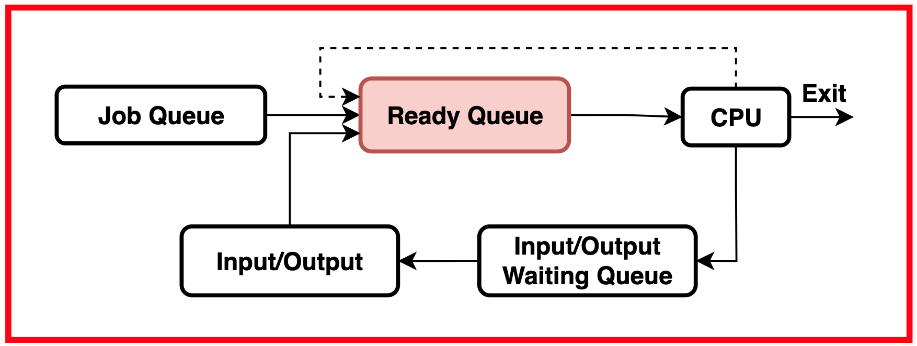
PROCESS MANAGEMENT

1. **Process Scheduling Algorithm (Pre-emptive and Non-Preemptive)**

**Process Scheduling**

It handles the **efficient allocation of system resources** like CPU time and memory to multiple processes or threads running at the same time. It ensures processes are executed in a way that **optimises system performance**, resource utilisation, and user responsiveness.





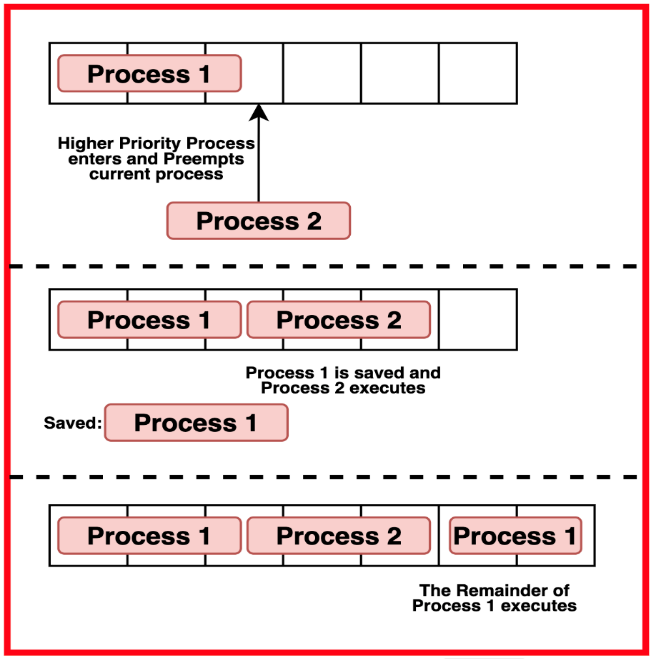
This mechanism allows the operating system to select a process from the **ready queue** and give it CPU time for execution. It decides which process should run next, how long it should run, and when to preempt or suspend it to let other processes execute.

**Preemptive Scheduling**

In **Preemptive Scheduling**, the operating system can **interrupt a currently running process** and give the CPU to another process. This can happen if a higher-priority process arrives, a timer indicates the current process's time slice is over, or an event needing immediate attention occurs.

In preemptive scheduling, processes may be **suspended or preempted** even if they haven't voluntarily given up the CPU. The operating system has more control over process execution, which allows it to allocate CPU resources dynamically.

This type of scheduling is ideal for situations where **responsiveness and fairness** are crucial, such as in real-time systems and multitasking environments.



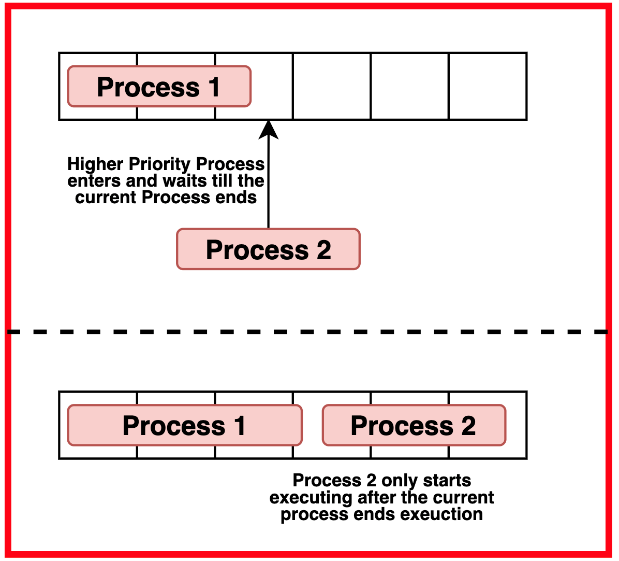


**Real-world Examples**

* Preemptive scheduling is crucial for modern multitasking operating systems like Windows, macOS, and Linux. These systems run multiple processes at the same time and ensure each process gets fair CPU time, allowing high-priority tasks to be executed promptly.
* In **real-time systems**, such as automotive systems, preemptive scheduling ensures timely responses to sensor inputs or critical actions like applying brakes, which requires quick reactions.
* For **network servers**, preemptive scheduling is beneficial for handling multiple client requests simultaneously. It ensures that no single client request dominates CPU resources, keeping the system responsive for all clients.

**Non-Preemptive Scheduling**

In **Non-Preemptive Scheduling**, a running process keeps the CPU until it voluntarily releases it by completing its task, waiting for an I/O operation, or entering a waiting state. These processes have **lower overhead** compared to preemptive scheduling, but it might lead to **slower responsiveness** if long-running processes dominate CPU time.





**Real-world Examples**

* Non-preemptive scheduling is used in **resource-constrained embedded systems** where timing requirements are strict. It simplifies timing analysis and ensures predictable execution, which is ideal for systems like industrial automation or IoT devices.
* It is also suitable for **batch processing systems** where tasks run sequentially without interruption. For example, in payroll systems, each calculation task can complete uninterrupted, ensuring data consistency.
* Non-preemptive scheduling can be used in **single-user systems** where users expect consistent behavior rather than responsiveness, such as in personal computers running desktop applications.

