Unit 2
PROCESSES AND THREADS

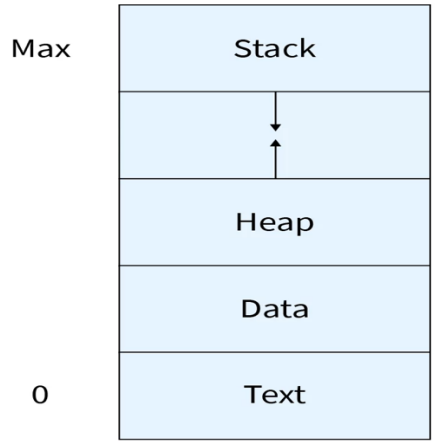
2.1. Introduction to Process:

- In an operating system, a process is an instance of a program in execution.
- It is a basic unit of work that can be scheduled and executed by the operating system.
- A process is a running program that serves as the foundation for all computation.

Components of a Process in OS

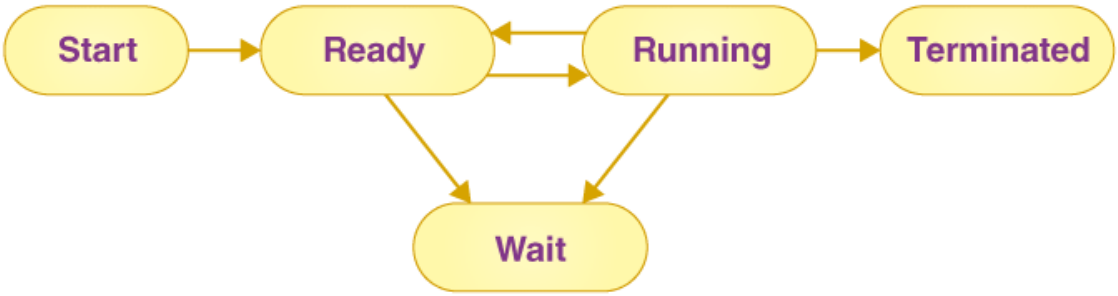
The components of a process are:

- **Program code/Text:** The instructions that the process will execute.
- **Data:** The data that the process will use during its execution.
- **Stack:** A data structure that is used to store temporary data, such as function parameters and return addresses.
- **Heap:** A data structure that is used to store dynamically allocated memory.
- **Process control block (PCB):** A data structure that contains information about the process, such as its state, priority, and memory usage.



Process Life Cycle

When a process runs, it goes through many states. Distinct operating systems have different stages, and the names of these states are not standardised. In general, a process can be in one of the five states listed below at any given time.



- **Start**

When a process is started/created first, it is in this state.

- **Ready**

Here, the process is waiting for a processor to be assigned to it. Ready processes are waiting for the operating system to assign them a processor so that they can run. The process may enter this state after starting or while running, but the scheduler may interrupt it to assign the CPU to another process.

- **Running**

When the OS scheduler assigns a processor to a process, the process state gets set to running, and the processor executes the process instructions.

- **Waiting**

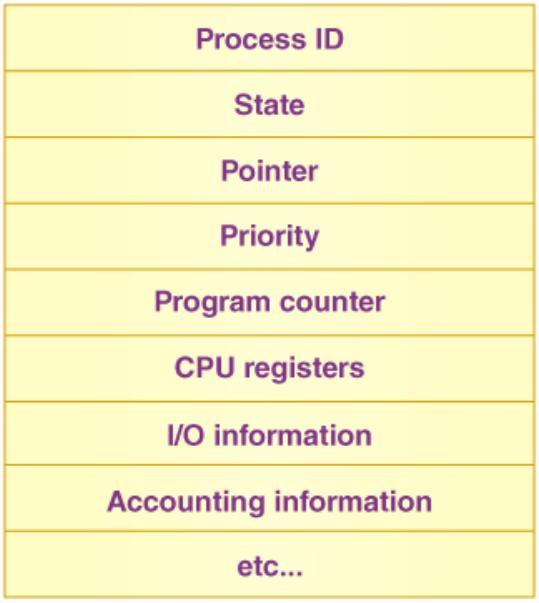
If a process needs to wait for any resource, such as for user input or for a file to become available, it enters the waiting state.

- **Terminated or Exit**

The process is relocated to the terminated state, where it waits for removal from the main memory once it has completed its execution or been terminated by the operating system.

Process Control Block (PCB):

Every process has a process control block, which is a data structure managed by the operating system. An integer process ID (or PID) is used to identify the PCB. As shown below, PCB stores all of the information required to maintain track of a process.



- **Process state**

The process's present state, such as whether it's ready, waiting, running, or whatever.

- **Process privileges**

This is required in order to grant or deny access to system resources.

- **Process ID**

Each process in the OS has its own unique identifier.

- **Pointer**

It refers to a pointer that points to the parent process.

- **Program counter**

The program counter refers to a pointer that points to the address of the process's next instruction.

- **CPU registers**

Processes must be stored in various CPU registers for execution in the running state.

- **CPU scheduling information**

Process priority and additional scheduling information are required for the process to be scheduled.

- **Memory management information**

This includes information from the page table, memory limitations, and segment table, all of which are dependent on the amount of memory used by the OS.

- **Accounting information**

This comprises CPU use for process execution, time constraints, and execution ID, among other things.

- **IO status information**

This section includes a list of the process's I/O devices.

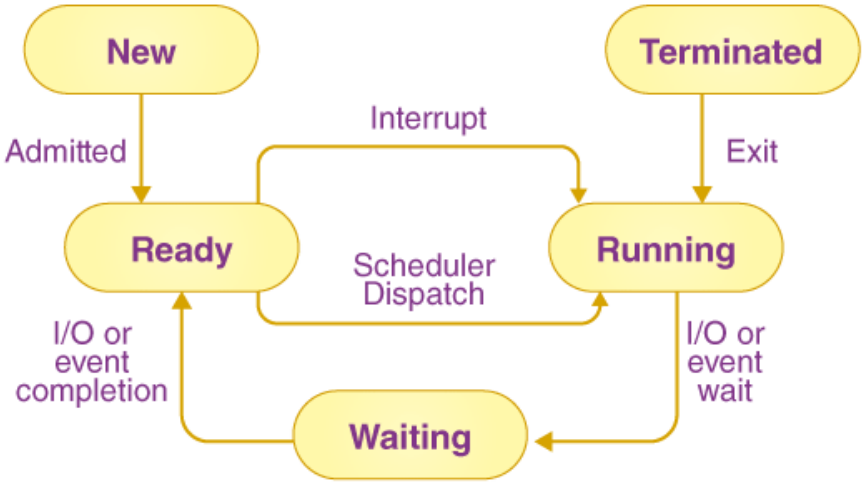
The PCB architecture is fully dependent on the operating system, and different operating systems may include different information. A simplified diagram of a PCB is shown below.

The PCB is kept for the duration of a procedure and then removed once the process is finished.

The Different Process States:

The operating system's processes can be in one of the following states:

- **NEW** – The creation of the process.
- **READY** – The waiting for the process that is to be assigned to any processor.
- **RUNNING** – Execution of the instructions.
- **WAITING –** The waiting of the process for some event that is about to occur (like an I/O completion, a signal reception, etc.).
- **TERMINATED** – A process has completed execution.



2.2. Process Creation and Termination:

✓ **Process Creation**

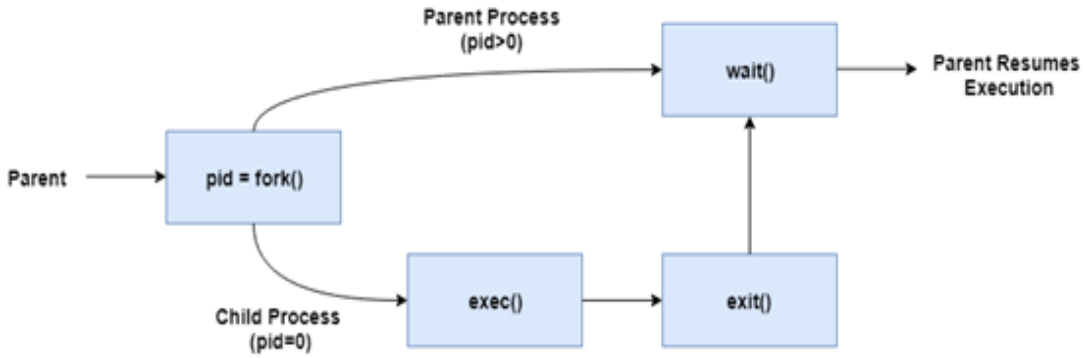
- A process may be created in the system for different operations.

Some of the events that lead to process creation are as follows −

- User request for process creation
- System Initialization
- Batch job initialization
- Execution of a process creation system call by a running process

- A process may be created by another process using fork().
- The creating process is called the parent process and the created process is the child process.
- A child process can have only one parent but a parent process may have many children.
- Both the parent and child processes have the same memory image, open files and environment strings.
- However, they have distinct address spaces.

A diagram that demonstrates process creation using fork() is as follows −



Process Creation Using fork()

✓ **Process Termination**

✓ Processes are terminated by themselves when they finish executing their last statement, then operating system USES exit( ) system call to delete its context.

✓ Then all the resources held by that process like physical and virtual memory, 10 buffers, open files, etc., are taken back by the operating system.

✓ A process P can be terminated either by the operating system or by the parent process of P.

A parent may terminate a process due to one of the following reasons:

1. When task given to the child is not required now.
2. When the child has taken more resources than its limit.
3. The parent of the process is exiting, as a result, all its children are deleted. This is called cascaded termination.

2.3. Process Hierarchy:

- ✓ In operating systems, a **process hierarchy** refers to the structured relationship of processes, resembling a family tree where processes are created, organized, and managed in parent-child relationships.
- ✓ This hierarchy is fundamental to how operating systems manage and control processes, allocate resources, and handle communication.

- **Parent Process:** A process that creates a new process.
- **Child Process:** A process that is created by a parent process.

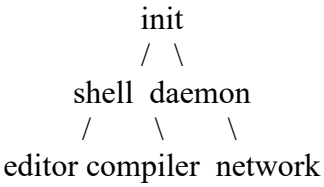
Process Hierarchy Structure:

- ✓ The hierarchy starts with a single root process, often called the "init" process.
- ✓ This process is responsible for starting other processes and managing the system.
- ✓ As the system boots up, the init process creates child processes, which in turn can create their own child processes, forming a tree-like structure.

Example:

1. **Init Process:** The root of the hierarchy.
2. **Shell Process:** Created by the init process to interact with the user.
3. **Text Editor Process:** Spawned by the shell to edit a file.
4. **Compiler Process:** Launched by the shell to compile code.

Visual Representation:



Importance of Process Hierarchy:

- **Resource Allocation:** The parent process can control the resources allocated to its child processes.
- **Process Termination:** A parent process can terminate its child processes.
- **Signal Handling:** Signals can be sent to a process and its child processes.
- **Process Grouping:** Related processes can be grouped together for easier management.

Process Management Operations:

- **Process Creation:** A parent process creates a child process using a system call like fork() in Unix-like systems.
- **Process Termination:** A process can terminate itself or be terminated by its parent or the operating system.
- **Process Scheduling:** The operating system determines which process to execute next.
- **Process Switching:** The OS switches between processes to provide the illusion of concurrent execution.

2.4. Implementation of Process:

- ✓ In operating systems, the implementation of processes involves creating and managing units of execution that can run independently, each with its own resources and state.
- ✓ Processes are essential for multitasking and resource sharing within the OS. Here's a breakdown of how processes are implemented:

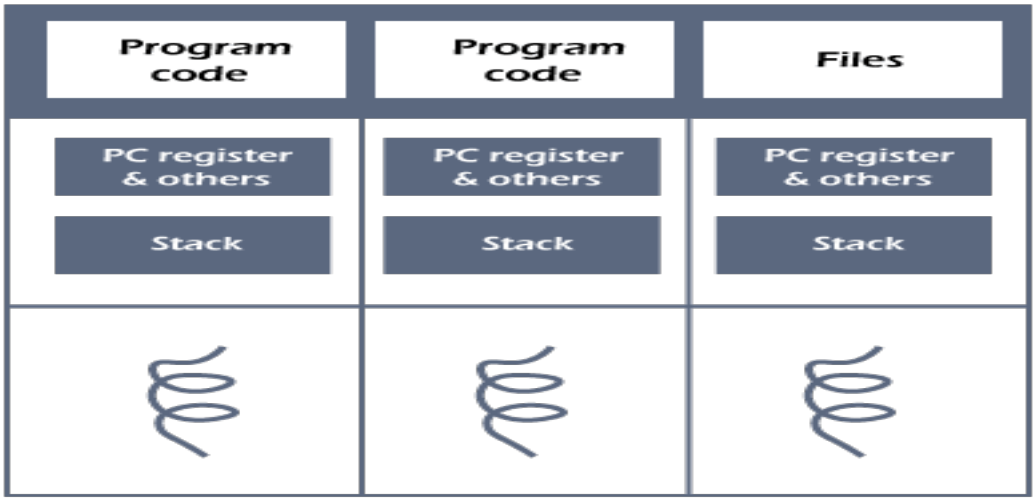
Implementation Steps:

1. **Process Creation:**
   - A system call is invoked to create a new process.
   - A new PCB is allocated for the process.
   - The process's initial state, program counter, and other relevant information are set.
   - The new process is added to the ready queue.
2. **Process Scheduling:**
   - The OS selects a process from the ready queue to execute.
   - The scheduler uses various algorithms (e.g., FIFO, round-robin, priority-based) to make this decision.
   - The selected process's PCB is loaded into the CPU's registers.
3. **Process Execution:**
   - The CPU executes the instructions of the selected process.
   - The process may transition to the waiting state if it needs to wait for an event.
4. **Process Switching (Context Switching):**
   - When a process is pre-empted or voluntarily yields the CPU, the OS saves the process's current state (registers, program counter) in its PCB.
   - The OS selects a new process from the ready queue.
   - The new process's state is loaded into the CPU's registers.
5. **Process Termination:**
   - A process may terminate normally (e.g., reaching the end of its program) or abnormally (e.g., due to an error).
   - The OS deallocates the process's memory and resources.
   - The process's PCB is removed from the system.

Program	Process
The program contains a set of instructions designed to complete a specific task.	The process is an instance of an executing program.
A program is a passive entity as it resides in the secondary memory.	The process is an active entity as it is created during execution and loaded into the main memory.
Program exists at a single place and continues to exist until it is deleted.	The process exists for a limited period as it gets terminated after the completion of the task.
A program is a static entity.	The process is a dynamic entity.
The program does not have any control block.	Process has its control block called Process Control Block.
Program contains instructions	The process is a sequence of instruction execution.

2.5. **Threads:**

- ✓ A thread is a single sequential flow of execution of tasks of a process so it is also known as thread of execution or thread of control.
- ✓ There is a way of thread execution inside the process of any operating system. Apart from this, there can be more than one thread inside a process.



Three threads of same process

- ✓ The process can be split down into so many threads.
- ✓ **For example**, in a browser, many tabs can be viewed as threads. MS Word uses many threads - formatting text from one thread, processing input from another thread, etc.

Need of Thread:

- It takes far less time to create a new thread in an existing process than to create a new process.
- Threads can share the common data; they do not need to use Inter- Process communication.
- Context switching is faster when working with threads.
- It takes less time to terminate a thread than a process.

Types of Threads

In the OS, there are two types of threads.

1. Kernel level thread.
2. User-level thread.

User-level thread

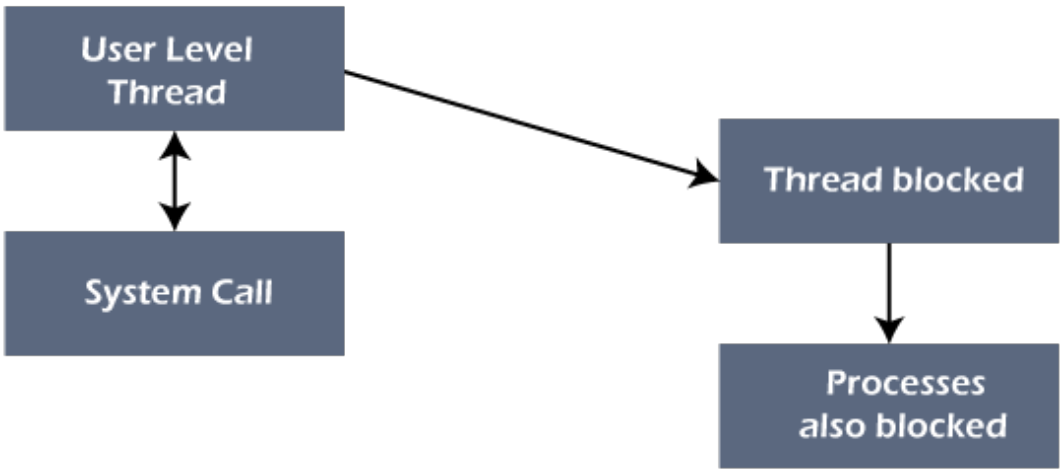
- ✓ The OS does not recognize the user-level thread. User threads can be easily implemented, and it is implemented by the user.
- ✓ If a user performs a user-level thread blocking operation, the whole process is blocked.
- ✓ The kernel level thread does not know nothing about the user level thread.
- ✓ The kernel-level thread manages user-level threads as if they are single-threaded processes?
- ✓ Examples: Java thread, POSIX threads, etc.

Advantages of User-Level Threads

- Implementation of the User-Level Thread is easier than Kernel Level Thread.
- Context Switch Time is less in User Level Thread.
- User-Level Thread is more efficient than Kernel-Level Thread.
- Because of the presence of only Program Counter, Register Set, and Stack Space, it has a simple representation.

