# DEPARTMENT OF AIML

## 23CST202- OPERATING SYSTEMS

## II YEAR IV SEM AIML-B

UNIT 2-PROCESS SCHEDULING AND SYNCHRONIZATION

TOPIC –REAL TIME SCHEDULING,ALGORITHM EVALUATION AND PROCESS SYNCHRONIZATION

# REAL TIME SCHEDULING

Real-time systems are systems that carry real-time tasks. These tasks need to be performed immediately with a certain degree of urgency. In particular, these tasks are related to control of certain events (or) reacting to them. Real-time tasks can be classified as hard real-time tasks and soft real-time tasks.

A hard real-time task must be performed at a specified time which could otherwise lead to huge losses. In soft real-time tasks, a specified deadline can be missed. This is because the task can be rescheduled (or) can be completed after the specified time,

In real-time systems, the scheduler is considered as the most important component which is typically a short-term task scheduler. The main focus of this scheduler is to reduce the response time associated with each of the associated processes instead of handling the deadline.

If a preemptive scheduler is used, the real-time task needs to wait until its corresponding tasks time slice completes. In the case of a non-preemptive scheduler, even if the highest priority is allocated to the task, it needs to wait until the completion of the current task. This task can be slow (or) of the lower priority and can lead to a longer wait.

A better approach is designed by combining both preemptive and non-preemptive scheduling. This can be done by introducing time-based interrupts in priority based systems which means the currently running process is interrupted on a time-based interval and if a higher priority process is present in a ready queue, it is executed by preempting the current process.

**Advantages of Scheduling in Real-Time Systems:**

- **Meeting Timing Constraints:** Scheduling ensures that real-time tasks are executed within their specified timing constraints. It guarantees that critical tasks are completed on time, preventing potential system failures or losses.
- **Resource Optimization:** Scheduling algorithms allocate system resources effectively, ensuring efficient utilization of processor time, memory, and other resources. This helps maximize system throughput and performance.
- **Priority-Based Execution:** Scheduling allows for priority-based execution, where higher-priority tasks are given precedence over lower-priority tasks. This

ensures that time-critical tasks are promptly executed, leading to improved system responsiveness and reliability.

- **Predictability and Determinism:** Real-time scheduling provides predictability and determinism in task execution. It enables developers to analyze and guarantee the worst-case execution time and response time of tasks, ensuring that critical deadlines are met.
- **Control Over Task Execution:** Scheduling algorithms allow developers to have fine-grained control over how tasks are executed, such as specifying task priorities, deadlines, and inter-task dependencies. This control facilitates the design and implementation of complex real-time systems.

**Disadvantages of Scheduling in Real-Time Systems:**

- **Increased Complexity:** Real-time scheduling introduces additional complexity to system design and implementation. Developers need to carefully analyze task requirements, define priorities, and select suitable scheduling algorithms. This complexity can lead to increased development time and effort.
- **Overhead:** Scheduling introduces some overhead in terms of context switching, task prioritization, and scheduling decisions. This overhead can impact system performance, especially in cases where frequent context switches or complex scheduling algorithms are employed.
- **Limited Resources:** Real-time systems often operate under resource-constrained environments. Scheduling tasks within these limitations can be challenging, as the available resources may not be sufficient to meet all timing constraints or execute all tasks simultaneously.
- **Verification and Validation:** Validating the correctness of real-time schedules and ensuring that all tasks meet their deadlines require rigorous testing and verification techniques. Verifying timing constraints and guaranteeing the absence of timing errors can be a complex and time-consuming process.
- **Scalability:** Scheduling algorithms that work well for smaller systems may not scale effectively to larger, more complex real-time systems. As the number of tasks and system complexity increases, scheduling decisions become more challenging and may require more advanced algorithms or approaches.

# ALGORITHM EVALUATION

**Evaluation of Process Scheduling Algorithms**

The first thing we need to decide is how we will evaluate the algorithms. To do this we need to decide on the relative importance of the factors we listed above (Fairness, Efficiency, Response Times, Turnaround and Throughput). Only once we have decided on our evaluation method can we carry out the evaluation.

**Deterministic Modeling**

This evaluation method takes a predetermined workload and evaluates each algorithm using that workload.

Assume we are presented with the following processes, which all arrive at time zero.

| Process | Burst Time |
|---|---|

| | |
|---|---|
| P1 | 9 |
| P2 | 33 |
| P3 | 2 |
| P4 | 5 |
| P5 | 14 |

Which of the following algorithms will perform best on this workload?
First Come First Served (FCFS), Non Preemptive Shortest Job First (SJF) and Round Robin (RR). Assume a quantum of 8 milliseconds.
Before looking at the answers, try to calculate the figures for each algorithm.
The advantages of deterministic modeling is that it is exact and fast to compute. The disadvantage is that it is only applicable to the workload that you use to test. As an example, use the above workload but assume P1 only has a burst time of 8 milliseconds. What does this do to the average waiting time?
Of course, the workload might be typical and scale up but generally deterministic modeling is too specific and requires too much knowledge about the workload.

