.
- → Simplifies the design of concurrent programs by encapsulating synchronization details.

CPU Scheduling Concepts

- ➢ CPU scheduling is a crucial aspect of operating system design that involves selecting the next process or thread from the ready queue to execute on the CPU. The goal is to optimize resource utilization, enhance system performance, and provide fair access to the CPU for all processes.

Scheduling Criteria

- ➢ Scheduling criteria refer to the metrics and factors used to evaluate and compare the performance of different CPU scheduling algorithms. The primary goals are to optimize resource utilization, enhance system performance, and provide a fair and efficient execution environment for processes. Here are key scheduling criteria

- **CPU Utilization:**
- Maximizing the percentage of time the CPU is actively executing a process.
- High CPU utilization indicates efficient use of computing resources.
- **Throughput:**
- The number of processes completed in a unit of time.
- Maximizing throughput ensures efficient execution and task completion.
- **Turnaround Time:**
- The total time taken to execute a process from submission to completion.
- It includes waiting time in the ready queue and execution time.
- **Waiting Time:**
- The total time a process spends waiting in the ready queue before getting CPU time.
- Minimizing waiting time enhances overall system responsiveness.
- **Response Time:**
- The time elapsed between submitting a request and receiving the first response.
- Important for interactive systems to provide a quick user interface response.

Scheduling Algorithms: First come First Serve (FCFS), Shortest Job First (SJF), Shortest Remaining Time First (SRTF), Round Robin

1. First- Come, First- serve Scheduling (FCFS):

- This is the simplest of all the scheduling algorithms.
- Basic principle of this algorithm is to allocate the CPU in the order in which the process arrives.
- Its implementation involves a ready queue that works in First - In- First- Out order.
- When the CPU is free, it is assigned to a process which is in front of the ready queue.
- FCFS is a non-preemptive scheduling algorithm because the CPU has been allocated to a process that keeps the CPU busy until it’s released.

Advantages

- Simple algorithm and easy to implement.
- Suitable specially, for batch system.
- Does not give priority to any random important tasks first so it’s a fair scheduling.

Disadvantages

- FCFS results in convoy effect which means if a process with higher burst time comes first in the ready queue then the processes with lower burst time may get blocked and may not be able to get the CPU if the higher burst time task takes time forever.
- Not suitable for time sharing system such as UNIX.

Consider the set of 5 processes whose arrival time and burst time are given below:

2. Shortest Job First

- The basic principle of this algorithm is to allocate the CPU to the process with least CPU- burst time.
- The process is available in the ready queue.

- CPU is always assigned to the process with the least CPU burst time requirement.
- If the new job needs less time to finish than the current process, The current process is suspended, and the new job is started.
- Please note that if two processes have the same CPU burst time then FCFS is used to break the tie.
- This algorithm can be either preemptive or non-preemptive.
- The preemptive SJF scheduling algorithm is sometimes called Shortest Remaining Time First (SRTF) Scheduling.
- SJF algorithm works optimally only when the exact future execution times of jobs or processes are known at the time of scheduling.

Advantages of SJF Scheduling

- Since this algorithm gives minimum average waiting time so it is an optimal algorithm.

Disadvantages of SJF Scheduling

- It is difficult to know the length of next CPU burst time.
- Big jobs are waiting for CPU. This results in aging problem.

3. Shortest Remaining Time First

- This algorithm is a preemptive scheduling algorithm.
- In SRTF, The short term scheduler always chooses the process that has the shortest remaining processing time.
- When a new process joins the ready queue, the short term scheduler compares the remaining time of executing process and new process, and if the new process has the least CPU burst time, the scheduler selects that job and allocates CPU else it continues with the old process.
- Once all the processes are available in the ready queue, No preemption will be done and the algorithm will work as SJF scheduling.
- The SRTF algorithm involves more overheads than the Shortest job first (SJF)scheduling, because in SRTF OS is required frequently in order to monitor the CPU time of the jobs in the READY queue and to perform context switching.
- In the Process Control Block, the context of the process is saved, when the process is removed from the execution and when the next process is scheduled. The PCB is accessed on the next execution of this process.

Advantages of SRTF

- The main advantage of the SRTF algorithm is that it makes the processing of the jobs faster than the SJF algorithm, mentioned it’soverhead charges are not counted.

Disadvantages of SRTF

- In SRTF, the context switching is done a lot more times than in SJN due to more consumption of the CPU&#39;s valuable time for processing. The consumed time of CPU then adds up to its processing time and which then diminishes the advantage of fast processing of this algorithm.