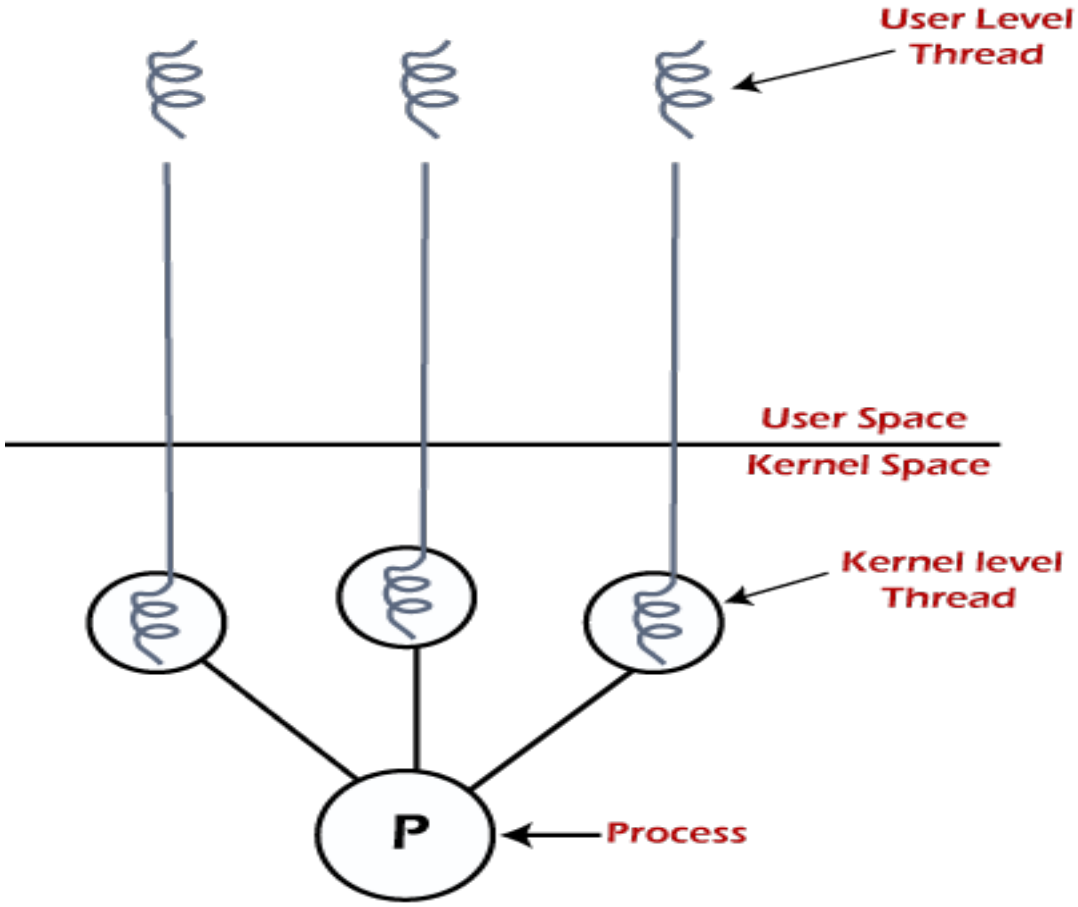
Disadvantages of User-Level Threads

- There is a lack of coordination between Thread and Kernel.
- In case of a page fault, the whole process can be blocked.



Kernel level thread

- ✓ The kernel thread recognizes the operating system.
- ✓ There is a thread control block and process control block in the system for each thread and process in the kernel-level thread.
- ✓ The kernel-level thread is implemented by the operating system.
- ✓ The kernel knows about all the threads and manages them.
- ✓ Example: Window ,Solaris.



Advantages of Kernel-level threads

1. The kernel-level thread is fully aware of all threads.
2. The scheduler may decide to spend more CPU time in the process of threads being large numerical.
3. The kernel-level thread is good for those applications that block the frequency.

Disadvantages of Kernel-level threads

1. The kernel thread manages and schedules all threads.
2. The implementation of kernel threads is difficult than the user thread.
3. The kernel-level thread is slower than user-level threads.

Components of Threads

1.Program Counter (PC):

- Stores the address of the next instruction to be executed by the thread.
- Each thread has its own PC, allowing them to independently follow their own execution path.

2.**Register Set:**

- A collection of registers used to store temporary data and control information during execution.
- Each thread has its own set of registers, ensuring isolation of their execution state.

3.**Stack:**

- A memory area used to store local variables, function parameters, and return addresses during function calls.
- Each thread has its own stack, preventing interference between their function calls.

Benefits of Threads

- **Enhanced throughput of the system:** When the process is split into many threads, and each thread is treated as a job, the number of jobs done in the unit time increases.
- **Effective Utilization of Multiprocessor system:** When you have more than one thread in one process, you can schedule more than one thread in more than one processor.
- **Faster context switch:** The context switching period between threads is less than the process context switching.
- **Responsiveness:** When the process is split into several threads, and when a thread completes its execution, that process can be responded to as soon as possible.
- **Communication:** Multiple-thread communication is simple because the threads share the same address space, while in process, we adopt just a few exclusive communication strategies for communication between two processes.
- **Resource sharing:** Resources can be shared between all threads within a process, such as code, data, and files.

Difference between User-Level & Kernel-Level Thread

S.N.	User-Level Threads	Kernel-Level Thread
1	User-level threads are faster to create and manage.	Kernel-level threads are slower to create and manage.
2	Implementation is by a thread library at the user level.	Operating system supports creation of Kernel threads.
3	User-level thread is generic and can run on any operating system.	Kernel-level thread is specific to the operating system.
4	Multi-threaded applications cannot take advantage of multiprocessing.	Kernel routines themselves can be multithreaded.

Threads Models (Many-to-one model, One-to-One Model, Many-to many model)

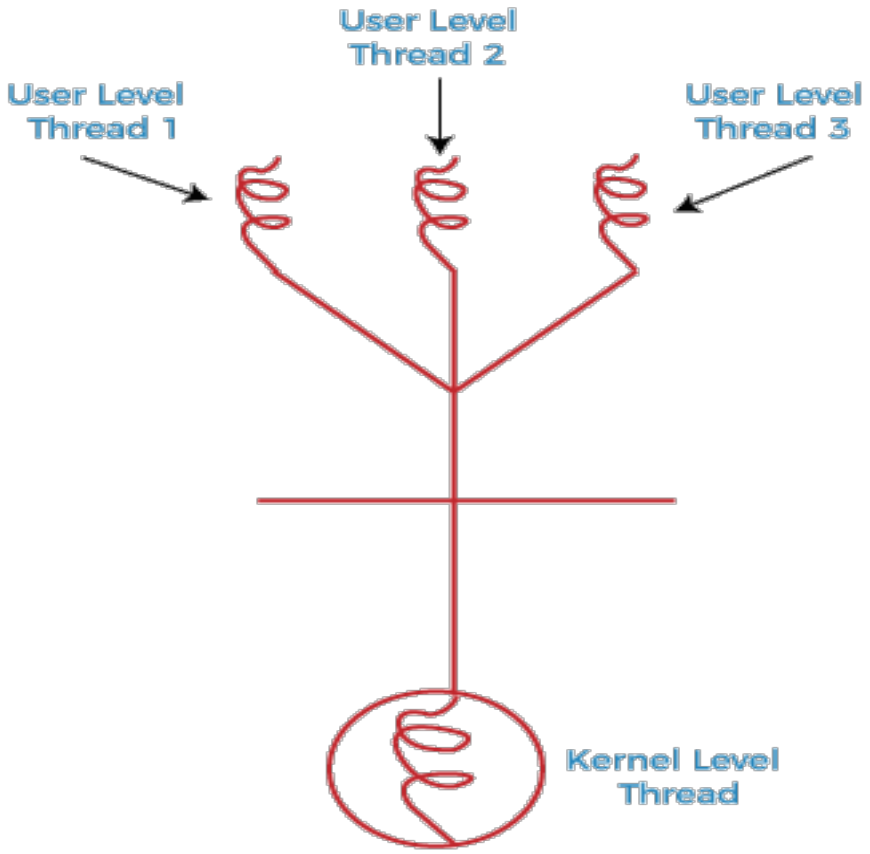
- ✓ A thread model defines how user-level threads are mapped to kernel-level threads.
- ✓ This mapping determines the level of concurrency, efficiency, and complexity of a multithreaded application.

Here are the three primary thread models:

1. Many-to-One Model

In this model, multiple user-level threads are mapped to a single kernel-level thread. This model is simple to implement but has limitations:

- **Single Thread Execution:** Only one user-level thread can execute at a time, even if multiple CPUs are available.
- **Blocking:** If one thread blocks, all other threads within the process are blocked.



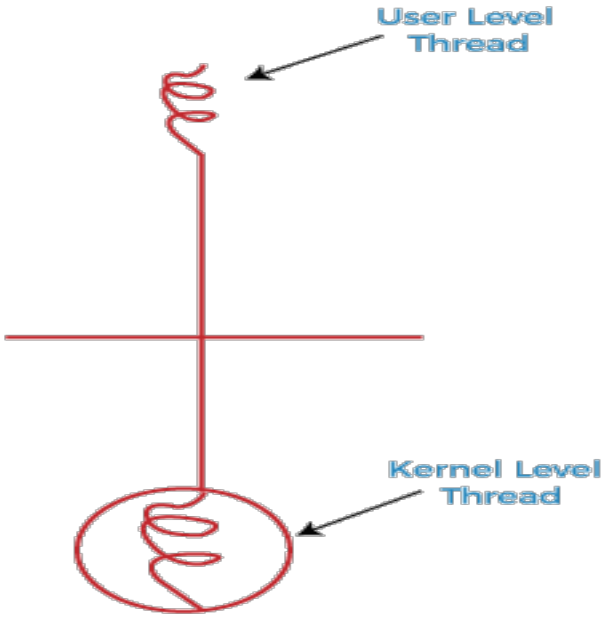
Many-to-one model

2. One-to-One Model

In this model, each user-level thread is mapped to a separate kernel-level thread. This model offers better concurrency and responsiveness:

- **Efficient Utilization of Multiple CPUs:** Multiple threads can execute concurrently on different CPUs.
- **Non-Blocking:** If one thread blocks, other threads can continue to execute.

However, this model can be less efficient for large numbers of threads due to the overhead of managing many kernel-level threads.

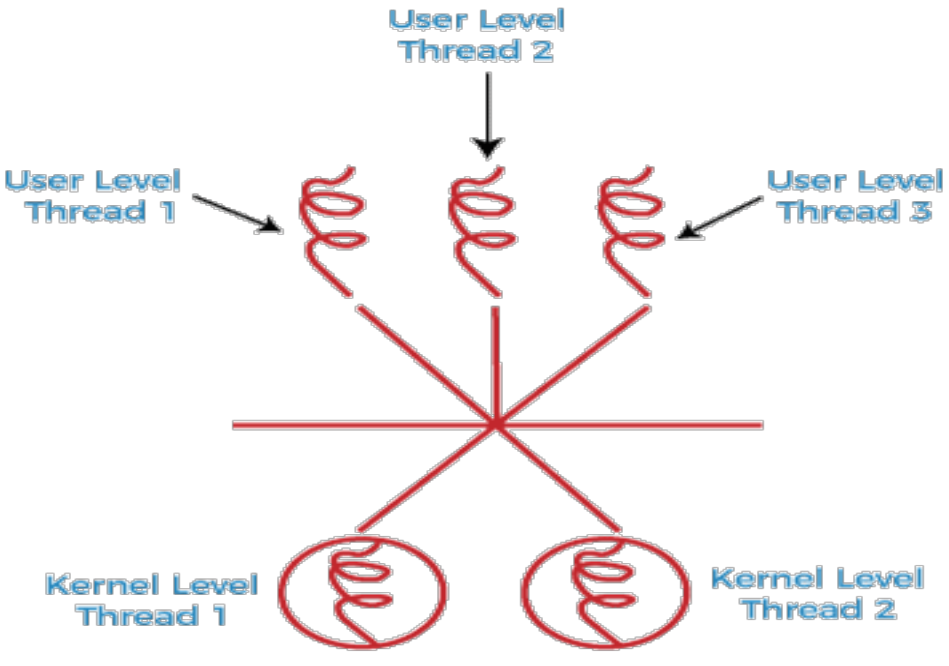


One-to-one model

3. Many-to-Many Model

This model provides a flexible balance between the previous two models:

- **Efficient Resource Utilization:** Multiple user-level threads can be mapped to a smaller number of kernel-level threads, reducing overhead.
- **Non-Blocking:** If one thread blocks, other threads can continue to execute.
- **Customizable Thread Scheduling:** The operating system can dynamically adjust the mapping between user-level and kernel-level threads to optimize performance.



Many-to-Many model

Choosing the Right Thread Model

The choice of thread model depends on the specific requirements of the application:

- **Many-to-One:** Suitable for simple applications with a small number of threads where efficiency is not a primary concern.
- **One-to-One:** Ideal for applications that require high concurrency and responsiveness, especially when utilizing multiple CPUs.
- **Many-to-Many:** A flexible option that can balance efficiency and responsiveness, making it suitable for a wide range of applications.

Types of Thread Process:

1. **Single-Threaded Process**

- A single-threaded process has only one thread of execution, meaning it can only perform one task at a time.
- The process runs sequentially, executing one instruction after another.

Characteristics:

- Simpler design and requires fewer resources for context management.
- Easier to implement and debug, as only one flow of control is present.
- Typically, there are no concurrency issues like race conditions since only one task is running.

Benefits of Single-Threaded Processes:

1. Simplicity: With only one execution path, it's easier to write, debug, and maintain the code.
2. Lower Overhead: No need for synchronization mechanisms, so it has less overhead in terms of memory and CPU.
3. Predictable Performance: Execution is sequential, so it's easier to predict and analyze the program's behavior.

Drawbacks:

- Cannot perform tasks in parallel, leading to underutilization of CPU in multi-core systems.
- Long-running operations can block the entire process, making the application unresponsive.

2. **Multi-Threaded Process**

- A multi-threaded process contains multiple threads within the same process, all sharing the same memory space.
- Each thread can execute concurrently, allowing for multiple tasks or operations to run simultaneously.

Characteristics:

- Each thread has its own stack and program counter but shares global memory with other threads in the same process.
- Threads can run in parallel (on multi-core processors) or interleave execution on a single-core CPU.
- The operating system or application provides synchronization mechanisms (like locks and semaphores) to handle access to shared resources.

Benefits of Multi-Threaded Processes:

1. **Responsiveness:** Threads allow a process to remain responsive by executing tasks in parallel. For instance, in a user interface, one thread can handle user input while another performs computations.
2. **Resource Sharing:** Threads within a process share resources (e.g., memory, files), reducing the need to duplicate resources as in separate processes.
3. **Better CPU Utilization:** On multi-core systems, multiple threads can execute simultaneously, improving CPU utilization and performance.
4. **Reduced Context Switching Overhead:** Switching between threads within the same process is faster and less resource-intensive than switching between separate processes.

Drawbacks:

- **Complexity:** Writing and managing multi-threaded applications is challenging, as it involves handling concurrency issues (e.g., race conditions, deadlock).
- **Synchronization Overhead:** Threads require synchronization mechanisms to prevent conflicts in accessing shared resources, which can lead to additional overhead.
- **Difficulty in Debugging:** Issues like race conditions and deadlocks make debugging multi-threaded applications more complex.

Example

Single-Threaded Example: A simple command-line application, like a calculator, where operations are performed one at a time in response to user input.

Multi-Threaded Example: A web browser where multiple threads handle different tasks—rendering web pages, playing videos, managing user input, and fetching data from the network—simultaneously to provide a smooth experience.

2.6. Hybrid Implementations of Threads:

- Hybrid thread models combine the best aspects of both user-level and kernel-level threads to provide a more efficient and flexible approach to multithreading.

Key Features of Hybrid Models

- **Efficient Resource Utilization:** Multiple user-level threads can be mapped to a smaller number of kernel-level threads, reducing overhead.
- **Non-Blocking:** If one thread blocks, other threads can continue to execute.
- **Customizable Thread Scheduling:** The operating system can dynamically adjust the mapping between user-level and kernel-level threads to optimize performance.

Common Hybrid Thread Models

1. **Many-to-Many (M:N) Model:**
    - **Multiple user-level threads** are mapped to a **smaller number of kernel-level threads**.
    - **User-level threads** are scheduled by a **user-level scheduler**.
    - **Kernel-level threads** are scheduled by the **kernel scheduler**.
    - **Advantages:** Efficient resource utilization, non-blocking behavior, and customizable scheduling.
    - **Example:** Windows NT and later.
2. **Two-Level Model:**
    - **Each process has one or more user-level threads**.

- **User-level threads** are mapped to **kernel-level threads** by the operating system.
- **Kernel-level threads** are scheduled by the **kernel scheduler**.
- **Advantages:** Improved performance and scalability compared to user-level threads, while still providing flexibility.
- **Example:** Solaris.

Hybrid Thread Model Implementation

Here's a simplified overview of how a hybrid thread model might be implemented:

1. **User-Level Threads:**
   - Created and managed by a user-level thread library.
   - Have their own program counter, stack, and registers.
   - Scheduled by a user-level scheduler.
2. **Kernel-Level Threads:**
   - Created and managed by the operating system kernel.
   - Have their own program counter, stack, and registers.
   - Scheduled by the kernel scheduler.
3. **Mapping:**
   - A mapping table is maintained to associate user-level threads with kernel-level threads.
   - When a user-level thread is ready to execute, it is mapped to an available kernel-level thread.
   - If a kernel-level thread blocks, other user-level threads associated with it can be scheduled on other kernel-level threads.

Advantages of Hybrid Thread Models

- **Efficient Resource Utilization:** Better utilization of multiple CPUs by mapping multiple user-level threads to fewer kernel-level threads.
- **Improved Performance:** Reduced context switching overhead compared to one-to-one models.
- **Enhanced Responsiveness:** Non-blocking behavior allows other threads to continue execution when one thread is blocked.
- **Increased Flexibility:** Customizable thread scheduling and dynamic mapping between user-level and kernel-level threads.

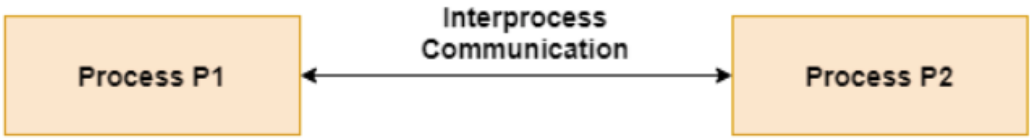
Difference between Threads and Processes

Process	Thread
Processes use more resources and hence they are termed as heavyweight processes.	Threads share resources and hence they are termed as lightweight processes.
Creation and termination times of processes are slower.	Creation and termination times of threads are faster compared to processes.
Processes have their own code and data/file.	Threads share code and data/file within a process.
Communication between processes is slower.	Communication between threads is faster.
Context Switching in processes is slower.	Context switching in threads is faster.
Eg: Opening two different browsers.	Eg: Opening two tabs in the same browser.