**Differences between Pre-emptive and Non-Preemptive Scheduling**

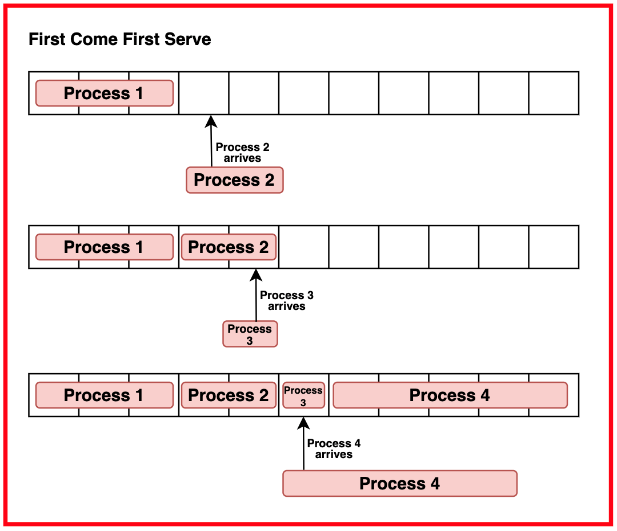
|  |  |  |
| --- | --- | --- |
| **Aspect** | **Preemptive** | **Non-Preemptive** |
| **Interruption** | Processes can be interrupted while executing | Processes cannot be interrupted while executing |
| **Control of CPU** | Operating system controls CPU allocation | Processes retain control of the CPU until completion |
| **Response Time** | Typically shorter response times | Response times may be longer |
| **Fairness** | Allows for fairness by using priority-based scheduling | May face fairness issues if long-running processes dominate the CPU |
| **Complexity** | More complex due to dynamic process switching | Simpler to implement and manage |
| **Resource Utilisation** | Generally more efficient CPU utilisation | May be less efficient if processes monopolise CPU |
| **Suitable Environments** | Ideal for multitasking and real-time systems | Suitable for simpler systems or those needing predictable behavior |

1. **Scheduling algorithms(FCFS, SJF, SRTF, HRNN, RR, PBS, MLQ, MLFQ)**



**First-Come, First-Served (FCFS)**

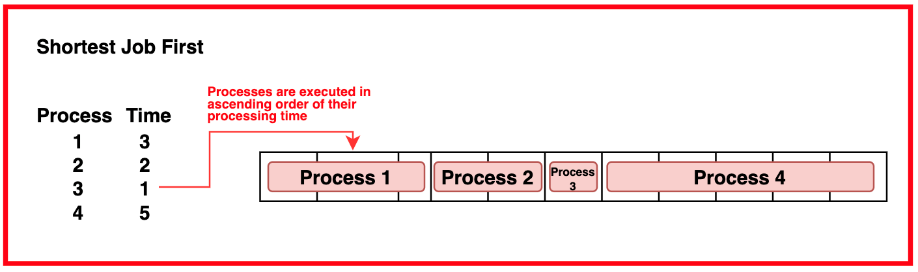
In **FCFS scheduling**, processes are executed in the order they arrive in the ready queue. The CPU is allocated to the first process in the queue, and it continues execution until it completes or enters a waiting state.



* It is simple and easy to implement.
* Fairness issues can arise if long-running processes take over CPU time.
* Can lead to poor average waiting times, especially for long processes that arrive early.

**Shortest Job First (SJF)**

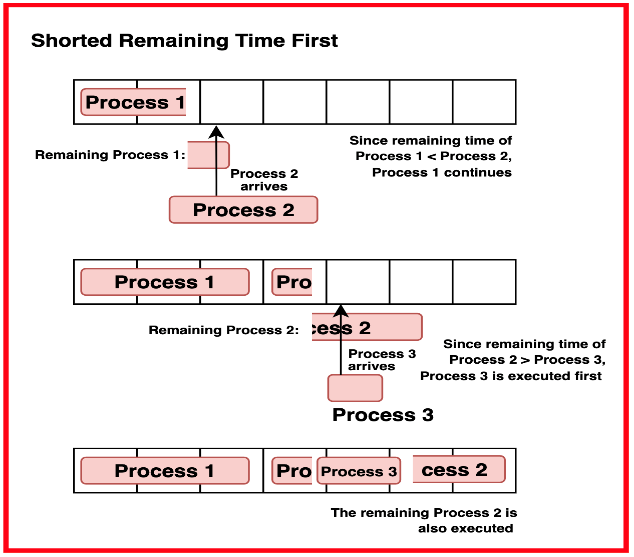
**SJF scheduling** selects the process with the shortest estimated CPU burst time next. This algorithm minimises the average waiting time by prioritising short processes.



* Requires knowledge of the CPU burst times of processes (either estimated or actual).
* Optimal for minimising average waiting time but can lead to starvation for long processes if short processes frequently arrive.

**Shortest Remaining Time First (SRTF)**

**SRTF** is a preemptive version of SJF where the currently executing process can be preempted by a shorter job. Whenever a new process arrives, the scheduler compares its burst time with the remaining time of the currently executing process and switches if the new process has a shorter remaining time.

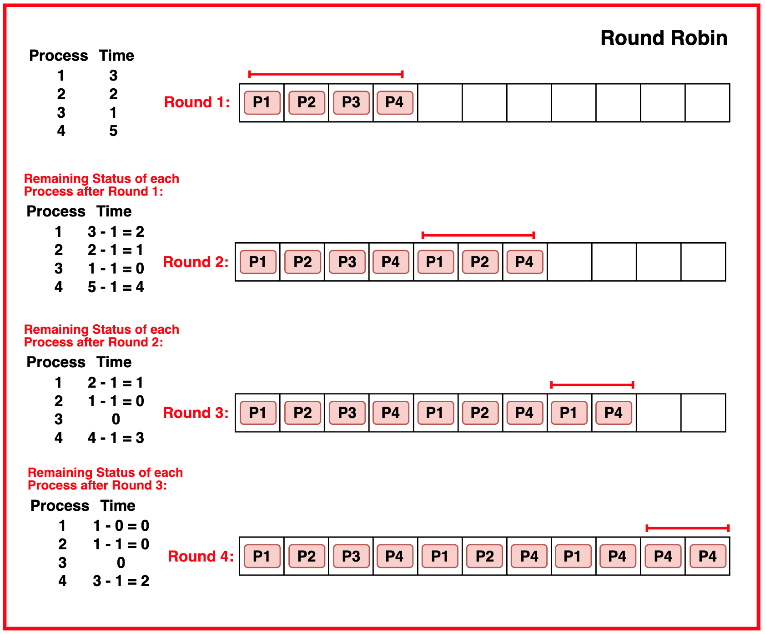




* SRTF provides optimal average waiting time.
* May cause frequent context switches, leading to increased overhead(extra resources).