**Queuing Models**

Another method of evaluating scheduling algorithms is to use queuing theory. Using data from real processes we can arrive at a probability distribution for the length of a burst time and the I/O times for a process. We can now generate these times with a certain distribution.
We can also generate arrival times for processes (arrival time distribution).
If we define a queue for the CPU and a queue for each I/O device we can test the various scheduling algorithms using queuing theory.
Knowing the arrival rates and the service rates we can calculate various figures such as average queue length, average wait time, CPU utilization etc.
One useful formula is ***Little's Formula.***

$n = \lambda w$

Where

n is the average queue length
$\lambda$ is the average arrival rate for new processes (e.g. five a second)
w is the average waiting time in the queue

Knowing two of these values we can, obviously, calculate the third. For example, if we know that eight processes arrive every second and there are normally sixteen processes in the queue we can compute that the average waiting time per process is two seconds.
The main disadvantage of using queuing models is that it is not always easy to define realistic distribution times and we have to make assumptions. This results in the model only being an approximation of what actually happens.

**Simulations**

Rather than using queuing models we simulate a computer. A Variable, representing a clock is incremented. At each increment the state of the simulation is updated.
Statistics are gathered at each clock tick so that the system performance can be analysed.
The data to drive the simulation can be generated in the same way as the queuing model, although this leads to similar problems.
Alternatively, we can use trace data. This is data collected from real processes on real machines and is fed into the simulation. This can often provide good results and good

comparisons over a range of scheduling algorithms.
However, simulations can take a long time to run, can take a long time to implement and the trace data may be difficult to collect and require large amounts of storage.

**Implementation**

The best way to compare algorithms is to implement them on real machines. This will give the best results but does have a number of disadvantages.

· It is expensive as the algorithm has to be written and then implemented on real hardware.

· If typical workloads are to be monitored, the scheduling algorithm must be used in a live situation. Users may not be happy with an environment that is constantly changing.

· If we find a scheduling algorithm that performs well there is no guarantee that this state will continue if the workload or environment changes.

# PROCESS SYNCHRONIZATION

**Process Synchronization** is used in a computer system to ensure that multiple processes or threads can run concurrently without interfering with each other.

The main objective of process synchronization is to ensure that multiple processes access shared resources without interfering with each other and to prevent the possibility of inconsistent data due to concurrent access. To achieve this, various synchronization techniques such as semaphores, monitors, and critical sections are used.

In a multi-process system, synchronization is necessary to ensure data consistency and integrity, and to avoid the risk of deadlocks and other synchronization problems. Process synchronization is an important aspect of modern operating systems, and it plays a crucial role in ensuring the correct and efficient functioning of multi-process systems.

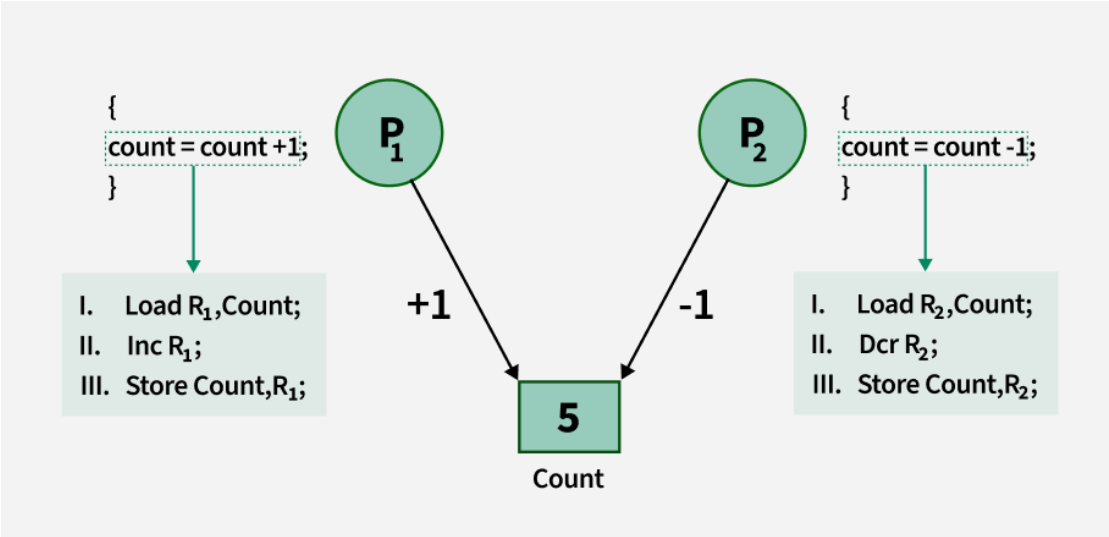
On the basis of synchronization, processes are categorized as one of the following two types:

- **Independent Process**: The execution of one process does not affect the execution of other processes.
- **Cooperative Process**: A process that can affect or be affected by other processes executing in the system.