2.7. Inter-process Communication:

- Interprocess communication is the mechanism provided by the OS that allows processes to communicate with each other.
- This communication could involve a process letting another process know that some event has occurred or the transferring of data from one process to another.
- "Inter-process communication is used for exchanging useful information between numerous threads in one or more processes (or programs)."

A diagram that illustrates Interprocess communication is as follows −



This is essential for many tasks, such as:

- Sharing data
- Coordinating activities
- Managing resources
- Achieving modularity

Synchronization in Inter Process Communication

- Synchronization in Inter Process Communication (IPC) is the process of ensuring that multiple processes are coordinated and do not interfere with each other.
- This is important because processes can share data and resources, and if they are not synchronized, they can overwrite each other's data or cause other problems.

There are a number of different mechanisms that can be used to synchronize processes, including:

- **Mutual exclusion:** This is a mechanism that ensures that only one process can access a shared resource at a time. This is typically implemented using a lock, which is a data structure that indicates whether a resource is available.
- **Condition variables:** These are variables that can be used to wait for a certain condition to be met. For example, a process could wait for a shared resource to become available before using it.
- **Barriers:** These are synchronization points that all processes must reach before they can proceed. This can be used to ensure that all processes have completed a certain task before moving on to the next task.
- **Semaphores:** These are variables that can be used to count the number of times a shared resource is being used. This can be used to prevent a resource from being used more than a certain number of times at the same time.

Here are some examples of how synchronization is used in IPC:

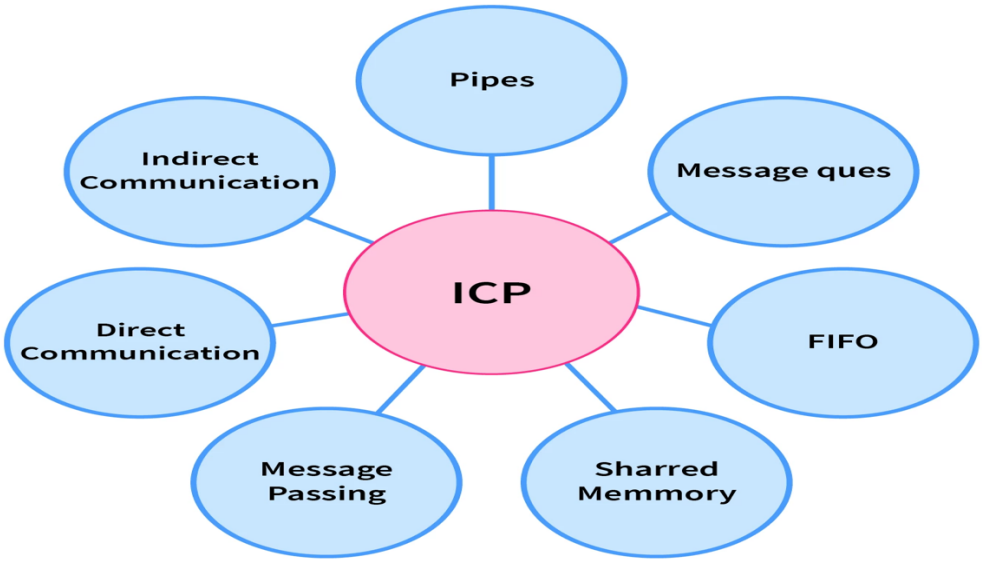
- **In a database**, multiple processes may need to access the same data. Synchronization is used to ensure that only one process can write to the data at a time, and that other processes do not read the data while it is being written.
- **In a web server**, multiple processes may need to handle requests from clients. Synchronization is used to ensure that only one process handles a request at a time, and that other processes do not interfere with the request.
- **In a distributed system**, multiple processes may need to communicate with each other. Synchronization is used to ensure that messages are sent and received in the correct order, and that processes do not take actions based on outdated information.

Need of IPC:

Inter-process communication (IPC) is required for a number of reasons:

- **Sharing data:** IPC allows processes to share data with each other. This is essential for many tasks, such as sharing files, databases, and other resources.
- **Coordinating activities:** IPC allows processes to coordinate their activities. This is essential for tasks such as distributed computing, where multiple processes are working together to solve a problem.
- **Managing resources:** IPC allows processes to manage resources such as memory, devices, and files. This is essential for ensuring that resources are used efficiently.
- **Achieving modularity:** IPC allows processes to be developed and maintained independently of each other. This makes it easier to develop and maintain large and complex software systems.
- **Flexibility:** IPC allows processes to run on different hosts or nodes in a network, providing greater flexibility and scalability in large and complex systems.

Approaches for Inter-Process Communication:



Pipe:-

The pipe is a type of data channel that is unidirectional in nature. It means that the data in this type of data channel can be moved in only a single direction at a time. Still, one can use two-channel of this type, so that he can able to send and receive data in two processes. Typically, it uses the standard methods for input and output. These pipes are used in all types of POSIX systems and in different versions of window operating systems as well.

Shared Memory:-

It can be referred to as a type of memory that can be used or accessed by multiple processes simultaneously. It is primarily used so that the processes can communicate with each other. Therefore the shared memory is used by almost all POSIX and Windows operating systems as well.

Message Queue:-

In general, several different messages are allowed to read and write the data to the message queue. In the message queue, the messages are stored or stay in the queue unless their recipients retrieve them. In short, we can also say that the message queue is very helpful in inter-process communication and used by all operating systems.

Message Passing:-

It is a type of mechanism that allows processes to synchronize and communicate with each other. However, by using the message passing, the processes can communicate with each other without restoring the hared variables.

Usually, the inter-process communication mechanism provides two operations that are as follows:

- send (message)
- received (message)

Direct Communication:-

In this type of communication process, usually, a link is created or established between two communicating processes. However, in every pair of communicating processes, only one link can exist.

Indirect Communication

Indirect communication can only exist or be established when processes share a common mailbox, and each pair of these processes shares multiple communication links. These shared links can be unidirectional or bi-directional.

FIFO:-

It is a type of general communication between two unrelated processes. It can also be considered as full duplex, which means that one process can communicate with another process and vice versa.

✓ **Race Conditions:**

- A race condition occurs when two or more threads or processes access a shared resource simultaneously, and the outcome of their operations depends on the specific timing of their execution.
- This can lead to unpredictable and often incorrect results.

Race condition might arise when:

1. **Shared Resource Access**: Multiple threads or processes attempt to read or write the same variable or memory location at nearly the same time.
2. **Lack of Synchronization**: If there's no mechanism (like locks or semaphores) to enforce an order or prevent simultaneous access, each thread might try to modify the resource, resulting in an inconsistent state.
3. **Unpredictable Behavior**: Since the exact order of thread execution can vary, the program's outcome becomes non-deterministic and can differ each time it runs, making debugging difficult.

Example:

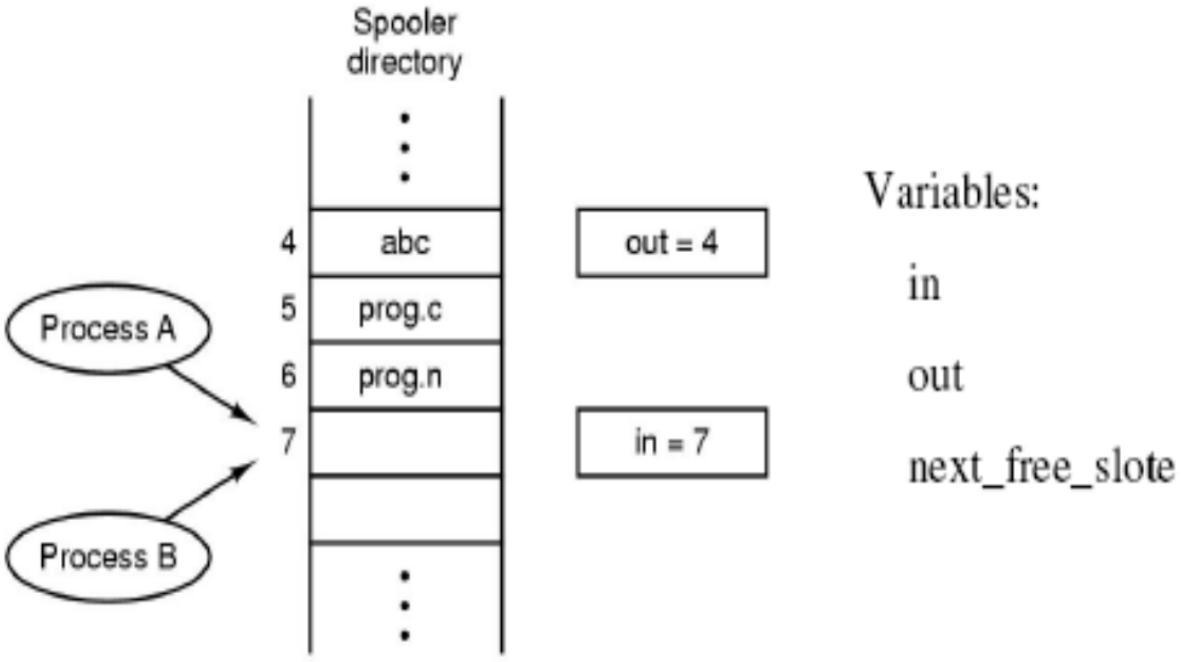


Fig: two processes want to access shared memory at the same time

Simultaneously, process A and B decide they want to queue a file for printing.
Let process A reads in and stores the value 7 in a local variable called next_free_slot. Just then clock interrupt occurs and CPU switches to process B. process B also reads in and gets a 7. It also stores it in a local variable next_free_slot. It stores the name of its file in slot 7 and updates it to be 8. Eventually, process A runs again starting from the place it left off. It looks at next_free_slot; finds 7, there and writes its filename in slot 7, erasing the name process B, just put there. Thus process B will never receive any output. The situation like this where two or more processes are reading or writing some shared data and the final result depends on who run precisely is called race condition.

✓ **Preventing Race Conditions:**

To avoid race conditions, **synchronization mechanisms** such as:

- **Locks (Mutexes)**: Only one thread can hold the lock and access the resource at a time.
- **Semaphores**: Control access to resources with a set number of available slots.
- **Atomic operations**: Ensures certain operations are completed without interference from other threads.

✓ **Mutual Exclusion:**

- Prohibiting more than one process from reading, writing the same data at the same time is called Mutual Exclusion. i.e. some way of making sure that if one process is using a shared variables or files, the other process will be excluded from doing the same thing.

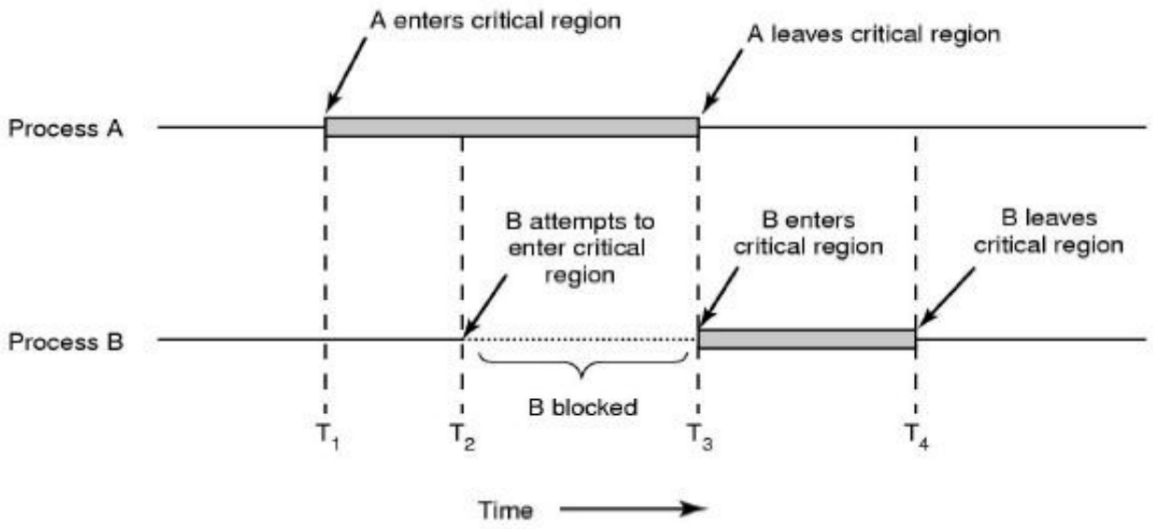
Example one process (say A) increments (use) the shared variables, all other processes desiring to do so at the same moment should be kept waiting, when that process (A) has finished accessing the shared variable, one of the processes waiting to do so should be allowed to processed. In this fashion, each process accessing the shared data excluded all other from doing from doing so simultaneously. This is called mutual exclusion.

✓ **Critical Regions:**

- The part of program in which shared variable is updated is called critical region or critical section.
- If we arrange the process in such a way that no two processes enter the critical section at the same time, then we could avoid race.
- When process is accessing shared data, the process is said to be in critical section.
- When one process is in its critical section all other processes are excluded from entering in its critical section.
- Each process must request permission to enter its critical section.

We need four conditions that hold a good solution for avoiding race condition:

- No two processes may be simultaneously inside their critical section.
- No process running outside its critical section may block other processes.
- When no process is in the critical section, any process requesting for entry in critical section must be permitted without delay.
- A process is granted entry in critical section for finite time only



Mutual exclusion using critical regions

Here process A enters its CR at time$T1$. A little later, at time $T2$ process B attempts to enter its CR but fails because another process is already in its CR and we allow only one at a time. Consequently, B is suspended until time $T3$ when A leaves its CR, allowing B to enter immediately. Eventually B leaves at $T4$ and we are back to the original situation with no process in their critical regions.

2.8. Mutual Exclusion with Busy Waiting:

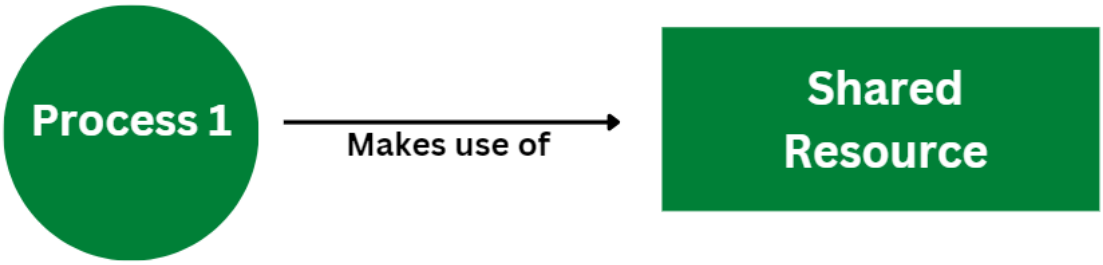
- Busy Waiting is defined as a process synchronization technique where the process waits and continuously keeps on checking for the condition to be satisfied before going ahead with its execution.
- Busy Waiting is also known as busy looping or spinning.
- The condition that is to be checked is the entry condition to be true such as availability of a resource or lock in the computer system.

Consider a scenario where a resource is required by the process for execution of a specific program. The resource is being used by another process and is unavailable at that moment. Therefore, the process needs to wait for the resource to become available.

Demonstration of Busy Waiting

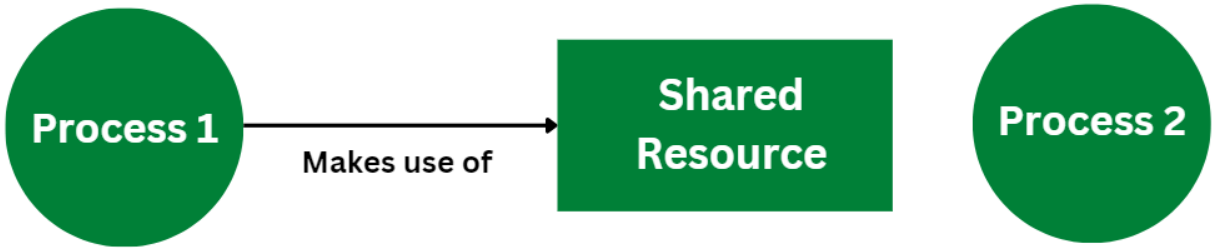
This process of busy waiting concerning the scenario is demonstrated below:

Step 1: Process 1 makes use of a shared resource



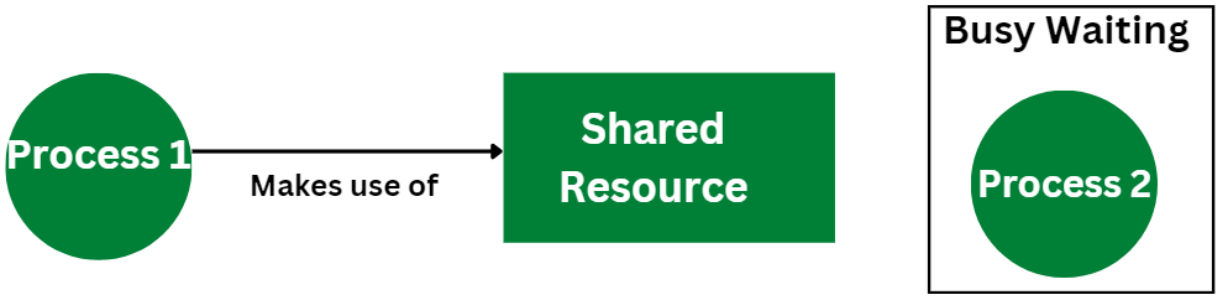
As shown in the above diagram, process 1 makes use of a shared resource. It releases the resource only once it has completed its task or any other high-priority process arises.

Step 2: Process 2 needs the shared resource that is already being used by process 1



As shown in above diagram the process 2 needs the shared resource, but the resource is already in used by the process 1. Therefore it needs to wait until the resource is free.