**Round-Robin (RR)**

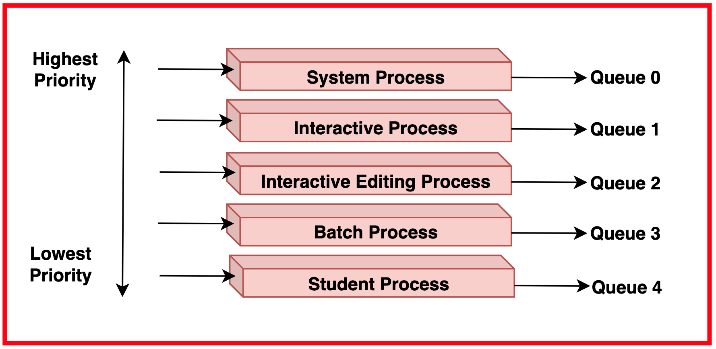
**RR scheduling** allocates CPU time to each process in turn, with a fixed time quantum (also known as time slice). When a time quantum expires, the currently executing process is preempted, and the CPU is allocated to the next process in the queue.



* Prevents starvation by ensuring that every process gets a fair share of CPU time.
* Fairly simple to implement.
* Performance depends on the choice of time quantum; shorter quantum improves response time but increases overhead.

**Multilevel Queue**

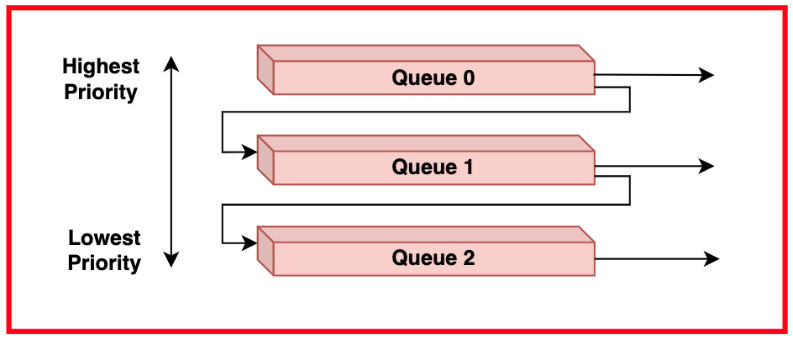
**Multilevel queue scheduling** divides the ready queue into multiple queues, each with its own scheduling algorithm and priority level. Processes are assigned to different queues based on criteria such as process type, priority, or characteristics.



* Allows for the classification of processes into different categories and the application of different scheduling policies.
* Processes may migrate between queues based on their behaviour or priority level.

**Multilevel Queue with Feedback**

**Multilevel queue with feedback scheduling** is an extension of multilevel queue scheduling where processes can move between queues based on their CPU burst time or execution behaviour. If a process uses up its time quantum without completing, it is moved to a lower-priority queue to prevent starvation.



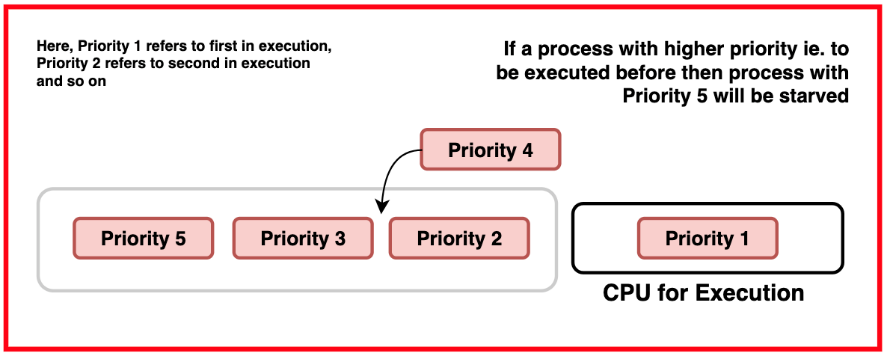
* Provides dynamic priority adjustment based on process behaviour.
* Prevents starvation of short processes in higher-priority queues.
* More complex to implement compared to a simple multilevel queue.

1. **Starving and Aging**

**Starvation in Process Scheduling**

**Starvation** in process scheduling happens when a process is repeatedly denied the resources it needs because other processes are prioritized. This often occurs in priority-based scheduling systems, where lower-priority processes may be delayed indefinitely if higher-priority ones keep arriving.

Starvation can be a serious issue because it may lead to **inefficient system performance**, causing some processes to never execute, which can make the system unstable and leave certain tasks unfinished.

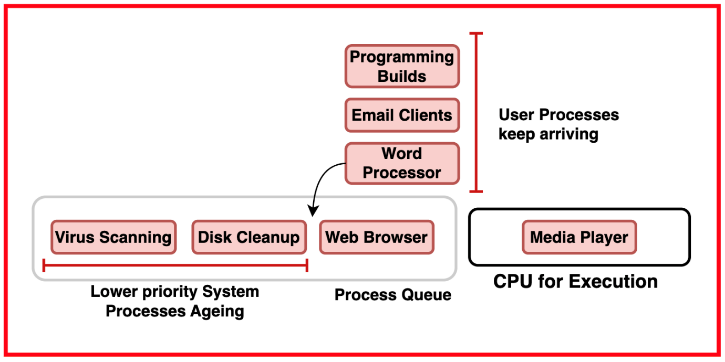




**Ageing as a Solution to the Starvation Problem**

**Ageing** is a method to prevent starvation by gradually increasing a process's priority as it waits in the queue. This ensures that all processes eventually become high-priority if they wait long enough, allowing them to get their turn for execution.