4. **Round Robin**

- One of the oldest, simplest, fairest and most widely used algorithm is round robin (RR).
- Round Robin is a CPU scheduling algorithm where each process is assigned a fixed time slot in a cyclic way.
- The period of time for which a process or job is allowed to run in a pre-emptive method is called time quantum or time slice. The CPU time is divided into slices.
- Each process or job present in the ready queue is assigned the CPU for single time quantum, if the execution of the process is completed during that time then the process will end else the process will go back to the end of the ready queue and wait for its next turn to complete the execution.
- Round Robin Scheduling is preemptive (at the end of time-slice) therefore it is effective in timesharing environments in which the system needs to guarantee reasonable response times for interactive users.
- This algorithm is similar to preemptive version of FCFS as process are executed in first come first serve manner.
- The performance of RR algorithm depends on given factor:

✓ Size of Time Quantum

✓ Number of Context Switch

Deadlock

➢ **Deadlock is a situation in a computer system where two or more processes are unable to proceed because each is waiting for the other to release a resource. There are several concepts related to deadlock that help in understanding and managing this situation:**

1. **Deadlock Prevention:**

- **Definition:** Strategies and techniques to avoid the occurrence of deadlock.
- **Deadlock Connection:** Prevention methods address the four necessary conditions (mutual exclusion, hold and wait, no preemption, and circular wait) to eliminate deadlock scenarios.

2. **Deadlock avoidance:**

→ Deadlock avoidance is a strategy used to prevent the occurrence of deadlocks in a computer system by carefully managing resource allocations. The goal is to dynamically analyze the resource needs of processes and make decisions that ensure a safe state, where deadlock cannot occur. Key techniques for deadlock avoidance include the Banker's Algorithm and dynamic allocation policies.

3. **Deadlock Detection:**

- **Definition:** Techniques for identifying the presence of a deadlock after it has occurred.
- **Deadlock Connection:** Detection methods involve periodically examining the resource allocation graph or system state to identify circular waits and resolve deadlocks.

4. **Deadlock Recovery:**

❖ Definition: Procedures for recovering from a deadlock after detection.

❖ Deadlock Connection: Recovery methods may involve terminating processes, releasing resources, or rolling back transactions to break the deadlock.

Q. Explain different methods for handling deadlocks

→ Handling deadlocks involves implementing strategies to detect, prevent, or recover from deadlock situations in a computer system. Here are different methods for handling deadlocks:

1. Deadlock Prevention:

Overview:

- Focuses on designing the system in a way that avoids the occurrence of deadlock by eliminating one or more necessary conditions.

Methods:

1. **Mutual Exclusion:**
- Allow multiple processes to share resources without mutually excluding each other.
2. **Hold and Wait:**
- Require a process to request and acquire all necessary resources at once, preventing it from holding a resource while waiting for another.
3. **No Preemption:**
- Allow the operating system to forcibly preempt resources from a process if needed.
4. **Circular Wait:**
- Impose a total ordering of all resource types and require that each process requests resources in increasing order.

2. Deadlock Detection and Recovery:

Overview:

→ Monitors the system to detect the occurrence of deadlocks and takes corrective actions to recover from them.

Methods:

- **Periodic Deadlock Detection:**

→ System periodically checks for the presence of deadlocks.

- **Single-Instance Resource Allocation Graph:**

→ Maintain a graph representing processes, resources, and their relationships. Detect a cycle in the graph to identify deadlocks.

- **Multiple-Instance Resource Allocation Graph:**

→ Extends the single-instance approach to handle multiple instances of resources.

- **Timeouts:**

→ Set timeouts for resource requests. If a process doesn't get the requested resources within a specified time, it is assumed to be in a deadlock and is terminated.

- **Abort and Rollback:**

→ Identify and abort one or more processes to break the deadlock. Rollback their state to a consistent checkpoint.

3. Deadlock Avoidance:

Overview:

- Involves making dynamic decisions about resource allocations based on the current state of the system to avoid deadlock.

Methods:

- **Banker's Algorithm:**

→ Ensures that resource allocations will not lead to an unsafe state.

→ Analyzes the system's state before granting resource requests.

- **Wait-Die and Wound-Wait Scheme (Database Management):**

→ Processes are categorized as "young" or "old" based on their timestamp.

→ Young processes requesting older processes' resources result in "wait-die" or "wound-wait" decisions.

4. Combined Approaches:

Overview:

→ Some systems use a combination of prevention, detection, and recovery strategies for a comprehensive approach.

Methods:

- **Hybrid Schemes:**

→ Combine elements of prevention, detection, and recovery.

→ Example: Using prevention strategies to reduce the likelihood of deadlocks and detection/recovery mechanisms to handle rare occurrences.

Considerations:

- **Performance Impact:**

→ Some methods, such as periodic deadlock detection, may have performance overhead.

- **Resource Utilization:**

→ Strategies like deadlock prevention may lead to conservative resource allocation, potentially underutilizing resources.

THE END