Step 3: Process 2 enters into busy waiting state until its get access to the shared resource



As shown in above diagram, the process 2 goes in busy waiting state that makes use of the processor and continuously checks for the shared resource to get allocated.

Need of Busy Waiting:

Busy waiting is required in operating system for achieving mutual exclusion. Mutual exclusion is used for preventing the processes from accessing the shared resources simultaneously. In operating system the critical section is defined as a program code in which concurrent access is avoided. In the critical section of processes they are granted with exclusive control for accessing it's resources without any interference from the other available processes in mutual exclusion.

Limitations of Busy Waiting:

Below are the limitations of Busy Waiting:

- Busy waiting keeps CPU busy at all the time until it's condition gets satisfied.
- The synchronisation mechanism that makes use of busy waiting suffers from the problem of priority inversion.
- In critical section the low priority processes gets executed.
- The system remains idle when process is in busy waiting state.
- Busy waiting consumes more power.

There are various ways or methods for achieving mutual exclusion, so that while one process is busy updating shared variable in its critical region, no other will enter its critical regions and cause the problems.

1. Interrupt Disabling

- The simples' solutions for achieving ME is to have each process disable all interrupts just after entering its critical region and re-enable them just before leaving it. With interrupts disable, no clock interrupts can occur. The CPU is only switches from process to process because of clock interrupts or other interrupts, but when interrupts are turned off the CPU will not be switched to another process. Thus, once a process has disabled interrupts, it can examine and update the shared variable without fear that any process will intervene.

```
DisableInterrupt()

//perform CR task

EnableInterrupt()
```

Advantages:

- ME can be achieved by implementing operating system primitives to disable and enable interrupts

Disadvantages:

- It is unwise to give user process the power to turn off interrupts.
- If the system is multiprocessor, with two or more CPUs, disabling interrupts affects only the CPU that executed the disabled instruction. The other ones will continue running and can access the shared memory, so it only works in single processor environment.

2. <u>Locked variable</u>:

- In this method a single shared (lock) variable is initially set to zero. When a process wants to enter its CR, it first checks the lock. If lock is 0, the process set it to 1 and enter the CR. If the lock us already 1, the process just waits until it becomes 0. Thus 0 means that no process is in its critical regions and 1 means that some process is in its critical regions.

Advantages:

- Seems no problem.
- ME can be achieved

Disadvantages:

- This idea contains exactly same fatal in spooler directory.

3. Strict Alternation:

- In this method, process share a common integer variable *turn*. If *turn* = = 0, then process *P*0 is allowed to execute in its critical region. If *turn* = = 1 then process *P*1 is allowed to execute in its critical region.
- Initially process *P*0 inspects turn, find it to be 0 and enter its CR. Process *P*1 also finds it to be 0 and therefore sits in a tight loop continually testing a variable until it becomes 1.
- Continuously testing a variable until some value appears is called ***busy waiting***. It should be avoided because it wastes CPU time.

Advantages:

- Ensures that only one process at a time can be in its critical regions.

Disadvantages:

- Taking *Turn* is not a good idea when one of the processes is much slower than the other.
- If process *P*0 just finishes critical regions and again need to enter critical regions and the process *P*1 is still busy at non critical regions, in this situation *turn* = = 1, but process *P*1 is busy at non critical work while process *P*0 wants to enter critical regions. This process *P*0 is blocked by process *P*1 , which violates the conditions of mutual exclusions (condition 3)

A. Dekkers Algorithms:

- Dekker's algorithm is the first known correct solution to the mutual exclusion problem in concurrent programming. Originally developed by Dekker in a different context, it was applied to the critical section problem by Dijkstra. It allows two threads to share a single-use resource without conflict, using only shared memory for communication. It avoids the strict alternation of an unaffected simplicity of nature or absence of artificiality of turn-taking algorithm, and was one of the first mutual exclusion algorithms to be invented.
- The mutual exclusion requirement is assured. No process will enter its critical section without setting its flag. Every process checks the other flag after setting its own. If both are set, the turn variable is used to allow only one process to proceed.
- The progress requirement is assured. The turn variable is only considered when both processes are using, or trying to use, the resource. If two processes attempt to enter a critical section at the same time, the algorithm will allow only one process in, based on whose turn it is. If one process is already in the critical section, the other process will busy wait for the first process to exit. This is done by the use of two flags, wants_to_enter[0] and wants_to_enter[1], which indicate an intention to enter the critical section on the part of processes 0 and 1, respectively, and a variable turn that indicates who has priority between the two processes.

Dekker's algorithm can be expressed in pseudocode, as follows

```
variables
wants_to_enter : array of 2 booleans
turn : integer
wants_to_enter[0] ← false
wants_to_enter[1] ← false
turn ← 0 // or 1

p0:
wants_to_enter[0] ← true
while wants_to_enter[1]
{
if turn ≠ 0 {
wants_to_enter[0] ← false
while turn ≠ 0 {
// busy wait
}
wants_to_enter[0] ← true
}
}
// critical section
...
turn ← 1
wants_to_enter[0] ← false
// remainder section

p1:
wants_to_enter[1] ← true
while wants_to_enter[0] {

if turn ≠ 1 {
wants_to_enter[1] ← false
while turn ≠ 1 {
// busy wait
}
wants_to_enter[1] ← true
}
}
// critical section
...
turn ← 0
wants_to_enter[1] ← false
// remainder section
```

Dekker's algorithm guarantees mutual exclusion, freedom from deadlock, and freedom from starvation

B. Petersons' Algorithm:

```
#define FALSE 0
#define TRUE 1
#define N   2                                   /* number of processes */
int turn;                                       /* whose turn is it? */
int interested[N];                              /* all values initially 0 (FALSE)*/
void enter_region(int process)                  /* process is 0 or 1 */
{
    int other;                                  /* number of the other process */
    other = 1 - process;                        /* the opposite of process */
    interested[process] = TRUE;                 /* show that we are interested */
    turn = process;                             /* set flag */
    while (turn == process && interested[other] == TRUE) /* null statement */;
}
void leave_region(int process)                  /* process: who is leaving */
{
    interested[process] = FALSE;                /* indicate departure from critical region */
}
```

- Before using the shared variables (i.e. before entering its CR), each process calls *enter_region* with its own parameter number 0 or 1, as parameter. This call will cause it to wait, if need be, until it is safe to enter. After it has finished with the shared variables, the process calls *leave_region* to indicate that it is done and to allow other process to enter, if it so desires.
- Let us see how this algorithm works. Initially neither process is in its critical region. Now, process 0 calls *enter_region*. It indicates its interest by setting its array element and sets turn to 0. Since process 1 is not interested, *enter_region* returns immediately. If process 1 now calls *enter_regions*, it will hang there until *interested [0]* goes to FALSE, an event that only happens when process 0 calls *leave_region* to exit the CR.

Advantages:

- preserves all condition of ME.

Disadvantages:

- Difficulty to program for n-process system and less efficient

4. Hardware Solutions –TSL (Test and Set Lock)

- (Test and Set Lock) that works as follows: it reads the contents of the memory word LOCK into register RX and then stores a nonzero value at the memory address LOCK.
- The operations of reading the word and storing into it are guaranteed to be indivisibleno other processor can access the memory word until the instruction is finished.
- The CPU executing the TSL instruction locks the memory bus to prohibit other CPUs from accessing memory until it is done

The solution for critical section problem using this mechanism is shown below:

```
enter_region:
    TSL REGISTER,LOCK    |copy LOCK to register and set LOCK to 1
    CMP REGISTER,#0      |was LOCK zero?
    JNE enter_region     |if it was non zero, LOCK was set, so loop
    RET                  |return to caller; critical region entered
leave_region:
    MOVE LOCK, #0        |store a 0 in LOCK
    RET                  |return to caller
```

Advantages:

- Preserves all conditions, easier programming task and improve system efficiency.

Problem:

- Difficulty in hardware design.

Alternative of Busy waiting:

1. Sleep and Wake up:

- Sleep and wakeup are system calls that blocks process instead of wasting CPU time when they are not allowed to enter their critical region. sleep is a system call that causes the caller to block, that is, be suspended until another process wakes it up. The wakeup call has one parameter, the process to be awakened. remove an item and decrement the counter.

  Example of Sleep and Wake Up :

Producer-Consumer problem

- It is a classical synchronization problem in the operating system.
- With the presence of more than one process and limited resources in the system the synchronization problem arises.
- If one resource is shared between more than one process at the same time then it can lead to data inconsistency.
- In the producer-consumer problem, the producer produces an item and the consumer consumes the item produced by the producer.
- In operating System Producer is a process which is able to produce data/item.
- Consumer is a Process that is able to consume the data/item produced by the Producer.
- Both Producer and Consumer share a common memory buffer.
- This buffer is a space of a certain size in the memory of the system which is used for storage.
- The producer produces the data into the buffer and the consumer consumes the data from the buffer.

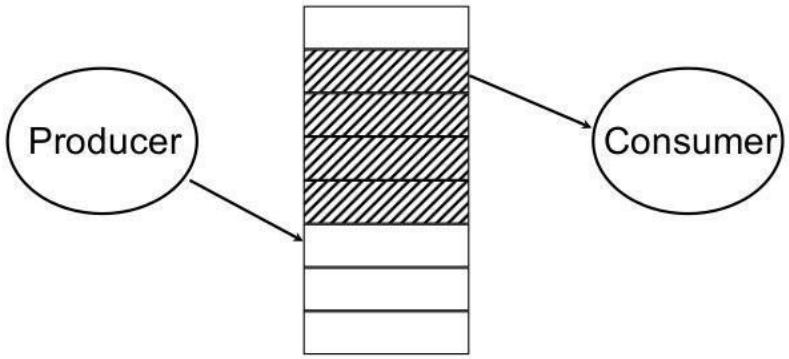
Problem arises when

1. The producer wants to put a new data in the buffer, but buffer is already full.

Solution: Producer goes to sleep and to be awakened when the consumer has removed data.

2. The consumer wants to remove data the buffer but buffer is already empty.

Solution: Consumer goes to sleep until the producer puts some data in buffer and wakes consumer up.



We introduce the Producer and Consumer methodology. Producer makes the item and consumed by the consumer.
The pseudo code is follows:

```
#define N 100                                   /* number of slots in the buffer */
int count = 0;                                  /* number of items in the buffer */
void producer(void)
{
    int item;
    while (TRUE){                               /* repeat forever */
        item = produce_item();                  /* generate next item */
        if (count == N) sleep();                /* if buffer is full, go to sleep
        insert_item(item); count                */ /* put item in buffer */
        = count + 1;                            /* increment count of items in buffer
        if (count == 1) wakeup(consumer);       */ /* was buffer empty? */
    }
}
void consumer(void)
{
    int item;
    while (TRUE){                               /* repeat forever */
        if (count == 0) sleep();                /* if buffer is empty, got to sleep
        item = remove_item();                   */ /* take item out of buffer */
        count = count - 1;                      /* decrement count of items in
 buffer */
        if (count ==N - 1) wakeup(producer);    /* was buffer full? */
        consume_item(item);                     /* print item */
    }
}
```

Fig: The producer-consumer problem with a fatal race condition

N → Size of Buffer
Count → a variable to keep track of the number of items in the buffer.

Producer code:

- The producers code is first test to see if count is N.
- If it is, the producer will go to sleep; if it is not the producer will add an item and increment count.

Consumer code:

- It is similar as of producer. First test count to see if it is 0. If it is, go to sleep; if it nonzero remove an item and decrement the counter.
- Each of the process also tests to see if the other should be awakened and if so wakes it up.
- This approach sounds simple enough, but it leads to the same kinds of race conditions as we saw in the spooler directory.

1. The buffer is empty and the consumer has just read count to see if it is 0.
2. At that instant, the scheduler decides to stop running the consumer temporarily and start running the producer. (Consumer is interrupted and producer resumed)

- The producer creates an item, puts it into the buffer, and increases count.
- Because the buffer was empty prior to the last addition (count was just 0), the producer tries to wake up the consumer.
- Unfortunately, the consumer is not yet logically asleep, so the **wakeup signal** is lost.
- When the consumer next runs, it will test the value of count it previously read, find it to be 0, and go to sleep.
- Sooner or later the producer will fill up the buffer and also go to sleep. Both will sleep forever.

The essence of the problem here is that a wakeup sent to a process that is not (yet) sleeping is lost. For temporary solution we can use wakeup waiting bit to prevent wakeup signal from getting lost, but it can't work for more processes.

2.9. Types of Mutual Exclusion:

1. Semaphores:

- A semaphore is a synchronization primitive used to control access to shared resources.
- It's a more flexible mechanism than locks, allowing for more complex synchronization scenarios.
- **A semaphore is a special kind of integer variable which can be initialized and can be accessed only through two atomic operations.**

Syntax:

The syntax of the semaphore may be used as:

```
1. // Wait Operation
2.     wait(Semaphore S) {
3.         while (S<=0);
4.             S--;
5.     }
6.     // Signal Operation
7.     signal(Semaphore S) {
8.         S++;
```

9. }

Types of Semaphores:

1. **Binary Semaphore:**
    - Can have two values: 0 or 1.
    - Often used to implement mutual exclusion, similar to a lock.
    - When the semaphore is 1, a thread can acquire it and enter the critical section.
    - When the semaphore is 0, no thread can acquire it.
2. **Counting Semaphore:**
    - Can have a non-negative integer value.
    - Used to control access to a fixed number of resources.
    - The initial value of the semaphore represents the number of available resources.
    - A thread can acquire the semaphore if its value is greater than 0.
    - After acquiring, the semaphore's value is decremented.
    - When a thread releases the semaphore, its value is incremented.

Working of Semaphores:

1. **Initialization:** A semaphore is initialized with a specific value.
2. **Wait (P) Operation:** A thread attempts to acquire the semaphore.
    - If the semaphore's value is greater than 0, the thread acquires it and decrements the value.
    - If the value is 0, the thread is blocked until the semaphore's value becomes positive.
3. **Signal (V) Operation:** A thread releases the semaphore.
    - The semaphore's value is incremented.
    - If there are any blocked threads waiting on the semaphore, one of them is awakened.

Advantages of Semaphores:

- **Flexibility:** Semaphores can be used for various synchronization tasks, beyond simple mutual exclusion.
- **Efficient Resource Management:** Counting semaphores can efficiently control access to a fixed number of resources.
- **Scalability:** Semaphores can be used in complex systems with multiple threads or processes.

Disadvantages of Semaphores:

- **Complexity:** Semaphores can be more complex to use than locks, especially when dealing with multiple semaphores and complex synchronization patterns.
- **Deadlocks:** Improper use of semaphores can lead to deadlocks, where two or more threads are waiting for each other to release resources.
- **Starvation:** It's possible for a thread to be indefinitely delayed if other threads continuously acquire the semaphore.

Producer-Consumer Problem using Semaphores

- The producer-consumer problem is a classic concurrency problem where one or more producers produce data and one or more consumers consume that data.
- The challenge lies in ensuring that the producers and consumers access a shared buffer in a synchronized manner, preventing race conditions and data corruption.

Solution using Semaphores:

To solve this problem using semaphores, we typically use three semaphores:

1. **Mutex:** This semaphore ensures mutual exclusion, allowing only one process (producer or consumer) to access the shared buffer at a time.
2. **Empty:** This semaphore counts the number of empty slots in the buffer. Initially, it's set to the buffer size.
3. **Full:** This semaphore counts the number of full slots in the buffer. Initially, it's set to 0.

Producer Process:

1. **Wait on Empty:** The producer waits on the empty semaphore. This ensures that there's at least one empty slot in the buffer.
2. **Wait on Mutex:** The producer waits on the mutex semaphore to gain exclusive access to the buffer.
3. **Produce Item:** The producer produces an item and places it in the buffer.
4. **Signal Mutex:** The producer signals the mutex semaphore, releasing the lock on the buffer.
5. **Signal Full:** The producer signals the full semaphore, indicating that one more slot in the buffer is full.

Consumer Process:

1. **Wait on Full:** The consumer waits on the full semaphore. This ensures that there's at least one item in the buffer.
2. **Wait on Mutex:** The consumer waits on the mutex semaphore to gain exclusive access to the buffer.
3. **Consume Item:** The consumer consumes an item from the buffer.
4. **Signal Mutex:** The consumer signals the mutex semaphore, releasing the lock on the buffer.
5. **Signal Empty:** The consumer signals the empty semaphore, indicating that one more slot in the buffer is empty.

Code Example (Pseudocode):

```
// Shared variables
int buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
// Semaphores
semaphore mutex = 1;
semaphore empty = BUFFER_SIZE;
semaphore full = 0;
// Producer Process
while (true) {
   wait(empty);
```

```
    wait(mutex);
    // Produce item and put it in buffer[in]
    in = (in + 1) % BUFFER_SIZE;
    signal(mutex);
    signal(full);
}
// Consumer Process
while (true) {
    wait(full);
    wait(mutex);
    // Consume item from buffer[out]
    out = (out + 1) % BUFFER_SIZE;
    signal(mutex);
    signal(empty);}
```

2.**Monitors:**

- It is a synchronization technique that enables threads to mutual exclusion and the **wait()** for a given condition to become true.
- It is an abstract data type. It has a shared variable and a collection of procedures executing on the shared variable.
- A process may not directly access the shared data variables, and procedures are required to allow several processes to access the shared data variables simultaneously.
- At any particular time, only one process may be active in a monitor.
- Other processes that require access to the shared variables must queue and are only granted access after the previous process releases the shared variables.

Syntax:

The syntax of the monitor may be used as:

```
1.  monitor  {
2.      //shared variable declarations
3.      data variables;
4.      Procedure P1() { ... }
5.      Procedure P2() { ... }
6.      .
7.      .
8.      .
9.      Procedure Pn() { ... }
10.     Initialization Code() { ... }
11. }
```

Advantages and Disadvantages of Monitor

Advantages

1. Mutual exclusion is automatic in monitors.
2. Monitors are less difficult to implement than semaphores.
3. Monitors may overcome the timing errors that occur when semaphores are used.
4. Monitors are a collection of procedures and condition variables that are combined in a special type of module.

Disadvantages

1. Monitors must be implemented into the programming language.
2. The compiler should generate code for them.
3. It gives the compiler the additional burden of knowing what operating system features is available for controlling access to crucial sections in concurrent processes.