Ageing helps maintain balance in the system, letting **low-priority processes** move up in priority over time. This prevents long waits and ensures a fairer distribution of resources for all processes.





**Starvation and Ageing in Different Scheduling Algorithms**

* **First-Come, First-Served (FCFS)**: Generally, FCFS avoids starvation because processes are handled in the order they arrive. But it can have a **convoy effect**, where short processes are delayed behind long ones.
* **Shortest Job Next (SJN)**: This can lead to starvation for longer processes, as shorter ones are always prioritized. **Ageing** can be used to adjust the priority of longer processes over time.
* **Priority Scheduling**: Prone to starvation for lower-priority processes. Ageing is often applied here to gradually increase the priority of waiting processes.
* **Round Robin (RR)**: RR usually prevents starvation since every process gets a fixed time slice in a cycle. Ageing is less needed because all processes get CPU time.

**Real-world Examples**

In a multi-user operating system, imagine a scenario where a **background system maintenance process** has lower priority than user applications. If user applications keep arriving, the maintenance process might starve and never get CPU time.

By using ageing, the maintenance process’s priority would gradually increase, ensuring it eventually gets CPU time even with many user applications. Another example is in **network packet scheduling**, where low-priority packets might starve if high-priority ones keep coming. Ageing helps ensure all packets are eventually transmitted.

**Preventing Starvation in Scheduling**

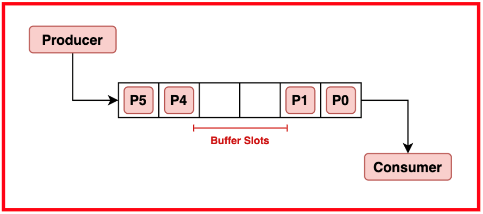
To prevent starvation, one can implement **ageing** in priority-based systems, so all processes gain priority over time and eventually get executed. Another approach is to use **Round Robin scheduling**, which naturally avoids starvation by giving each process a turn.

Additionally, setting a **maximum wait time** or using **feedback queues** can help. In feedback queues, processes move between different priority levels based on their waiting time or service needs. These methods help ensure a balanced and fair distribution of resources across all processes.

1. **Producer-consumer problem**

The **producer-consumer synchronisation** is used to manage the communication between two software processors: producers and consumers.

* **Producers** are responsible for generating data or items that need to be consumed by other parts of the system. Producers add items to a shared buffer or queue.
* **Consumers** are responsible for consuming or processing the items generated by the producers. Consumers retrieve items from the shared buffer or queue and process them.



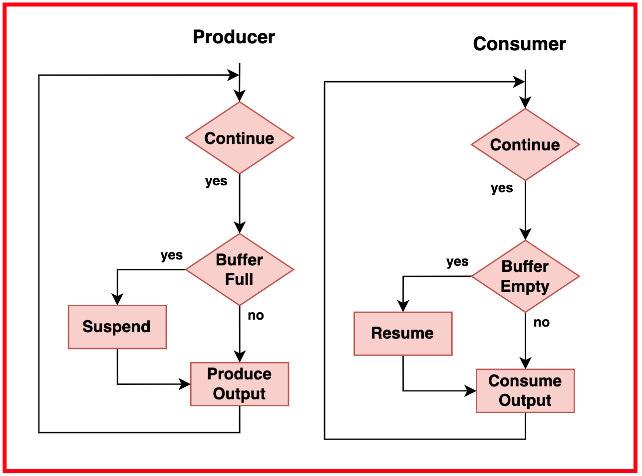
**Shared Buffer/Queue**

There is a **Shared Buffer/Queue** that serves as the communication medium between producers and consumers. Producers add items to this buffer, and consumers remove items from it.

Both producers and consumers may run concurrently, meaning they can execute simultaneously. However, to prevent issues like race conditions or data corruption, proper synchronisation mechanisms must be used.

**Producers** generate items and add them to the shared buffer. If the buffer is full, they may need to wait until space becomes available. **Consumers** retrieve and process items from the buffer, and may need to wait if it is empty.

Common synchronisation mechanisms include **locks, semaphores, condition variables**, or specialised data structures like **blocking queues**.



**Challenges in Producer-Consumer Processing**

* **Data Synchronisation**: Ensures that producers and consumers operate in a coordinated way, avoiding issues like race conditions, data corruption, or buffer overflow/underflow. It also ensures that the order of produced and consumed items is maintained.
* **Mutual Exclusion**: Enforces that only one producer or consumer can access the shared buffer at a time, preventing conflicts and maintaining data integrity. It is also important to avoid deadlocks, where producers or consumers wait indefinitely for resources held by each other.
* **Resource Utilisation**: It is crucial to manage the buffer size to avoid **buffer overflow** (when full) or **buffer underflow** (when empty). The rates of production and consumption must be balanced to prevent inefficiency or system instability.

**Solutions to the Producer-Consumer Problem**

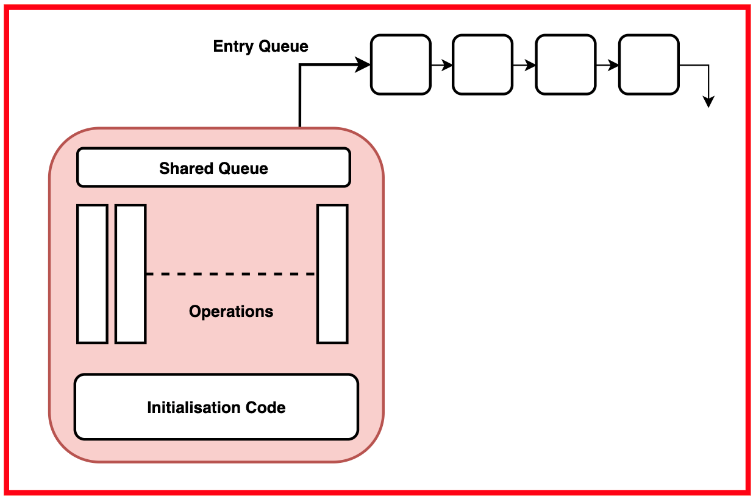
**Semaphores**

A **semaphore** is a synchronisation primitive used in concurrent programming to control access to a shared resource by multiple processes or threads.