Process synchronization problem arises in the case of Cooperative processes also because resources are shared in Cooperative processes.

**Process Synchronization**

Process Synchronization is the coordination of execution of multiple processes in a multi-process system to ensure that they access shared resources in a controlled and predictable manner. It aims to resolve the problem of race conditions and other synchronization issues in a concurrent system.




Lack of Synchronization in Inter Process Communication Environment leads to following problems:

1. **Inconsistency:** When two or more processes access shared data at the same time without proper synchronization. This can lead to conflicting changes, where one process's update is overwritten by another, causing the data to become unreliable and incorrect.
2. **Loss of Data:** Loss of data occurs when multiple processes try to write or modify the same shared resource without coordination. If one process overwrites the data before another process finishes, important information can be lost, leading to incomplete or corrupted data.
3. **Deadlock:** Lack of Synchronization leads to Deadlock which means that two or more processes get stuck, each waiting for the other to release a resource. Because none of the processes can continue, the system becomes unresponsive and none of the processes can complete their tasks.

**Types of Process Synchronization**

The two primary type of process Synchronization in an Operating System are:

1. **Competitive:** Two or more processes are said to be in Competitive Synchronization if and only if they compete for the accessibility of a shared resource.
   **Lack of Synchronization among Competing process** may lead to either Inconsistency or Data loss.
2. **Cooperative:** Two or more processes are said to be in Cooperative Synchronization if and only if they get affected by each other i.e. execution of one process affects the other process.
   **Lack of Synchronization among Cooperating process** may lead to Deadlock.

**Example:**

Let consider a Linux code:

```
>>ps/grep "chrome"/wc
```

- ps command produces list of processes running in linux.
- grep command find/count the lines form the output of the ps command.
- wc command counts how many words are in the output.

Therefore, three processes are created which are ps, grep and wc. grep takes input from ps and wc takes input from grep.

From this example, we can understand the concept of cooperative processes, where some processes produce and others consume, and thus work together. This type of problem must be handled by the operating system, as it is the manager.

**Conditions That Require Process Synchronization**

1. **Critical Section:** It is that part of the program where shared resources are accessed. Only one process can execute the critical section at a given point of time. If there are no shared resources, then no need of synchronization mechanisms.
2. **Race Condition:** It is a situation wherein processes are trying to access the critical section and the final result depends on the order in which they finish their update. Process Synchronization mechanism need to ensure that instructions are being executed in a required order only.
3. **Pre Emption:** Preemption is when the operating system stops a running process to give the CPU to another process. This allows the system to make sure that important tasks get enough CPU time. This is important as mainly issues arise when a process has not finished its job on shared resource and got preempted. The other process might end up reading an inconsistent value if process synchronization is not done.

**What is Race Condition?**

A race condition is a situation that may occur inside a critical section. This happens when the result of multiple process/thread execution in the critical section differs according to the order in which the threads execute. Race conditions in critical sections can be avoided if the critical section is treated as an atomic instruction. Also, proper thread synchronization using locks or atomic variables can prevent race conditions.

Let us consider the following example.

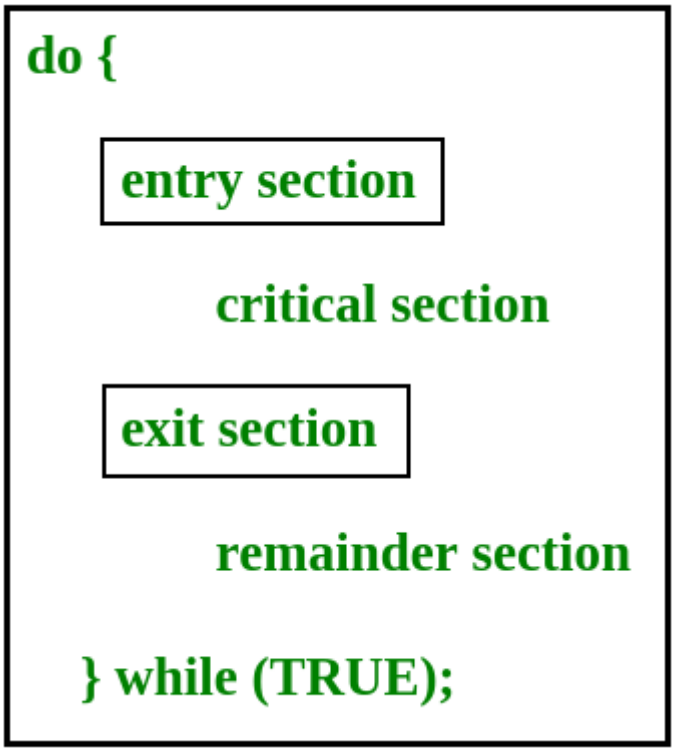
- There is a shared variable balance with value 100.
- There are two processes deposit(10) and withdraw(10). The deposit process does balance = balance + 10 and withdraw process does balance = balance – 10.
- Suppose these processes run in an interleaved manner. The deposit() fetches the balance as 100, then gets preempted.
- Now withdraw() get scheduled and makes balance 90.
- Finally deposit is rescheduled and makes the value as 110. This value is not correct as the balance after both operations should be 100 only