Producer-Consumer Problem Using Monitors

Understanding the Problem:

The producer-consumer problem is a classic concurrency problem where one or more producers generate data and one or more consumers consume that data. The challenge is to synchronize the access to a shared buffer to avoid race conditions and data corruption.

Solution Using Monitors:

Monitors offer a structured approach to synchronization, making it easier to solve the producer-consumer problem. Here's a basic outline of how it can be implemented:

Monitor Definition:

```
monitor ProducerConsumer {
   int buffer[BUFFER_SIZE];
   int in, out;
   condition notFull, notEmpty;

   procedure produce(item) {
      while (in == (out + BUFFER_SIZE) % BUFFER_SIZE) {
         wait(notFull);
      }
      buffer[in] = item;
      in = (in + 1) % BUFFER_SIZE;
      signal(notEmpty);
   }

   procedure consume() {
      while (in == out) {
         wait(notEmpty);
      }
      item = buffer[out];
      out = (out + 1) % BUFFER_SIZE;
      signal(notFull);
   }
}
```

Explanation:

1. **Shared Data:**
   - buffer: The shared buffer to store produced items.
   - in: Index of the next empty slot in the buffer.
   - out: Index of the next item to be consumed from the buffer.
2. **Condition Variables:**
   - notFull: Signaled when there's an empty slot in the buffer.
   - notEmpty: Signaled when there's an item in the buffer to be consumed.
3. **Producer Procedure:**

- Checks if the buffer is full. If so, waits on the notFull condition.
- Produces an item and places it in the buffer.
- Signals the notEmpty condition to wake up a waiting consumer.

4. **Consumer Procedure:**
   - Checks if the buffer is empty. If so, waits on the notEmpty condition.
   - Consumes an item from the buffer.
   - Signals the notFull condition to wake up a waiting producer.

3. Message Passing:

With the trend of distributed operating system, many operating systems are used to communicate through internet, intranet and remote data processing etc.
Message passing method of inter-process communication uses two primitives send and receive.

MP Synchronization

1. Blocking sends, blocking receive
2. Nonblocking send, blocking receive
3. Nonblocking send, nonblocking receive

MP Addressing

- **Direct addressing**: send to a specific process; receive from a specific process or allow from any process but be able to determine the source
- **Indirect addressing:** send message to an intermediate structure (queue called a mailbox), receive a message from the mailbox.

Solving producer- consumer problem using monitors:

```
#define N 100                                   /*number of slots in the buffer*/
void producer(void)
{    int item;
     message m;                                 /*message buffer*/
     while (TRUE){
          item = produce_item();       /*generate something */
          receive(consumer, &m);       /*wait for an empty to arrive*/
          build_message(&m, item);  /*construct a message to send*/
          send(consumer, &m); }
}
void consumer(void)
{    int item;
     message m;
     for(i = 0; i<N; i++) send(producer, &m);      /*send N empties*/
     while(TRUE){
          receive(producer, &m);          /* get message containing item*/
          item = extract_item(&m);        /* extract item from message*/
          send(producer, &m);             /* send back empty reply*/
          consume_item(item);             /*do something with item*/
     }}
```

4. **Locks(Mutex):**

- A **mutex** (short for "mutual exclusion") is a synchronization primitive used to protect shared resources in multi-threaded environments.
- It allows only one thread to hold the lock and access the critical section at any given time, enforcing mutual exclusion.
- Other threads that attempt to acquire the lock while it's held must wait until it becomes available.

Syntax of Locks (Mutexes)

```
#include <pthread.h>
pthread_mutex_t lock;
void initialize() {
   pthread_mutex_init(&lock, NULL); // Initialize the mutex
}
void critical_section() {
   pthread_mutex_lock(&lock);       // Acquire the lock
   // Critical section code here
   pthread_mutex_unlock(&lock);     // Release the lock
}
void cleanup() {
   pthread_mutex_destroy(&lock);    // Destroy the mutex
}
```

How Mutexes Work

1. **Locking**: When a thread enters the critical section, it attempts to "lock" the mutex.
    - If the mutex is unlocked, the thread acquires it and enters the critical section.
    - If the mutex is already locked by another thread, the thread will be put in a waiting state until the lock is released.
2. **Unlocking**: When the thread exits the critical section, it "unlocks" the mutex, allowing other waiting threads to acquire it.
3. **Blocking**: Threads that attempt to lock an already locked mutex are either blocked or placed in a queue until the mutex is available. Once the lock is released, the next thread in line acquires the lock and enters the critical section.
4. **Automatic Release**: In some languages (like C++ with std::lock_guard), locks are automatically released when they go out of scope, simplifying the syntax and reducing errors.

Advantages of Locks (Mutexes)

1. **Simple to Use**: Mutexes provide a straightforward and clear way to enforce mutual exclusion in critical sections.
2. **Efficient for Short Critical Sections**: Mutexes work well for short critical sections, where locking and unlocking are done quickly without significant blocking.
3. **Automatic Handling in High-Level Languages**: Some languages and libraries handle locking and unlocking automatically, reducing the risk of bugs.
4. **Supports Priority**: In some systems, priority inversion protocols can be used to handle priority in lock acquisition.
5. **Scalable**: With careful design, mutexes can be used to scale applications across multiple threads in multi-core environments.

Disadvantages of Locks (Mutexes)

1. **Deadlock Risk**: If two or more threads lock resources in a circular dependency, they can create a deadlock where none can proceed.
2. **Risk of Starvation**: If one thread holds the mutex for too long or if high-priority threads consistently acquire it, lower-priority threads may be starved.
3. **Blocking and Performance Impact**: When a thread is waiting for a lock, it can be in a blocked state, leading to wasted CPU cycles and reduced performance.
4. **Complexity in Managing Multiple Locks**: Managing multiple locks in nested or interdependent critical sections increases complexity and the risk of deadlock.
5. **Overhead**: Locking and unlocking operations introduce overhead, which can become significant in highly concurrent applications with high lock contention.

Producer-Consumer Problem:

- It is a classic synchronization problem in concurrent programming.
- It involves two types of processes: **producers**, which generate data and add it to a buffer, and **consumers**, which take data from the buffer and process it.
- The challenge is to ensure that producers don't add data when the buffer is full and consumers don't remove data when the buffer is empty.

In this solution, we'll use a **mutex** to ensure mutual exclusion while accessing the shared buffer, and two **condition variables** to synchronize the producer and consumer actions.

Problem Setup

- A shared buffer with limited capacity.
- **Producers** add items to the buffer if it's not full.
- **Consumers** remove items from the buffer if it's not empty.

Solution Using Mutex and Condition Variables (C++ Example with Pthreads)

```
#include <iostream>
#include <thread>
#include <mutex>
#include <condition_variable>
#include <queue>
const int BUFFER_SIZE = 5; // Buffer capacity
std::queue<int> buffer;     // Shared buffer
std::mutex mtx;             // Mutex for mutual exclusion
std::condition_variable not_full, not_empty;  // Condition variables
void producer(int id) {
   int item = 0;
   while (true) {
      std::unique_lock<std::mutex> lock(mtx);  // Lock the mutex

      // Wait until the buffer is not full
      not_full.wait(lock, [] { return buffer.size() < BUFFER_SIZE; });
      // Produce an item
      item++;
      buffer.push(item);
      std::cout << "Producer " << id << " produced item: " << item << "\n";
```

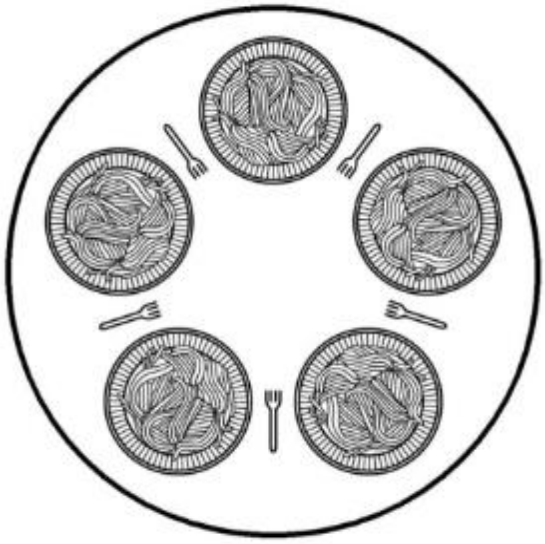
```
        // Notify the consumer that the buffer is not empty
        not_empty.notify_all();
        lock.unlock();  // Unlock the mutex
        std::this_thread::sleep_for(std::chrono::milliseconds(100));  // Simulate work
    }
}
void consumer(int id) {
    while (true) {
        std::unique_lock<std::mutex> lock(mtx);  // Lock the mutex
        // Wait until the buffer is not empty
        not_empty.wait(lock, [] { return !buffer.empty(); });
        // Consume an item
        int item = buffer.front();
        buffer.pop();
        std::cout << "Consumer " << id << " consumed item: " << item << "\n";

        // Notify the producer that the buffer is not full
        not_full.notify_all();

        lock.unlock();  // Unlock the mutex

        std::this_thread::sleep_for(std::chrono::milliseconds(150));  // Simulate work
    }
}

int main() {
    std::thread producers[2], consumers[2];

    // Create two producer threads
    for (int i = 0; i < 2; i++) {
        producers[i] = std::thread(producer, i + 1);
    }
    // Create two consumer threads
    for (int i = 0; i < 2; i++) {
        consumers[i] = std::thread(consumer, i + 1);
    }
    // Join the threads
    for (int i = 0; i < 2; i++) {
        producers[i].join();
        consumers[i].join();
    }
    return 0;
}
```

2.10. Classical IPC problems:

1. **Dining Philosophers Problems:**

- There are N philosophers sitting around a circular table eating spaghetti and discussing philosophy.
- The problem is that each philosopher needs 2 forks to eat, and there are only N forks, one between each 2 philosophers.
- Design an algorithm that the philosophers can follow that ensures that none starves as long as each philosopher eventually stops eating, and such that the maximum number of philosophers can eat at once.

✓ Philosophers eat/think
✓ Eating needs 2 forks
✓ Pick one fork at a time
✓ How to prevent deadlock



Problem:
From theroritical view point the solution is adequate but from practical one, it has a performance bug;
only one philosopher can be eating at any instance with five forks available, we should be able to allow two should be able to allow two philosophers to eat at the same time.
Solution :

```
#define N           5           /* number of philosophers */
#define LEFT        (i+N-1)%N   /* number of i's left neighbor */
#define RIGHT       (i+1)%N     /* number of i's right neighbor */
#define THINKING    0           /* philosopher is thinking */
#define HUNGRY      1           /* philosopher is trying to get forks */
#define EATING      2           /* philosopher is eating */
typedef int semaphore;          /* semaphores are a special kind of int */
int state[N];                   /* array to keep track of everyone's state */
semaphore mutex = 1;            /* mutual exclusion for critical regions */
semaphore s[N];                 /* one semaphore per philosopher */
void philosopher(int i)         /* i: philosopher number, from 0 to N1 */
{
    while (TRUE){               /* repeat forever */
        think();                /* philosopher is thinking */
        take_forks(i);          /* acquire two forks or block */
        eat();                  /* yum-yum, spaghetti */
        put_forks(i);           /* put both forks back on table */
    }
}
void take_forks(int i)          /* i: philosopher number, from 0 to N1 */
{
    down(&mutex);               /* enter critical region */
    state[i] = HUNGRY;          /* record fact that philosopher i is hungry */
    test(i);                    /* try to acquire 2 forks */
    up(&mutex);                 /* exit critical region */
    down(&s[i]);                /* block if forks were not acquired */
}
}
void put_forks(i)               /* i: philosopher number, from 0 to N1 */
{
    down(&mutex);               /* enter critical region */
    state[i] = THINKING;        /* philosopher has finished eating */

    test(LEFT);                 /* see if left neighbor can now eat */
    test(RIGHT);                /* see if right neighbor can now eat */
    up(&mutex);                 /* exit critical region */
}
void test(i)                    /* i: philosopher number, from 0 to N1* /
{
    if (state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
        state[i] = EATING;
        up(&s[i]);
    }
}
```

Here, we use an array state to keep track of whether a philosopher is EATING, THINKING or HUNGRY (trying to acquire forks). A philosopher may move into EATING state if neither neighbor is EATING. Philosopher i's neighbor are defined by LEFT and RIGHT. In other words, if i is 2, LEFT is 1 and RIGHT is 3.
The problem uses an array of semaphores, one per philosopher, so HUNGRY philosophers can block if the needed forks are busy. Here each process runs the procedure philosopher as its main code, but the other procedure, take_forks, put_forks and test are ordinary procedure and not separated processes.

2. Reader and Writers Problem:

- The dining philosopher's problem is useful for modeling processes that are competing for exclusive access to a limited number of resources, such as I/O devices.
- Another famous problem is the readers and writers' problem which models access to a database.
- Imagine, for example, an airline reservation system, with many competing processes wishing to read and write it.
- It is acceptable to have multiple processes reading the database at the same time, but if one process is updating (writing) the database, no other process may have access to the database, not even a reader.

The question is how do we program the readers and the writers? One solution is shown below.

Solution to Readers Writer problems

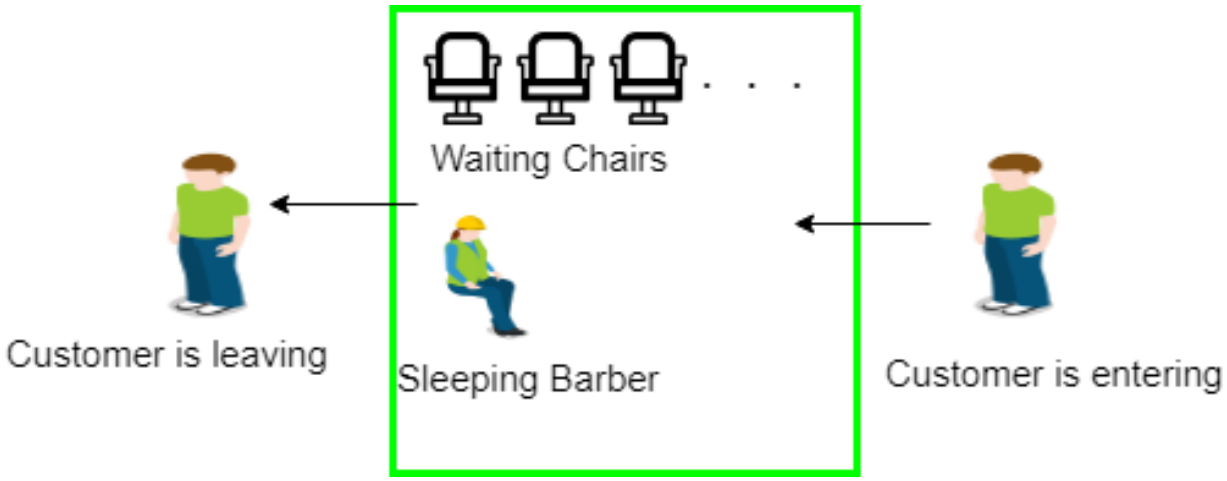
```
typedef int semaphore;                  /* use our imagination */
semaphore mutex = 1;                    /* controls access to 'rc' */
semaphore db = 1;                       /* controls access to the database */
int rc = 0;                             /* # of processes reading or wanting to */
void reader(void)
{
    while (TRUE){                       /* repeat forever */
        down(&mutex);                   /* get exclusive access to 'rc' */
        rc = rc + 1;                    /* one reader more now */
        if (rc == 1) down(&db);         /* if this is the first reader ... */
        up(&mutex);                     /* release exclusive access to 'rc' */
        read_data_base();               /* access the data */
        down(&mutex);                   /* get exclusive access to 'rc' */
        rc = rc 1;                      /* one reader fewer now */
        if (rc == 0) up(&db);           /* if this is the last reader ... */
        up(&mutex);                     /* release exclusive access to 'rc' */
        use_data_read();                /* noncritical region */
    }
}

void writer(void)
{
    while (TRUE){                       /* repeat forever */
        think_up_data();                /* noncritical region */
        down(&db);                      /* get exclusive access */
        write_data_base();              /* update the data */
        up(&db);                        /* release exclusive access */
    }
}
```

3. Sleeping Barber Problem:

Problem : The analogy is based upon a hypothetical barber shop with one barber. There is a barber shop which has one barber, one barber chair, and n chairs for waiting for customers if there are any to sit on the chair.

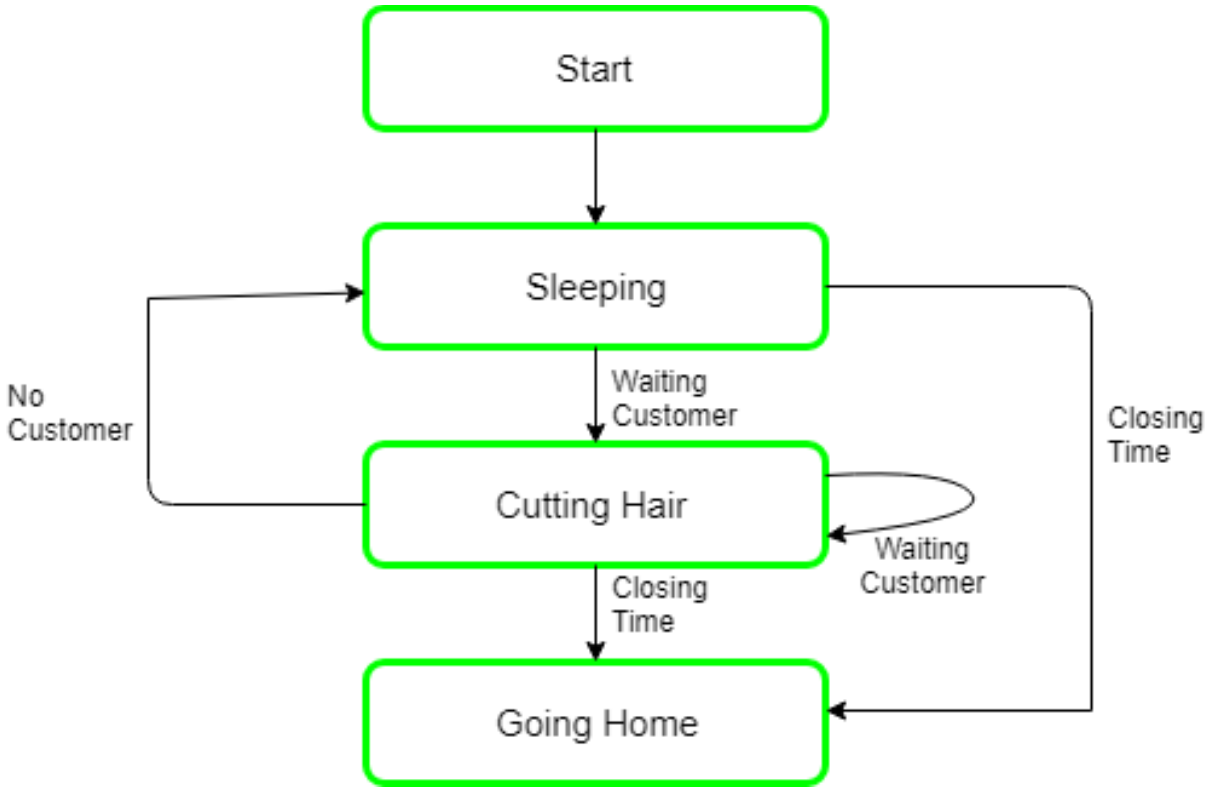
- If there is no customer, then the barber sleeps in his own chair.
- When a customer arrives, he has to wake up the barber.
- If there are many customers and the barber is cutting a customer's hair, then the remaining customers either wait if there are empty chairs in the waiting room or they leave if no chairs are empty.