Two semaphores can be used to represent the empty and full slots in the buffer. Producers decrement the empty semaphore and increment the full semaphore when adding items. Consumers do the reverse.

**Using Monitors**

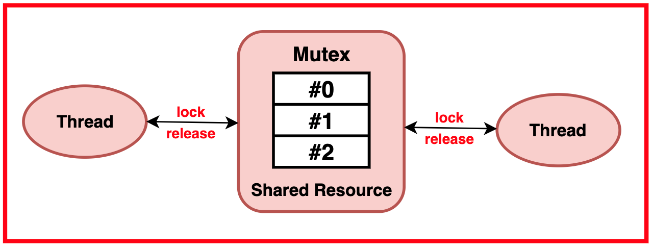
**Monitors** encapsulate shared data and procedures. Condition variables are used to block and wake up threads waiting for certain conditions. Producers wait on a condition variable when the buffer is full, and consumers wait when the buffer is empty. Signal operations notify waiting threads when the buffer state changes.



**Mutex and Condition Variables**

A **mutex** (lock) ensures mutual exclusion when accessing the shared buffer. Producers and consumers acquire and release this mutex to prevent concurrent access.

**Condition variables** can block and unblock producers and consumers based on the buffer's state. Producers wait if the buffer is full, and consumers wait if the buffer is empty.



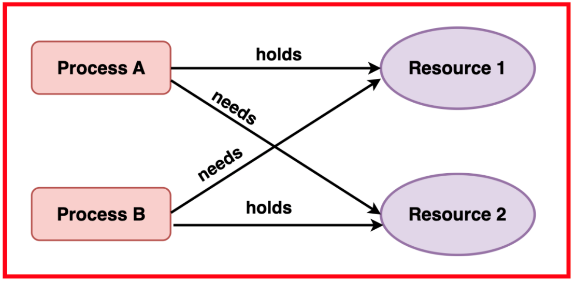
**Blocking Queues**

Using a **blocking queue** simplifies the producer-consumer problem as it automatically handles synchronisation. Producers can push items into the queue, and consumers can pop items from it. If the queue is full, producers block until space is available; if the queue is empty, consumers block until items are available.

1. **Critical Section Problem**

In concurrent programming, a **critical section** is a portion of code that accesses shared resources (such as variables, data structures, or files) which must be accessed by multiple threads or processes. It is critical because if multiple threads or processes execute this section concurrently, it could lead to unexpected behaviour, data corruption, or incorrect results due to race conditions.

**Understanding the Critical Section Problem**



The critical section problem arises when multiple processes or threads need to access and modify shared resources simultaneously. The main challenge is to ensure that only one process or thread enters the critical section at a time to prevent conflicts.

If not managed properly, this can lead to **race conditions**, where the system's behaviour depends on the sequence or timing of uncontrollable events, causing unpredictable and erroneous results known as data inconsistency.

For instance, if two threads simultaneously update a shared variable without proper synchronisation, the final value of the variable might be incorrect, leading to potential system failures or incorrect computations.

**Race Conditions**

A **race condition** occurs when the outcome of a program depends on the sequence or timing of uncontrollable events, such as the order in which threads are scheduled to run. This can lead to unpredictable and erroneous behaviour in concurrent systems.

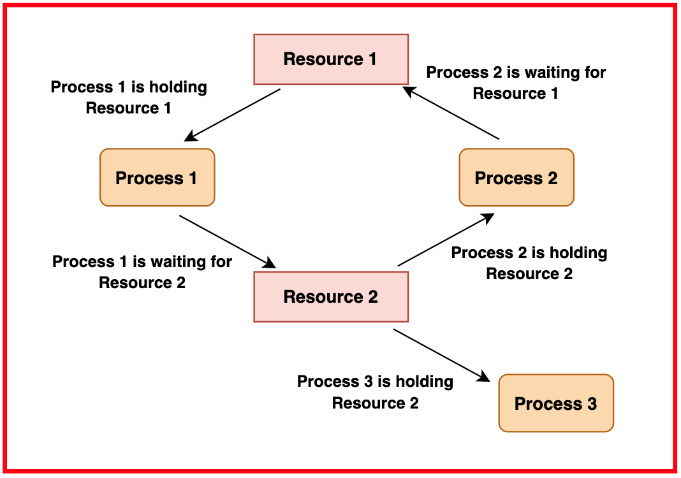
Consider two threads attempting to increment a shared counter variable. If both read the

counter's value simultaneously and then increment it, the final value will reflect only one increment instead of two, leading to incorrect results.

Race conditions often result in data inconsistency. For example, in a banking application, if two transactions are processed concurrently without proper synchronisation, the final account balance may be incorrect.

To prevent race conditions, synchronisation mechanisms like locks, mutexes, or other atomic operations are necessary. These ensure that only one thread accesses the critical section at a time.

**Deadlock**





A **deadlock** is a situation where a set of processes are blocked because each process is holding a resource and waiting for another resource held by another process. The conditions for a deadlock to occur include:

* **Mutual Exclusion:** At least one resource must be held in a non-shareable mode.
* **Hold and Wait:** A process must be holding at least one resource while waiting to acquire additional resources.
* **No Preemption:** Resources cannot be forcibly taken from processes until they voluntarily release them.
* **Circular Wait:** A set of processes exists where each process waits for a resource held by the next process in the sequence.

To prevent deadlock, multiple strategies can be employed, such as algorithms like the Banker's algorithm, which dynamically examine resource allocation states to ensure that circular wait conditions do not occur. Techniques to detect deadlock and methods to recover from it include resource preemption and process termination.

**Starvation**

**Starvation** is a situation where a process is perpetually denied necessary resources to proceed with its execution. It is often caused by priority scheduling, where low-priority processes never get the resources they need if high-priority processes keep preempting them.

To prevent starvation, techniques such as **aging** can be used, where the priority of a process increases the longer it waits.

1. **Process synchronization and its tools**

Process Synchronisation refers to the coordination of processes to ensure that they can operate concurrently without interfering with each other, particularly when they need to access shared resources. Synchronisation is critical to maintain consistency and correctness of data, prevent race conditions, and avoid other issues such as deadlocks and starvation.