We can not notice that the different segments of two processes running in different order would give different values of balance.

**Critical Section Problem**

A critical section is a code segment that can be accessed by only one process at a time. The critical section contains shared variables that need to be synchronized to maintain the consistency of data variables. Sothe critical section problem means designing a way for cooperative processes to access shared resources without creating data inconsistencies.

In the above example, the operations that involve balance variable should be put in critical sections of both deposit and withdraw.




In the entry section, the process requests for entry in the **Critical Section.**

Any solution to the critical section problem must satisfy three requirements:

- **Mutual Exclusion**: If a process is executing in its critical section, then no other process is allowed to execute in the critical section.
- **Progress**: If no process is executing in the critical section and other processes are waiting outside the critical section, then only those processes that are not executing in their remainder section can participate in deciding which will enter the critical section next, and the selection cannot be postponed indefinitely.
- **Bounded Waiting**: A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.

Critical section Problem – Solutions

**Classical IPC Problems**

Various classical Inter-Process Communication (IPC) problems include:

- Producer Consumer Problem
- Readers-Writers Problem
- Dining Philosophers Problem

**Producer Consumer Problem**

The Producer-Consumer Problem is a classic example of process synchronization. It describes a situation where two types of processes producers and consumers share a common, limited-size buffer or storage.

- **Producer**: A producer creates or generates data and puts it into the shared buffer. It continues to produce data as long as there is space in the buffer.
- **Consumer**: A consumer takes data from the buffer and uses it. It continues to consume data as long as there is data available in the buffer.

The challenge arises because both the producer and the consumer need to access the same buffer at the same time, and if they do not properly synchronize their actions, issues can occur.

**Key Problems in the Producer-Consumer Problem:**

1. **Buffer Overflow**: If the producer tries to add data when the buffer is full, there will be no space for new data, causing the producer to be blocked.
2. **Buffer Underflow**: If the consumer tries to consume data when the buffer is empty, it has nothing to consume, causing the consumer to be blocked.

Producer-Consumer Problem – Solution (using Semaphores)

**Readers-Writers Problem**

The Readers-Writers Problem is a classic synchronization problem where multiple processes are involved in reading and writing data from a shared resource. This problem typically involves two types of processes:

- **Readers**: These processes only read data from the shared resource and do not modify it.
- **Writers**: These processes modify or write data to the shared resource.

The challenge in the Reader-Writer problem is to allow multiple readers to access the shared data simultaneously without causing issues. However, only **one writer** should be allowed to write at a time, and no reader should be allowed to read while a writer is writing. This ensures the integrity and consistency of the data.

Readers-Writers Problem – Solution (Readers Preference Solution)
Readers-Writers Problem – Solution (Writers Preference Solution)

**Dining Philosophers Problem**

The Dining Philosophers Problem is a well-known example that shows the difficulties of sharing resources and preventing deadlock when multiple processes are involved. The problem involves a set of philosophers sitting around a dining table. Each philosopher thinks deeply, but when they are hungry, they need to eat. However, to eat, they must pick up two forks from the table, one from the left and one from the right.

**Problem Setup:**

- There are **five philosophers** sitting around a circular table.
- Each philosopher has a plate of food in front of them and a fork to their left and right.
- A philosopher needs both forks to eat. If they pick up a fork, they hold it until they finish eating.
- After eating, they put down both forks and start thinking again.

The problem arises when multiple philosophers try to pick up forks at the same time, which can lead to a situation where each philosopher holds one fork but cannot get the second fork, leading to a **deadlock**. Additionally, there's a risk of **starvation** if some philosophers are continually denied the opportunity to eat.

Dining Philosophers Problem – Solution (using Semaphores)

**Advantages of Process Synchronization**

- Ensures data consistency and integrity
- Avoids race conditions
- Prevents inconsistent data due to concurrent access
- Supports efficient and effective use of shared resources

**Disadvantages of Process Synchronization**

- Adds overhead to the system
- This can lead to performance degradation
- Increases the complexity of the system
- Can cause  deadlock if not implemented properly.