.



Solution :

- The solution to this problem includes three semaphores.First is for the customer which counts the number of customers present in the waiting room (customer in the barber chair is not included because he is not waiting). Second, the barber 0 or 1 is used to tell whether the barber is idle or is working, And the third mutex is used to provide the mutual exclusion which is required for the process to execute.
- In the solution, the customer has the record of the number of customers waiting in the waiting room if the number of customers is equal to the number of chairs in the waiting room then the upcoming customer leaves the barbershop. When the barber shows up in the morning, he executes the procedure barber, causing him to block on the semaphore customers because it is initially 0.
- Then the barber goes to sleep until the first customer comes up. When a customer arrives, he executes customer procedure the customer acquires the mutex for entering the critical region, if another customer enters thereafter, the second one will not be able to anything until the first one has released the mutex.
- The customer then checks the chairs in the waiting room if waiting customers are less then the number of chairs then he sits otherwise he leaves and releases the mutex. If the chair is available then customer sits in the waiting room and increments the variable waiting value and also increases the customer's semaphore this wakes up the barber if he is sleeping. At this point, customer and barber are both awake and the barber is ready to give that person a haircut. When the haircut is over, the customer exits the procedure and if there are no customers in waiting room barber sleeps.



Algorithm for Sleeping Barber problem:

```
Semaphore Customers = 0;
Semaphore Barber = 0;
Mutex Seats = 1;
int FreeSeats = N;

Barber {
        while(true) {
                        /* waits for a customer (sleeps). */
                        down(Customers);
                        /* mutex to protect the number of available seats.*/
                        down(Seats);
                        /* a chair gets free.*/
                        FreeSeats++;
                        /* bring customer for haircut.*/
                        up(Barber);
                        /* release the mutex on the chair.*/
                        up(Seats);
                        /* barber is cutting hair.*/
        }
}
```

```
Customer {
        while(true) {
                        /* protects seats so only 1 customer tries to sit
                        in a chair if that's the case.*/
                        down(Seats); //This line should not be here.
                        if(FreeSeats > 0) {

                                /* sitting down.*/
                                FreeSeats--;

                                /* notify the barber. */
                                up(Customers);

                                /* release the lock */
                                up(Seats);

                                /* wait in the waiting room if barber is busy. */
                                down(Barber);
                                // customer is having hair cut
                        } else {
                                /* release the lock */
                                up(Seats);
                                // customer leaves
                        }
        }
}
```

2.11. Serializability in Operating Systems

- In the context of operating systems, serializability is a crucial concept in ensuring the correctness of concurrent execution of processes or threads that access shared resources.
- It guarantees that the final state of the system is equivalent to some serial execution of the processes.

1. Looking Protocols:

- **Two-Phase Locking (2PL):**
  - A process acquires locks before accessing shared resources and releases them only after it's done.
  - **Strict 2PL:** A process must acquire all locks before releasing any.
  - **Conservative 2PL:** A process acquires all locks it needs before starting execution.
  - **Optimistic 2PL:** A process executes without acquiring locks, and at commit time, it checks for conflicts.

2. Timestamp-Based Protocols:

- **Basic Timestamp Ordering (BTO):**
  - Each process is assigned a unique timestamp.
  - A process can read a data item only if its timestamp is greater than the timestamp of the last writer.
  - A process can write a data item only if its timestamp is greater than the timestamp of all previous readers and writers.
- **Timestamp Ordering Protocol (TO):**
  - Similar to BTO, but with additional mechanisms to handle write-write conflicts.
  - If a process T1 writes a data item and later a process T2 tries to write the same item with a smaller timestamp, T2 is aborted.

Implementation Challenges in OS:

- **Overheads:** Both looking and timestamp-based protocols incur overhead in terms of lock acquisition, release, and timestamp management.
- **Deadlocks:** Strict 2PL can lead to deadlocks if not implemented carefully.
- **Starvation:** A process may starve if it's continuously denied access to shared resources.
- **Live lock:** Processes may continuously retry operations without making progress.

Mitigation Techniques:

- **Timeout Mechanisms:** Processes can set timeouts on lock acquisitions to prevent indefinite waiting.
- **Priority-Based Scheduling:** Prioritizing processes can help prevent starvation.
- **Conflict Detection and Resolution:** Efficient algorithms can be used to detect and resolve conflicts.
- **Hybrid Approaches:** Combining different protocols can improve performance and reliability.

2.12. Avoiding Locks: Read-Copy-Update

- RCU is a synchronization mechanism used in operating systems to avoid traditional locking, allowing readers to access shared data concurrently without blocking each other or the writers.
- This technique is particularly popular in the Linux kernel, where it is used to improve performance in read-heavy workloads.

RCU is based on three main principles:

1. **Readers Never Block**: Readers can access shared data concurrently without acquiring locks, which makes reading operations very fast.
2. **Writers Create Copies**: When an update is needed, writers do not modify the shared data in place. Instead, they create a copy, apply changes to it, and then update the reference to point to the modified copy.
3. **Deferred Reclamation**: Old data that is no longer in use is not immediately deleted. Instead, it is marked for deferred reclamation, which means it is only removed once all ongoing readers are guaranteed to have finished accessing it.

RCU Works:

1. **Reading Phase**:
   - When a reader accesses data, it does so without acquiring a lock.
   - RCU provides functions (like rcu_read_lock() and rcu_read_unlock() in the Linux kernel) that mark the beginning and end of a read-side critical section, ensuring that the data structure remains stable during this period.
   - These "read locks" are not real locks; they simply notify the RCU mechanism of the reader's presence, allowing concurrent access with minimal overhead.
2. **Updating Phase**:
   - Writers make a copy of the data structure or data item that needs modification.
   - They apply the modifications to this copy, ensuring that the original version remains unchanged for ongoing readers.
   - After updating, the writer atomically swaps the reference to point to the new, updated copy. The old version is still accessible to readers who started their access before the swap.
3. **Grace Period and Reclamation**:
   - RCU introduces a "grace period" before reclaiming memory associated with old data.
   - The OS waits for all ongoing readers to complete before reclaiming the old data, ensuring that no readers are disrupted by the update.
   - Once the grace period has passed (i.e., all readers that started before the update are finished), the old data is safely freed.

2.13. Deadlocks:

Deadlock is a situation in computing where two or more processes are unable to proceed because each is waiting for the other to release resources.

Key concepts include mutual exclusion, resource holding, circular wait, and no Preemption.

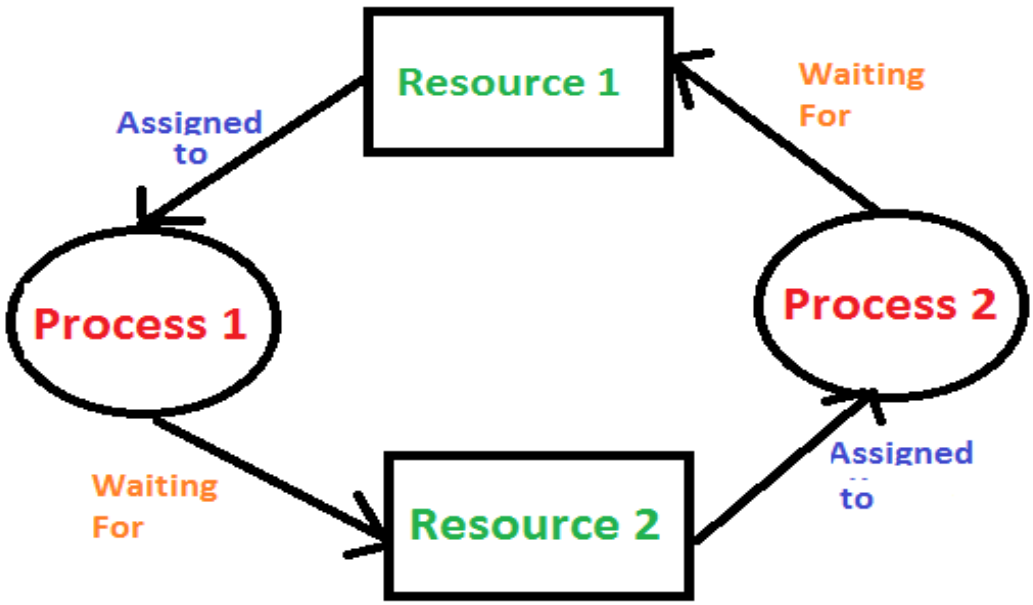
Consider an example when two trains are coming toward each other on the same track and there is only one track, none of the trains can move once they are in front of each other. This is a practical example of deadlock.

How Does Deadlock occur in the Operating System?

Before going into detail about how deadlock occurs in the Operating System, let's first discuss how the Operating System uses the resources present. A process in an operating system uses resources in the following way.

- Requests a resource
- Use the resource
- Releases the resource

A situation occurs in operating systems when there are two or more processes that hold some resources and wait for resources held by other(s). For example, in the below diagram, Process 1 is holding Resource 1 and waiting for resource 2 which is acquired by process 2, and process 2 is waiting for resource 1.



Necessary Conditions for Deadlock in OS

Deadlock can arise if the following four conditions hold simultaneously (Necessary Conditions)

- **Mutual Exclusion:** Two or more resources are non-shareable (Only one process can use at a time).
- **Hold and Wait:** A process is holding at least one resource and waiting for resources.
- **No Preemption:** A resource cannot be taken from a process unless the process releases the resource.
- **Circular Wait:** A set of processes waiting for each other in circular form.

Strategies for handling Deadlock

1. Deadlock Ignorance

Deadlock Ignorance is the most widely used approach among all the mechanism. This is being used by many operating systems mainly for end user uses. In this approach, the Operating system assumes that deadlock never occurs. It simply ignores deadlock. This approach is best suitable for a single end user system where User uses the system only for browsing and all other normal stuff. There is always a tradeoff between Correctness and performance. The operating systems like Windows and Linux mainly focus upon performance. However, the performance of the system decreases if it uses deadlock handling mechanism all the time if deadlock happens 1 out of 100 times then it is completely unnecessary to use the deadlock handling mechanism all the time.

In these types of systems, the user has to simply restart the computer in the case of deadlock. Windows and Linux are mainly using this approach.

2. Deadlock prevention

Deadlock happens only when Mutual Exclusion, hold and wait, No preemption and circular wait holds simultaneously. If it is possible to violate one of the four conditions at any time then the deadlock can never occur in the system.

The idea behind the approach is very simple that we have to fail one of the four conditions but there can be a big argument on its physical implementation in the system.

3. **Deadlock avoidance**

In deadlock avoidance, the operating system checks whether the system is in safe state or in unsafe state at every step which the operating system performs. The process continues until the system is in safe state. Once the system moves to unsafe state, the OS has to backtrack one step.

In simple words, The OS reviews each allocation so that the allocation doesn't cause the deadlock in the system.

4. **Deadlock detection and recovery**

This approach let the processes fall in deadlock and then periodically check whether deadlock occur in the system or not. If it occurs then it applies some of the recovery methods to the system to get rid of deadlock.

Resource Allocation Graph (RAG)

A resource allocation graphs shows which resource is held by which process and which process is waiting for a resource of a specific kind. It is amazing and straight – forward tool to outline how interacting processes can deadlock. Therefore, resource allocation graph describe what the condition of the system as far as process and resources are concern like what number of resources are allocated and what is the request of each process. Everything can be represented in terms of graph. One of the benefit of having a graph is, sometimes it is conceivable to see a deadlock straight forward by utilizing RAG and however you probably won't realize that by taking a glance at the table.

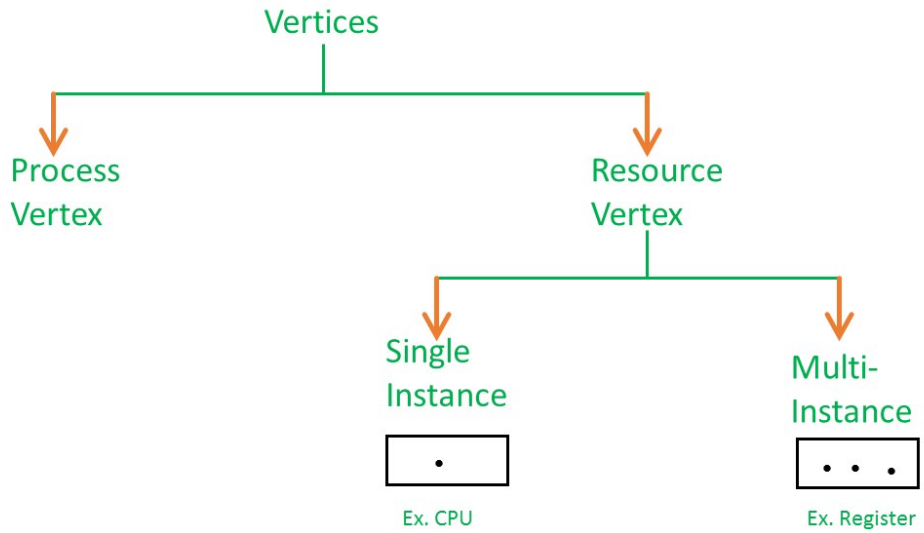
Types of Vertices in RAG

So RAG also contains vertices and edges. In RAG vertices are two types

1. Process Vertex: Every process will be represented as a process vertex. Generally, the process will be represented with a circle.

2. Resource Vertex: Every resource will be represented as a resource vertex. It is also two types:

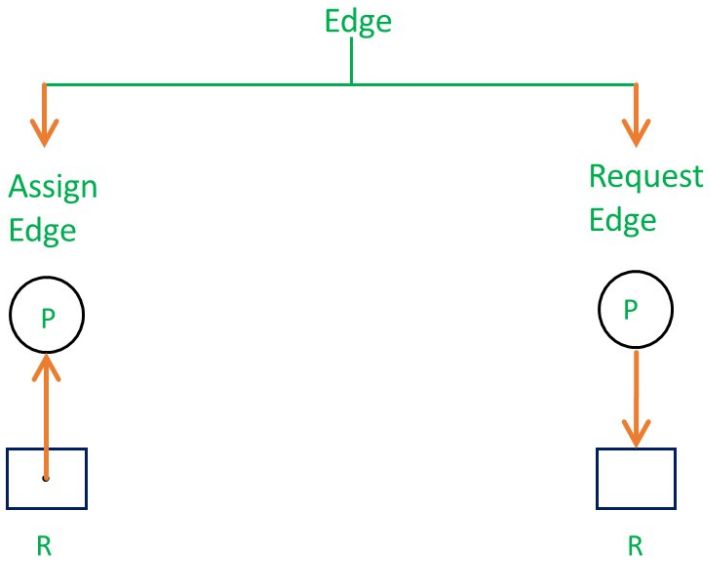
- **Single instance type resource:** It represents as a box, inside the box, there will be one dot.So the number of dots indicate how many instances are present of each resource type.
- **Multi-resource instance type resource:** It also represents as a box, inside the box, there will be many dots present.



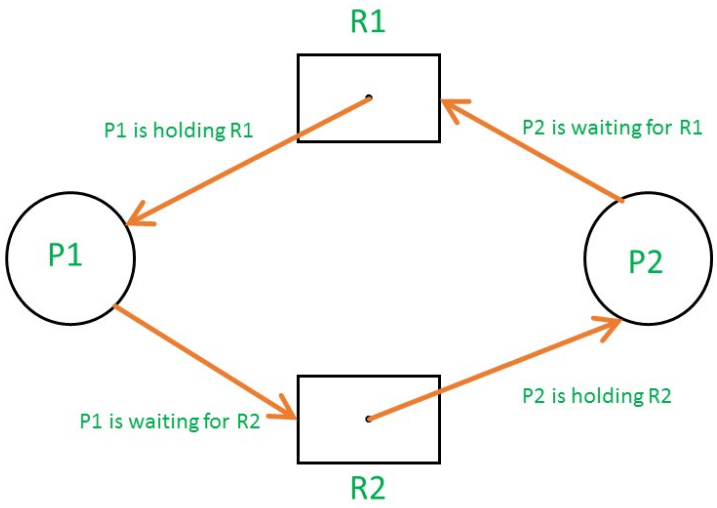
How many Types of Edges are there in RAG?

Now coming to the edges of RAG.There are two types of edges in RAG –

- **Assign Edge:** If you already assign a resource to a process then it is called Assign edge.
- **Request Edge:** It means in future the process might want some resource to complete the execution, that is called request edge.