**Process Synchronisation in Concurrent Systems**

**Maintaining Data Consistency**

When multiple processes or threads access and modify shared resources (like variables, files, or databases), there is a risk of data inconsistency. Synchronisation mechanisms ensure that only one process accesses the critical section of code that modifies shared resources at any given time. In a banking system, if two processes simultaneously update an account balance without synchronisation, the final balance might reflect only one of the updates, leading to incorrect results.

**Preventing Race Conditions**

Race Conditions occur when the system's behaviour depends on the sequence or timing of uncontrollable events, leading to unpredictable results. Proper synchronisation ensures that the operations in the critical section are executed in a controlled manner. Two threads incrementing a shared counter without synchronisation might both read the same initial value and write back the same incremented value, resulting in an incorrect final count.

**Avoiding Deadlocks**

Deadlocks happen when a set of processes are blocked because each process is waiting for a resource held by another process. Synchronisation techniques, such as lock hierarchies and timeout mechanisms, help prevent deadlocks by ensuring proper resource allocation and release. If two processes each hold a lock and try to acquire the lock held by the other, they will be stuck waiting indefinitely unless proper deadlock avoidance strategies are in place.

**Fair Resource Allocation**

Starvation occurs when a process is perpetually denied access to the resources it needs for execution. Synchronisation mechanisms can include fairness policies to ensure that all processes get a chance to execute.

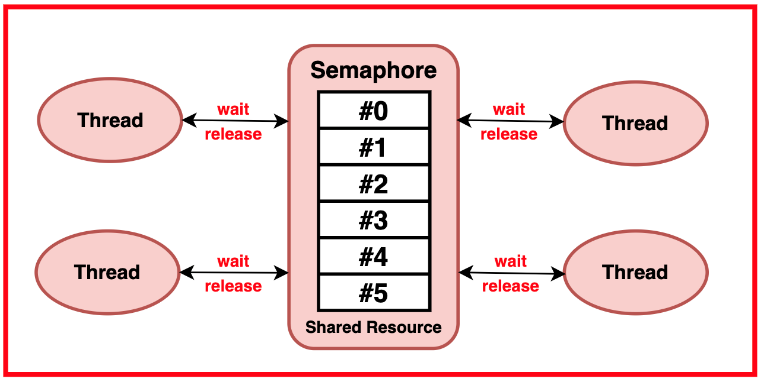
**Synchronisation Mechanisms**

**Locks and Mutexes**

Provide mutual exclusion, ensuring that only one thread or process can enter the critical section at a time.

**Semaphores**

Use signalling mechanisms to control access to resources, allowing multiple threads to use resources concurrently while avoiding conflicts.



**Monitors**

Combine mutual exclusion locks with condition variables to manage access to shared resources in a structured way.

**Atomic Operations**

Provide a way to perform operations atomically

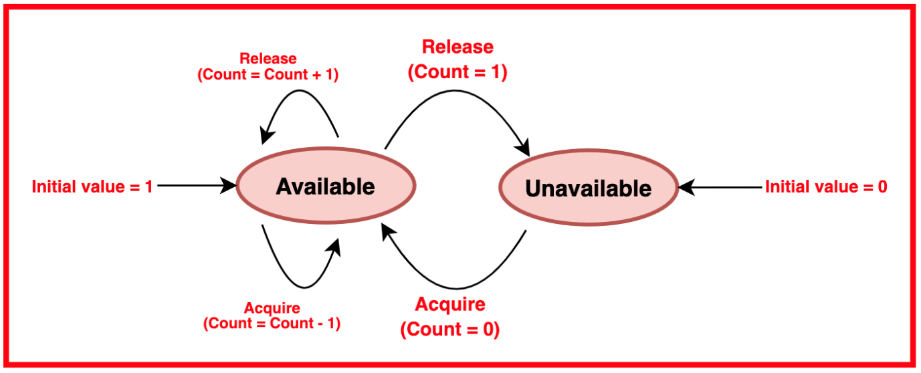
without the need for explicit locks, reducing overhead.

**Condition Variables**

Condition variables are used in conjunction with mutexes to allow threads to wait for certain conditions to become true. A thread can wait on a condition variable and will be awakened when another thread signals that the condition has changed. This mechanism is useful for scenarios where a thread needs to wait until a specific resource is available or a particular state is reached.

**Read-Write Locks**

Read-write locks allow concurrent read access for multiple threads while providing exclusive write access for a single thread. This mechanism improves performance in scenarios where read operations are more frequent than write operations, as multiple threads can read the data simultaneously without waiting for write locks.

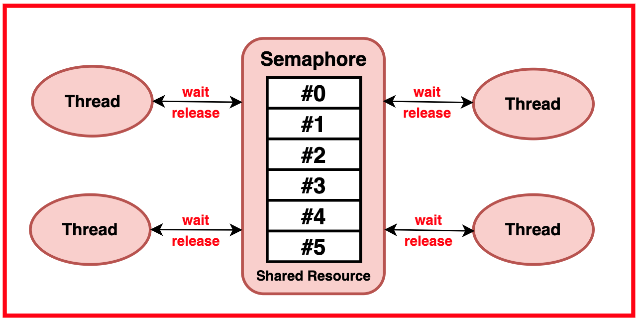


**Barrier Synchronisation**

Barrier synchronisation allows multiple threads to wait until all threads reach a certain point in execution before proceeding. This is useful in parallel programming, where it is necessary to ensure that all threads have completed their current task before moving on to the next stage, thus coordinating their progress in a structured manner.

1. **Semaphores and its type**

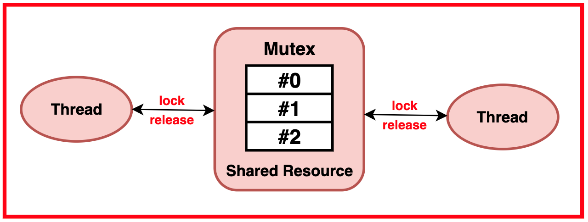
**Semaphores** are synchronisation primitives used to manage concurrent processes by controlling access to shared resources in a computing environment.



A semaphore is essentially a variable that signals whether a particular resource is available. Using operations like **wait** (or P) and **signal** (or V), semaphores coordinate the activities of different processes, ensuring that only a specified number of processes can access the critical section at any given time. This coordination is crucial in multi-threaded applications to maintain data consistency and integrity.