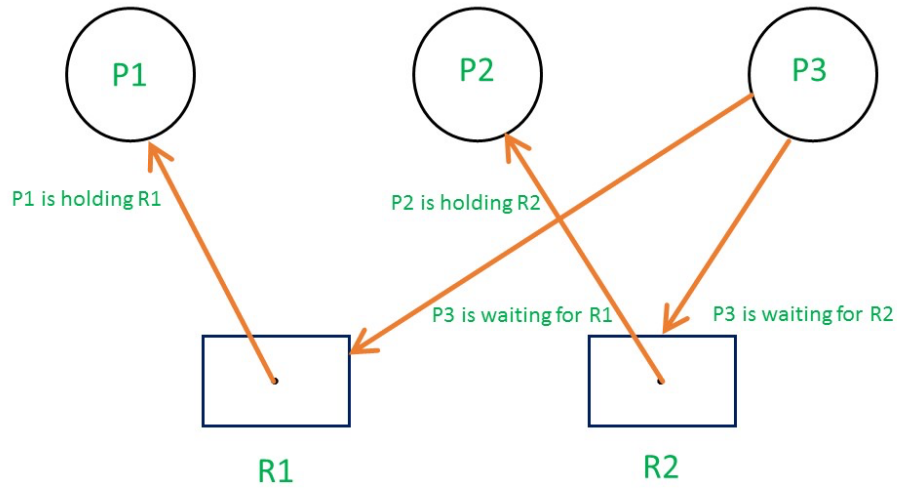


Example 1 (Single instances RAG)



SINGLE INSTANCE RESOURCE TYPE WITH DEADLOCK

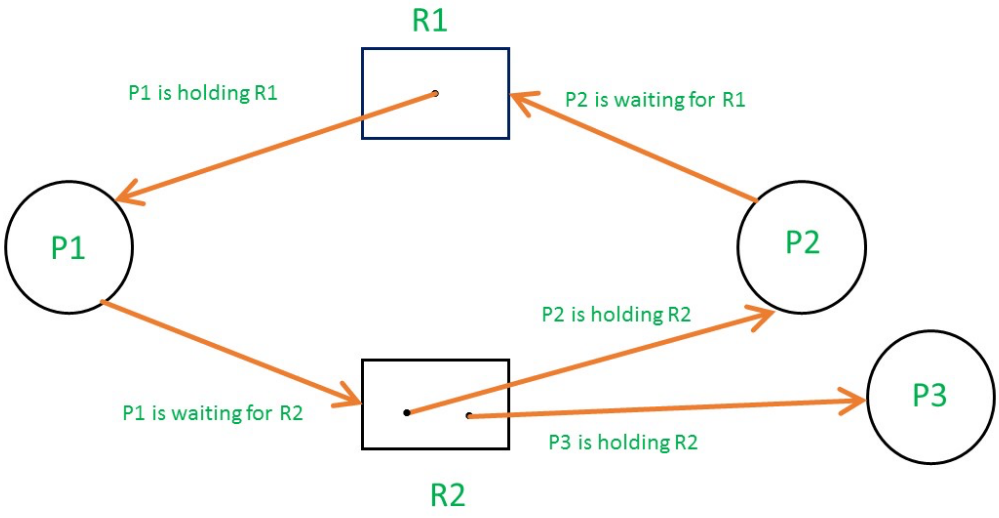
If there is a cycle in the Resource Allocation Graph and each resource in the cycle provides only one instance, then the processes will be in deadlock. For example, if process P1 holds resource R1, process P2 holds resource R2 and process P1 is waiting for R2 and process P2 is waiting for R1, then process P1 and process P2 will be in deadlock.



SINGLE INSTANCE RESOURCE TYPE WITHOUT DEADLOCK

Here's another example, that shows Processes P1 and P2 acquiring resources R1 and R2 while process P3 is waiting to acquire both resources. In this example, there is no deadlock because there is no circular dependency. So cycle in single-instance resource type is the sufficient condition for deadlock.

Example 2 (Multi-instances RAG)



MULTI INSTANCES WITHOUT DEADLOCK

From the above example, it is not possible to say the RAG is in a safe state or in an unsafe state.So to see the state of this RAG, let's construct the allocation matrix and request matrix.

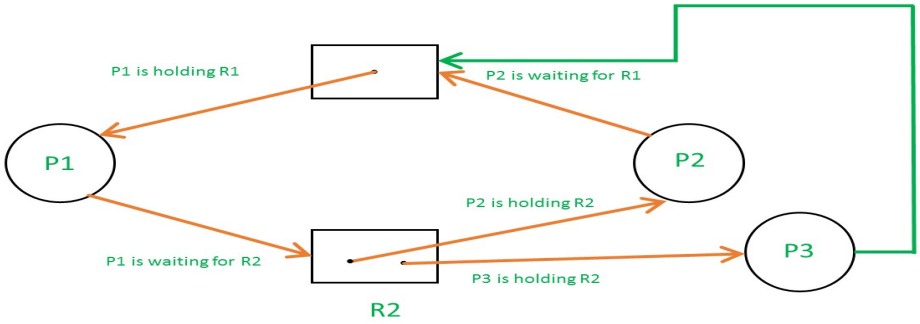
Process	Allocation		Request	
	Resource		Resource	
	R1	R2	R1	R2
P1	1	0	0	1
P2	0	1	1	0
P3	0	1	0	0

- The total number of processes are three; P1, P2 & P3 and the total number of resources are two; R1 & R2.

- **Allocation matrix –**
- For constructing the allocation matrix, just go to the resources and see to which process it is allocated.
- R1 is allocated to P1, therefore write 1 in allocation matrix and similarly, R2 is allocated to P2 as well as P3 and for the remaining element just write 0.
- **Request matrix –**
- In order to find out the request matrix, you have to go to the process and see the outgoing edges.
- P1 is requesting resource R2, so write 1 in the matrix and similarly, P2 requesting R1 and for the remaining element write 0.
- So now available resource is = (0, 0).
- **Checking deadlock (safe or not) –**

```
Available = 0  0   (As P3 does not require any extra resource to complete the execution and after
   P3       0  1    completion P3 release its own resource)
_____________________________
New Available = 0  1  (As using new available resource we can satisfy the requirment of process P1
   P1           1  0   and P1 also release its prevous resource)
_____________________________
New Available = 1   1 (Now easily we can satisfy the requirement of process P2)
   P2           0  1
_____________________________
New Available =  1   2
```

-
- So, there is no deadlock in this RAG.Even though there is a cycle, still there is no deadlock.Therefore in multi-instance resource cycle is not sufficient condition for deadlock.



MULTI INSTANCES WITH DEADLOCK

-

- Above example is the same as the previous example except that, the process P3 requesting for resource R1. So the table becomes as shown in below.

Process	Allocation		Request	
	Resource		Resource	
	R1	R2	R1	R2
P1	1	0	0	1
P2	0	1	1	0
P3	0	1	1	0

- So,the Available resource is = (0, 0), but requirement are (0, 1), (1, 0) and (1, 0).So you can't fulfill any one requirement.Therefore, it is in deadlock. Therefore, every cycle in a multi-instance resource type graph is not a deadlock, if there has to be a deadlock, there has to be a cycle.So, in case of RAG with multi-instance resource type, the cycle is a necessary condition for deadlock, but not sufficient.

Banker's Algorithm

The Banker's Algorithm is a resource allocation and deadlock avoidance algorithm that tests for safety by simulating the allocation for the predetermined maximum possible amounts of all resources, then makes an "s-state" check to test for possible activities, before deciding whether allocation should be allowed to continue.

Why Banker's Algorithm is Named So?

The banker's algorithm is named so because it is used in the banking system to check whether a loan can be sanctioned to a person or not. Suppose there are n number of account holders in a bank and the total sum of their money is S. Let us assume that the bank has a certain amount of money Y. If a person applies for a loan then the bank first subtracts the loan amount from the total money that the bank has (Y) and if the remaining amount is greater than S then only the loan is sanctioned. It is done because if all the account holders come to withdraw their money then the bank can easily do it.

It also helps the OS to successfully share the resources between all the processes. It is called the banker's algorithm because bankers need a similar algorithm, they admit loans that collectively exceed the bank's funds and then release each borrower's loan in installments. The banker's algorithm uses the notation of a safe allocation state to ensure that granting a resource request cannot lead to a deadlock either immediately or in the future.
In other words, the bank would never allocate its money in such a way that it can no longer satisfy the needs of all its customers. The bank would try to be in a safe state always.

Example: Considering a system with five processes P0 through P4 and three resources of type A, B, C. Resource type A has 10 instances, B has 5 instances and type C has 7 instances. Suppose at time t0 following snapshot of the system has been taken:

Process	Allocation	Max	Available
	A B C	A B C	A B C
P_0	0 1 0	7 5 3	3 3 2
P_1	2 0 0	3 2 2	
P_2	3 0 2	9 0 2	
P_3	2 1 1	2 2 2	
P_4	0 0 2	4 3 3	

Q.1 What will be the content of the Need matrix?

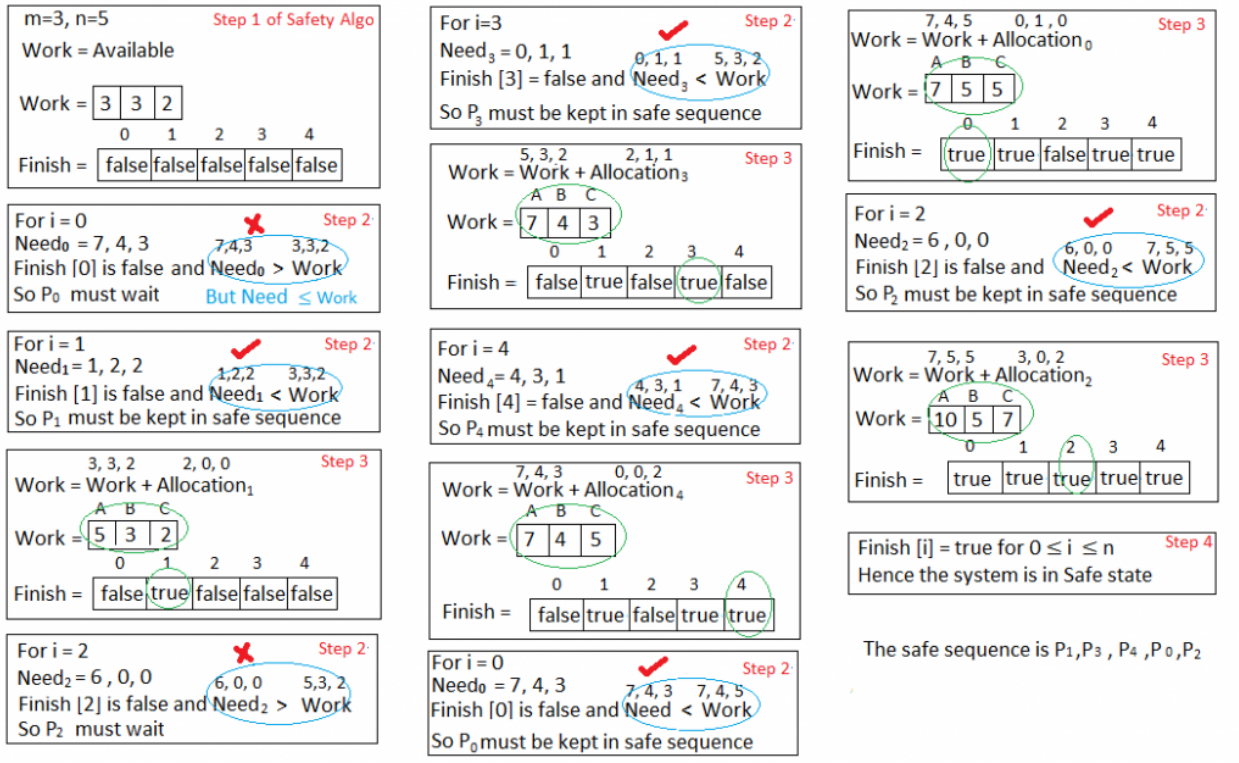
Need [i, j] = Max [i, j] – Allocation [i, j]

Process	Need		
	A	B	C
P_0	7	4	3
P_1	1	2	2
P_2	6	0	0
P_3	0	1	1
P_4	4	3	1

So, the content of Need Matrix is:

Q.2 Is the system in a safe state? If Yes, then what is the safe sequence?

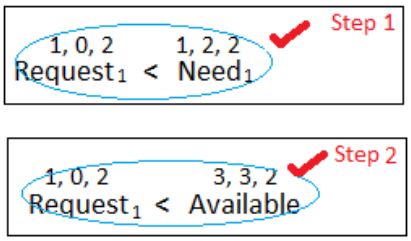
Applying the Safety algorithm on the given system,



Q.3 What will happen if process P1 requests one additional instance of resource type A and two instances of resource type C?

A B C
$Request_1$ = 1, 0, 2

To decide whether the request is granted we use Resource Request algorithm



Step 3

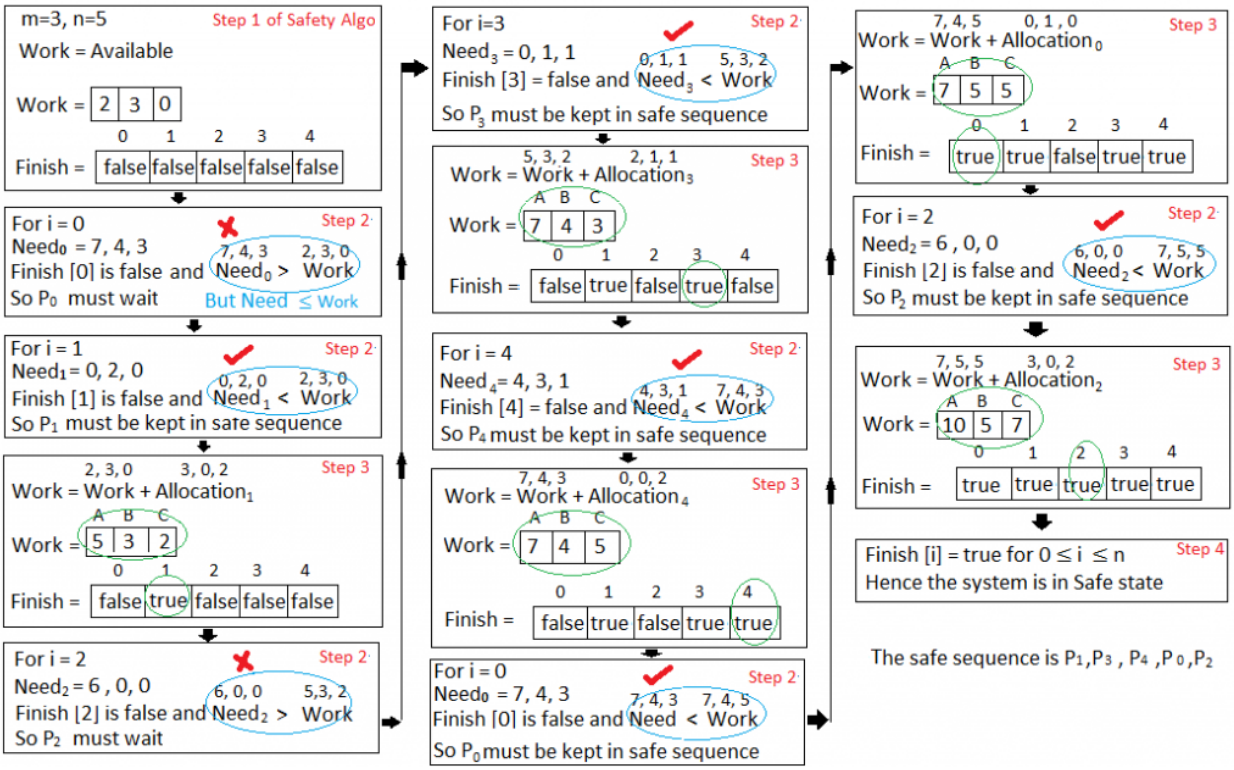
Available = Available – $Request_1$

$Allocation_1$ = $Allocation_1$ + $Request_1$

$Need_1$ = $Need_1$ - $Request_1$

Process	Allocation	Need	Available
	A B C	A B C	A B C
P_0	0 1 0	7 4 3	2 3 0
P_1	3 0 2	0 2 0	
P_2	3 0 2	6 0 0	
P_3	2 1 1	0 1 1	
P_4	0 0 2	4 3 1	

We must determine whether this new system state is safe. To do so, we again execute Safety algorithm on the above data structures.



Hence the new system state is safe, so we can immediately grant the request for process P1 .

2.14. Process scheduling Algorithm:

CPU scheduling is a process used by the operating system to decide which task or program gets to use the CPU at a particular time. Since many programs can run at the same time, the OS needs to manage the CPU's time so that every program gets a proper chance to run. The purpose of CPU Scheduling is to make the system more efficient and faster.

What is the Need for a CPU Scheduling Algorithm?

CPU scheduling is the process of deciding which process will own the CPU to use while another process is suspended. The main function of CPU scheduling is to ensure that whenever the CPU remains idle, the OS has at least selected one of the processes available in the ready-to-use line.

In multiprogramming ,if the long-term scheduler selects multiple I/O binding processes then most of the time, the CPU remains idle. The function of an effective program is to improve resource utilization.

If most operating systems change their status from performance to waiting then there may always be a chance of failure in the system. So in order to minimize this excess, the OS needs to schedule tasks in order to make full use of the CPU and avoid the possibility of deadlock.

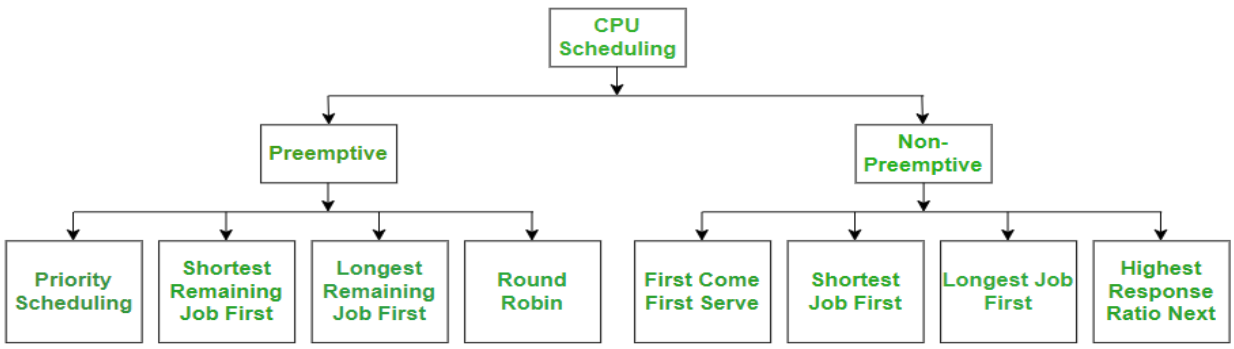
Terminologies Used in CPU Scheduling

- **Arrival Time:** The time at which the process arrives in the ready queue.
- **Completion Time:** The time at which the process completes its execution.
- **Burst Time:** Time required by a process for CPU execution.
- **Turn Around Time:** Time Difference between completion time and arrival time.