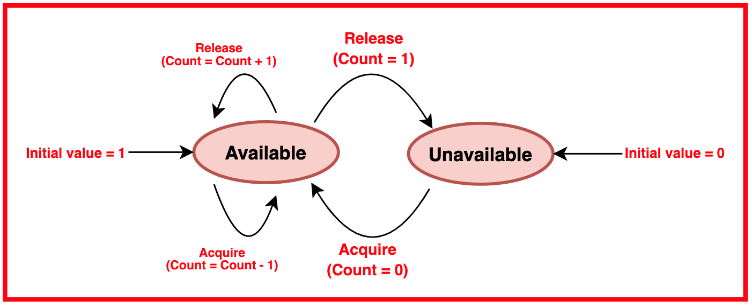
**Binary Semaphores**

Binary semaphores, also known as **mutexes** (mutual exclusion), have only two states: 0 and 1. They ensure that a resource is accessed by only one thread at a time. When a thread acquires the semaphore, it sets the value to 0, and when it releases the semaphore, it sets the value back to 1. Binary semaphores are typically used for simple lock/unlock mechanisms where mutual exclusion is required.



**Counting Semaphores**

Counting semaphores can take on any non-negative integer value and are used to control access to a resource with a limited number of instances. The semaphore's value is initialised to the number of available resources. Each **wait** operation decrements the value, and each **signal** operation increments it. If the value is zero, indicating no resources are available, the process attempting to decrement the semaphore is blocked until another process increments it. This type of semaphore is useful in scenarios where a pool of identical resources needs to be managed, such as a fixed number of database connections.



**Semaphores Vs Other Synchronisation Primitives**

Semaphores are often compared with other synchronisation primitives like **locks, monitors,** and **condition variables**:

* Unlike locks, which are used for mutual exclusion, semaphores offer more versatility with their counting capability, allowing multiple instances of a resource to be managed.
* Monitors encapsulate both mutual exclusion and the condition variables needed for complex waiting scenarios, offering a higher level of abstraction compared to semaphores.
* Condition variables, used with locks, allow threads to wait for certain conditions to be met, sometimes making them easier to use than semaphores.

However, semaphores provide a simpler and more direct way to implement certain synchronisation scenarios, especially those involving limited resource pools.

**Deadlock Avoidance**

To avoid deadlocks with semaphores, techniques such as resource ordering, timeout mechanisms, and deadlock detection algorithms can be employed. Ensuring that all semaphores are requested in a predefined global order can prevent circular wait conditions, a key factor in deadlock.

**Priority Inversion**

**Priority inversion** happens when a higher-priority task is waiting for a resource held by a lower-priority task, causing the higher-priority task to be indirectly preempted. This can be mitigated using **priority inheritance protocols**, where a lower-priority task temporarily inherits the higher priority of the blocked task, ensuring that critical sections are exited more quickly.

1. **Deadlock and Methods for preventing deadlock**

Deadlock occurs in concurrent systems when a set of processes are each waiting for an event that only another process in the set can cause. As a result, none of the processes can proceed, and the system is essentially stuck.

**Key Characteristics of Deadlock**

Key characteristics of deadlock include:

* Processes being in a state of indefinite wait.
* No single process being able to progress due to unavailable resources held by other waiting processes.

Deadlock is often represented by a resource allocation graph where a cycle indicates its presence. Understanding these characteristics is essential for diagnosing and addressing deadlock in multi-threaded or multi-process environments.

**Detecting and Handling Deadlocks**

Various methods exist for detecting and handling deadlock, each with its strengths and trade-offs:

* **Deadlock Prevention:** This method designs systems so that one of the necessary conditions for deadlock cannot occur.
* **Deadlock Avoidance:** This method dynamically examines resource allocation to ensure the system never enters a deadlock state, using algorithms like the Banker's algorithm.
* **Deadlock Detection:** This allows the system to enter a deadlock state but periodically checks for deadlock conditions using techniques like resource allocation graphs, then recovers from it through process termination or resource preemption.
* **Deadlock Recovery:** This involves strategies to break the deadlock once detected, such as aborting one or more processes to reclaim resources.

**Deadlock Prevention Techniques**

**One-Shot Allocation (All-or-Nothing Allocation)**

One-Shot Allocation requires a process to request and be allocated all the resources it will need at once. If any resources are not available, none are allocated, and the process waits until all can be provided together. This prevents the hold and wait condition.

Advantages:

* Simplifies resource allocation logic.
* Ensures that a process will not enter a wait state after starting.

Disadvantages:

* Leads to low resource utilization and possible starvation.
* Difficult to predict and request all needed resources in advance.

**Preemptive Resource Allocation**

Preemptive Resource Allocation allows the system to forcibly take back resources from processes to avoid deadlock. If a process requests more resources while holding some, the system may temporarily preempt these resources to allocate them to other waiting processes.

Advantages:

* Reduces the likelihood of deadlock by breaking hold-and-wait conditions.
* Improves resource utilization by reallocating resources dynamically.

Disadvantages:

* Complex to implement and manage, requiring mechanisms to save and restore process states.
* May lead to overhead and performance issues due to frequent preemptions.

**Dining Philosophers Problem**

The Dining Philosophers Problem is a classic synchronization problem that illustrates the challenges of allocating limited resources among competing processes without causing deadlock. The problem is set up as follows:

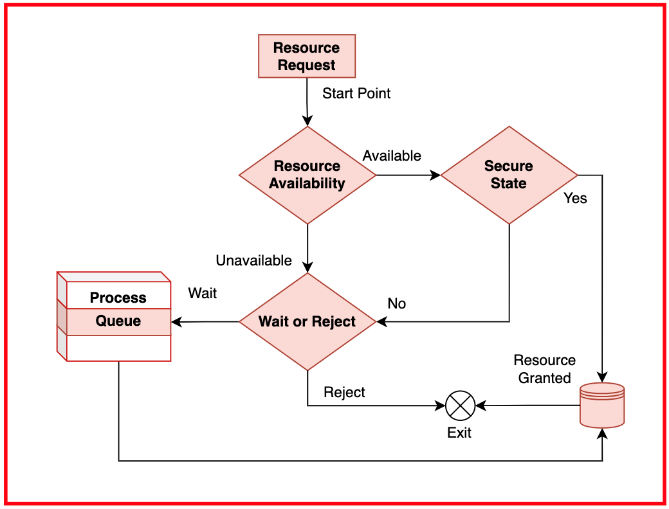
* Five philosophers sit around a circular table, alternating between thinking and eating.
* Each philosopher has a plate of pasta, and there is one fork between each pair of adjacent philosophers.
* To eat, a philosopher needs both forks, and can only pick up one at a time.

The deadlock scenario arises when each philosopher simultaneously picks up the fork to their left

1. **Banker's algorithm**

The **Banker's Algorithm**, developed by Edsger W. Dijkstra, is a deadlock avoidance algorithm that ensures a safe sequence of resource allocation, preventing system deadlocks. Similar to a banking system where a banker allocates funds to clients, it operates under the assumption that there are multiple resources available and each process has a maximum claim for each resource.

The algorithm evaluates whether it is safe to grant a resource request by simulating the allocation and checking if the system can still fulfill the maximum resource needs of all processes without leading to a deadlock. If a safe sequence of resource allocation exists, the request is granted; otherwise, the process must wait.

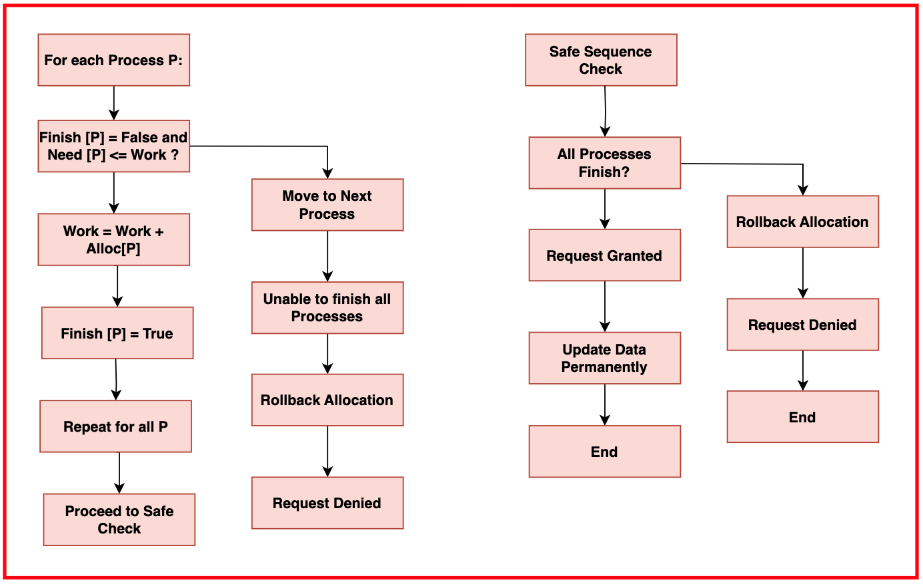




**The Algorithm**

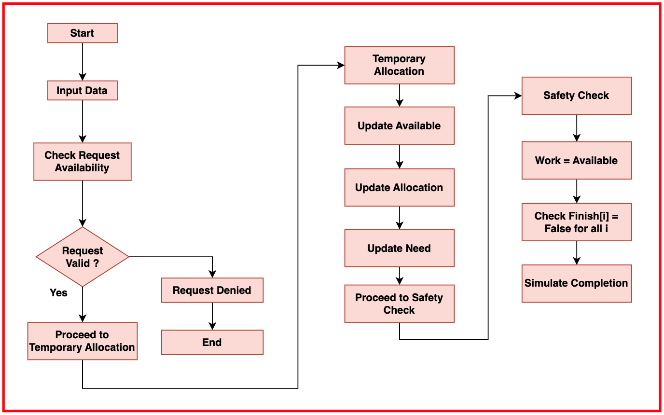
The core principle of the Banker's Algorithm is to avoid unsafe states where a deadlock would occur. It works by maintaining data on available resources, maximum demands, currently allocated resources, and the remaining needs of each process.

When a process requests resources, the algorithm checks if fulfilling the request keeps the system in a safe state. This is done by temporarily allocating the resources and then determining if the remaining resources are sufficient for the maximum needs of the other processes. If the system remains in a safe state, the allocation is confirmed; otherwise, the request is denied and the process must wait.



**How Resources Are Allocated**

* **Initialisation**: Set up matrices and vectors for available resources, maximum demands, allocated resources, and remaining needs.
* **Request Handling**: When a process requests resources, check if the request is less than its remaining need and the available resources.
* **Safety Check**: Temporarily allocate the requested resources to the process and update the available resources.
* **Determine Safe State**: Simulate the allocation to check if the remaining resources can satisfy the needs of all other processes. This involves finding at least one sequence of process completion that leads to all processes eventually receiving their maximum required resources.
* **Grant or Deny**: If a safe sequence exists, confirm the allocation. If not, revert to the previous state and make the process wait.



**Example**

Consider a system with three types of resources (A, B, C) and three processes (P1, P2, P3). Suppose the available resources are [3, 3, 2], and the maximum demands and current allocations are:

* **P1**: Max [7, 5, 3], Allocated [0, 1, 0]
* **P2**: Max [3, 2, 2], Allocated [2, 0, 0]
* **P3**: Max [9, 0, 2], Allocated [3, 0, 2]

When P1 requests [1, 0, 2], the algorithm checks if this request can be granted:

1. **Check request validity**:

* [1, 0, 2] ≤ [7, 4, 3] (remaining need for P1), and [1, 0, 2] ≤ [3, 3, 2] (available).
* Each process can only request resources up to its **maximum demand** minus what is **already allocated**.
* The request [1, 0, 2] is checked against P1's remaining needs:  
  [ [7, 5, 3] - [0, 1, 0] = [7, 4, 3] ] Since ([1, 0, 2] ≤ [7, 4, 3]), this is valid.

1. **Safety check**:

Temporarily allocate the resources to P1 and update available to [2, 3, 0].

1. **Simulate completion**:

Check if the system can fulfill the maximum needs of P2 and P3 with the remaining resources. If a safe sequence is found, the request is granted