Turn Around Time = Completion Time – Arrival Time

- **Waiting Time(W.T):** Time Difference between turn around time and burst time.

Waiting Time = Turn Around Time – Burst Time



1. First Come First Serve

FCFS considered to be the simplest of all operating system scheduling algorithms. First come first serve scheduling algorithm states that the process that requests the CPU first is allocated the CPU first and is implemented by using FIFO queue.

Characteristics of FCFS

- FCFS supports non-preemptive and preemptive CPU scheduling algorithms.
- Tasks are always executed on a First-come, First-serve concept.
- FCFS is easy to implement and use.
- This algorithm is not much efficient in performance, and the wait time is quite high.

Advantages of FCFS

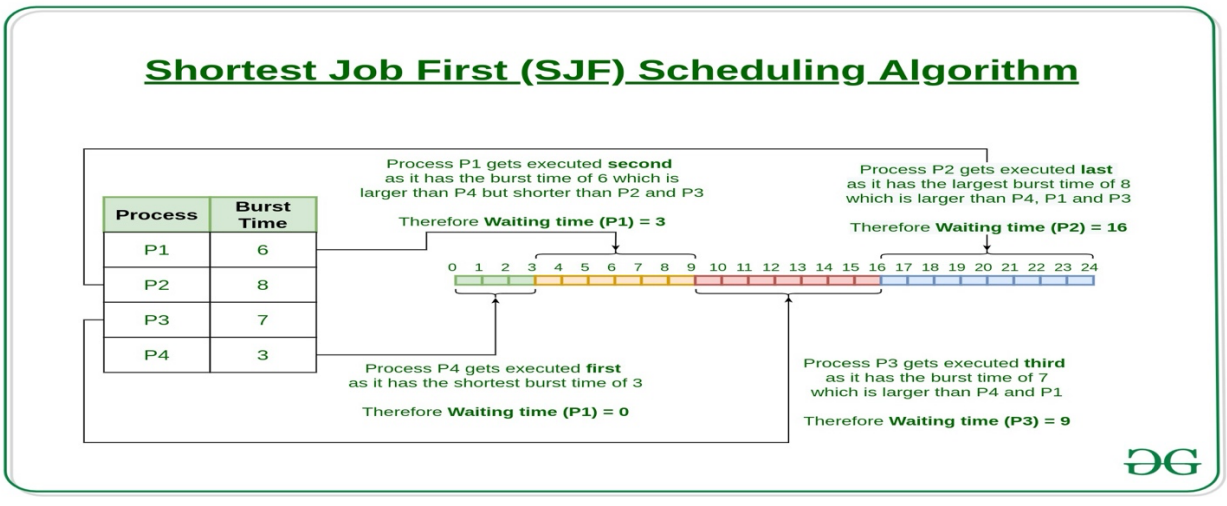
- Easy to implement
- First come, first serve method

Disadvantages of FCFS

- FCFS suffers from **Convoy effect**.
- The average waiting time is much higher than the other algorithms.
- FCFS is very simple and easy to implement and hence not much efficient.

2. Shortest Job First(SJF)

Shortest job first (SJF) is a scheduling process that selects the waiting process with the smallest execution time to execute next. This scheduling method may or may not be preemptive. Significantly reduces the average waiting time for other processes waiting to be executed. The full form of SJF is Shortest Job First.



Characteristics of SJF

- Shortest Job first has the advantage of having a minimum average waiting time among all operating system scheduling algorithms.
- It is associated with each task as a unit of time to complete.
- It may cause starvation if shorter processes keep coming. This problem can be solved using the concept of ageing.

Advantages of SJF

- As SJF reduces the average waiting time thus, it is better than the first come first serve scheduling algorithm.
- SJF is generally used for long term scheduling

Disadvantages of SJF

- One of the demerit SJF has is starvation.
- Many times it becomes complicated to predict the length of the upcoming CPU request

3. Longest Job First(LJF)

Longest Job First(LJF) scheduling process is just opposite of shortest job first (SJF), as the name suggests this algorithm is based upon the fact that the process with the largest burst time is processed first. Longest Job First is non-preemptive in nature.

Characteristics of LJF

- Among all the processes waiting in a waiting queue, CPU is always assigned to the process having largest burst time.
- If two processes have the same burst time then the tie is broken using FCFS i.e. the process that arrived first is processed first.
- LJF CPU Scheduling can be of both preemptive and non-preemptive types.

Advantages of LJF

- No other task can schedule until the longest job or process executes completely.
- All the jobs or processes finish at the same time approximately.

Disadvantages of LJF

- Generally, the LJF algorithm gives a very high average waiting time and average turn-around time for a given set of processes.
- This may lead to convoy effect.

4. Priority Scheduling

Preemptive Priority CPU Scheduling Algorithm is a pre-emptive method of CPU scheduling algorithm that works **based on the priority** of a process. In this algorithm, the editor sets the functions to be as important, meaning that the most important process must be done first. In the case of any conflict, that is, where there is more than one process with equal value, then the most important CPU planning algorithm works based on the FCFS (First Come First Serve) algorithm.

Characteristics of Priority Scheduling

- Schedules tasks based on priority.
- When the higher priority work arrives and a task with less priority is executing, the higher priority process will take the place of the less priority process and
- That is suspended until the execution is complete.
- Lower is the number assigned, higher is the priority level of a process.

Advantages of Priority Scheduling

- The average waiting time is less than FCFS
- Less complex

Disadvantages of Priority Scheduling

- One of the most common demerits of the Preemptive priority CPU scheduling algorithm is the starvation Problem. This is the problem in which a process has to wait for a longer amount of time to get scheduled into the CPU. This condition is called the starvation problem.

5. Round Robin

Round Robin is a CPU scheduling algorithm where each process is cyclically assigned a fixed time slot. It is the preemptive version of First come First Serve CPU Scheduling algorithm. Round Robin CPU Algorithm generally focuses on Time Sharing technique.

Characteristics of Round robin

- It's simple, easy to use, and starvation-free as all processes get the balanced CPU allocation.
- One of the most widely used methods in CPU scheduling as a core.
- It is considered preemptive as the processes are given to the CPU for a very limited time.

Advantages of Round robin

- Round robin seems to be fair as every process gets an equal share of CPU.
- The newly created process is added to the end of the ready queue.

6. Shortest Remaining Time First (SRTF)

Shortest remaining time first is the preemptive version of the shortest job first which we have discussed earlier where the processor is allocated to the job closest to completion. In SRTF the process with the smallest amount of time remaining until completion is selected to execute.

Characteristics of SRTF

- SRTF algorithm makes the processing of the jobs faster than SJF algorithm, given its overhead charges are not counted.
- The context switch is done a lot more times in SRTF than in SJF and consumes the CPU's valuable time for processing. This adds up to its processing time and diminishes its advantage of fast processing.

Advantages of SRTF

- In SRTF the short processes are handled very fast.
- The system also requires very little overhead since it only makes a decision when a process completes, or a new process is added.

Disadvantages of SRTF

- Like the shortest job first, it also has the potential for process starvation.
- Long processes may be held off indefinitely if short processes are continually added.

7. Longest Remaining Time First (LRTF)

The longest remaining time first is a preemptive version of the longest job first scheduling algorithm. This scheduling algorithm is used by the operating system to program incoming processes for use in a systematic way. This algorithm schedules those processes first which have the longest processing time remaining for completion.

Characteristics of LRTF

- Among all the processes waiting in a waiting queue, the CPU is always assigned to the process having the largest burst time.

- If two processes have the same burst time then the tie is broken using FCFS i.e. the process that arrived first is processed first.
- LRTF CPU Scheduling can be of both preemptive and non-preemptive.
- No other process can execute until the longest task executes completely.
- All the jobs or processes finish at the same time approximately.

Advantages of LRTF

- Maximizes Throughput for Long Processes.
- Reduces Context Switching.
- Simplicity in Implementation.

Disadvantages of LRTF

- This algorithm gives a very high average waiting time and average turn-around time for a given set of processes.
- This may lead to a convoy effect.

8. Highest Response Ratio Next(HRRN)

Highest Response Ratio Next is a non-preemptive CPU Scheduling algorithm and it is considered as one of the most optimal scheduling algorithms. The name itself states that we need to find the response ratio of all available processes and select the one with the highest Response Ratio. A process once selected will run till completion.

Characteristics of HRRN

- The **criteria** for HRRN is **Response Ratio,** and the **mode** is **Non-Preemptive.**
- HRRN is considered as the modification of SJF to reduce the problem of Starvation.
- In comparison with SJF, during the HRRN scheduling algorithm, the CPU is allotted to the next process which has the **highest response ratio** and not to the process having less burst time.

Response Ratio = (W + S)/S

*Here, **W** is the waiting time of the process so far and **S** is the Burst time of the process.*

Advantages of HRRN

- HRRN Scheduling algorithm generally gives better performance than the shortest job first Scheduling.
- There is a reduction in waiting time for longer jobs and it encourages shorter jobs.

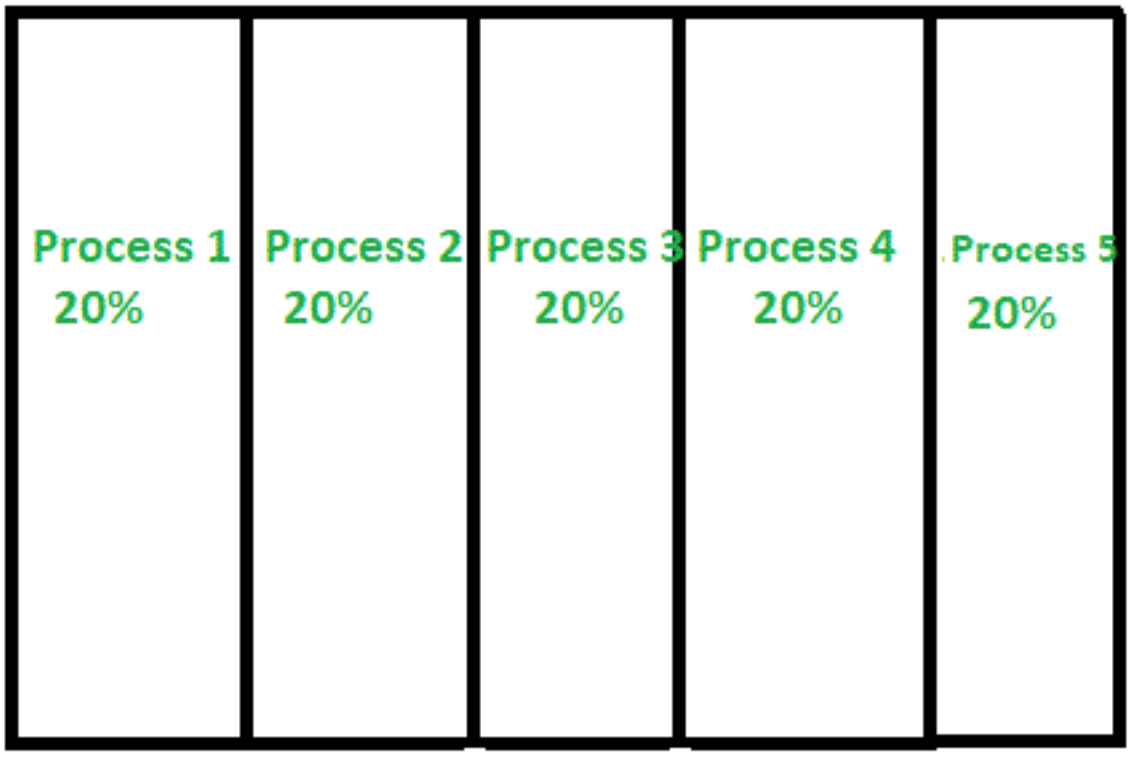
Disadvantages of HRRN

- The implementation of HRRN scheduling is not possible as it is not possible to know the burst time of every job in advance.
- In this scheduling, there may occur an overload on the CPU.

For Numericals related to these Scheduling Algorithms please have a look at the class lecturer Slides.

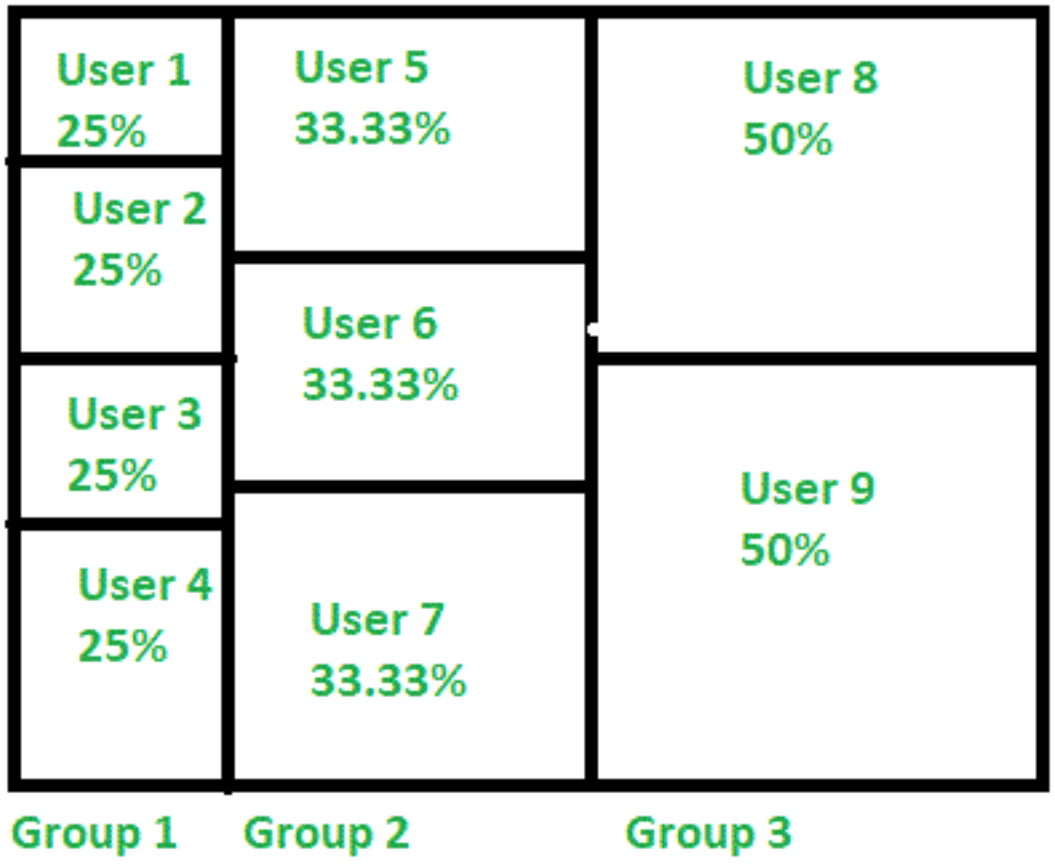
2.15. Other Scheduling Algorithms:

1. **Fair-share scheduling**:
    - It was first designed by Judy Kay and Piers Lauder at Sydney University in the 1980s.
    - It is a scheduling algorithm for computer operating systems that dynamically distributes the time quanta "equally" to its users.
    - Specificity of Fair-share scheduling:
    - This algorithm equally distributes the processor time to its users, for instance, there are 5 users (A, B, C, D, E) each of them are simultaneously executing a process, the scheduler divides the CPU periods such that all the users get the same share of the CPU cycles (100%/5) that is 20%. Even though, a user moves onto the second while the other at the first, the algorithm is so specific that it ensures that this user is attributed with only 10% for the second process making it a total of 20%.



1 CPU time slices are equally by 5 processses in Fair-share scheduling operating system.

The scheduler logically divides an equal amount even though, another layer of partition is added, for example, if there were 3 groups present with different number of people in each group, the algorithm would still divide the same time for those groups, 100%/3= 33.33%, this 33.33% would be shared equally in the respective group depending on the number of users present in the group.



2. **Guaranteed Scheduling:**

- A scheduling algorithm used in multitasking operating systems that guarantees fairness by monitoring the amount of CPU time spent by each user and allocating resources accordingly is guaranteed scheduling.
- It makes real promises to the users about performance.
- If there are n users logged in while we are working, we will receive about 1 /n of the CPU power. Similarly, on a single-user system with n processes running, all things being equal, each one should get 1/n of the CPU cycles.
- To make good on this promise, the system must keep track of how much CPU each process has had since its creation.
- CPU Time entitled = (Time Since Creation)/n
- Then compute the ratio of Actual CPU time consumed to the CPU time entitled.
- A ratio of 0.5 means that a process has only had half of what it should have had, and a ratio of 2.0 means that a process has had twice as much as it was entitled to.
- The algorithm is then to run the process with the lowest ratio until its ratio has moved above its closest competitor

3. **Lottery Scheduling:**

- Lottery Scheduling is a probabilistic scheduling algorithm for processes in an operating system.
- Processes are each assigned some number of lottery tickets for various system resources such as CPU time and the scheduler draws a random ticket to select the next process.
- This technique can be used to approximate other scheduling algorithms, such as shortest job next and Fair-share scheduling.
- Lottery scheduling solves the problem of starvation. Giving each process at least one lottery ticket guarantees that it has non-zero probability of being selected at each scheduling operation. More important process can be given extra tickets to increase their odd of winning. If there are 100 tickets outstanding, & one process holds 20 of them it will have 20% chance of winning each lottery.
- In the long run it will get 20% of the CPU. A process holding a fraction f of the tickets will get about a fraction of the resource in questions.