UNIT-2

OPERATING SYSTEMS

UNIT II:

PART 1-Process Concept

PART 2-Multithreaded Programming

PART 3-Inter-process Communication

PART 1 - PROCESS

Process, Process Structure, Operations on processes, Inter-process Communication, Communication in client server systems.

PART 2 – MULTITHREADED PROGRAMMING

Multithreaded Programming: Multithreading models, Thread libraries, threading issues.

Process Scheduling: Basic concepts, Scheduling criteria, Scheduling algorithms, multiple processor scheduling, thread scheduling.

PART 3 – INTERPROCESS COMMUNICATION

Inter-process Communication: Race conditions, Critical Regions, Mutual exclusion with busy waiting, Sleep and wakeup, Semaphores, Mutexes, Monitors, Message passing. Barriers, Classical IPC Problems- Dining philosophers problem, Readers and writers problem.

PART-1 - PROCESS

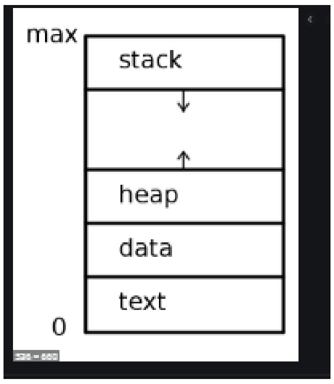
➢ **Process concepts**

Process : A process is a program in execution. A process is more than the program code, which is sometimes known as the text section. It also includes the current activity, as represented by the value of the program counter and the contents of the processor's registers. A process generally also includes the process stack, which contains temporary data (such as function parameters, return addresses, and local variables), and a data section, which contains global variables. A process may also include a heap, which is memory that is dynamically allocated during process run time.

➢ **Structure of a process**

We emphasize that a program by itself is not a process; a program is a passive entity, such as a file containing a list of instructions stored on disk (often called an executable file), whereas a process is an active entity, with a program counter specifying the next instruction to execute and a set of associated resources. A program becomes a process when an executable file is loaded into memory.

➢ **Process**



We emphasize that a program by itself is not a process; a program is a passive entity, such as a file containing a list of instructions stored on disk (often called an executable file), whereas a process is an active entity, with a program counter specifying the next instruction to execute and a set of associated resources. A program becomes a process when an executable file is loaded into memory.

Two common techniques for loading executable files are double-clicking an icon representing the executable file and entering the name of the executable file on the command line (as in prog. exe or a.out.)

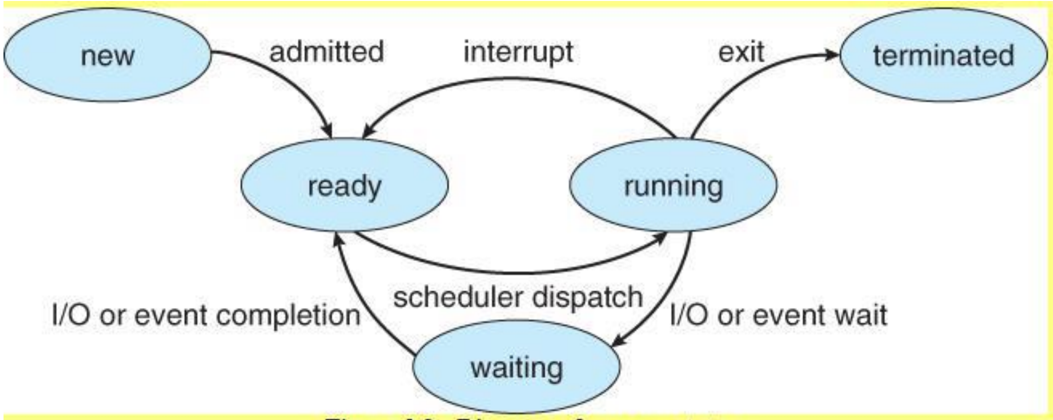


Figure 3.2 - Diagram of process state

➢ Process State

As a process executes, it changes **state.** The state of a process is defined in part by the current activity of that process. Each process may be in one of the following states:

• **New.** The process is being created.

• **Running.** Instructions are being executed.

• **Waiting.** The process is waiting for some event to occur (such as an I/O completion or reception of a signal).

• **Ready.** The process is waiting to be assigned to a processor.

• **Terminated.** The process has finished execution. These names are arbitrary, and they vary across operating systems. The states that they represent are fotind on all systems, however. Certain operating systems also more finely delineate process states. It is important to realize that only one process can be *running* on any processor at any instant.

➢ Process Control Block

Each process is represented in the operating system by a **process control block (PCB)**— also called a **task control block**.

Process state. The state may be new, ready, running, and waiting, halted, and so on.

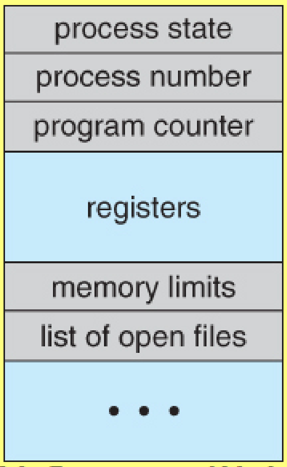


Figure 3.3 - Process control block (PCB)

- **Program counter-**The counter indicates the address of the next instruction to be executed for this process.

- CPU **registers-** The registers vary in number and type, depending on the computer architecture. They include accumulators, index registers, stack pointers, and general-purpose registers, plus any condition code information.
- **CPU-scheduling information-** This information includes a process priority, pointers to scheduling queues, and any other scheduling parameters.
- **Memory-management information-** This information may include such information as the value of the base and limit registers, the page tables, or the segment tables, depending on the memory system used by the operating system.
- **Accounting information-**This information includes the amount of CPU and real time used, time limits, account members, job or process numbers, and so on.
- **I/O status information-**This information includes the list of I/O devices allocated to the process, a list of open files, and so on.

❖ OPERATIONS ON PROCESSES

In an operating system, various operations are performed on processes to manage their execution and ensure the efficient utilization of system resources. There are many operations that can be performed on processes. Some of these are process creation, process preemption, process blocking, and process termination. These are given in detail as follows

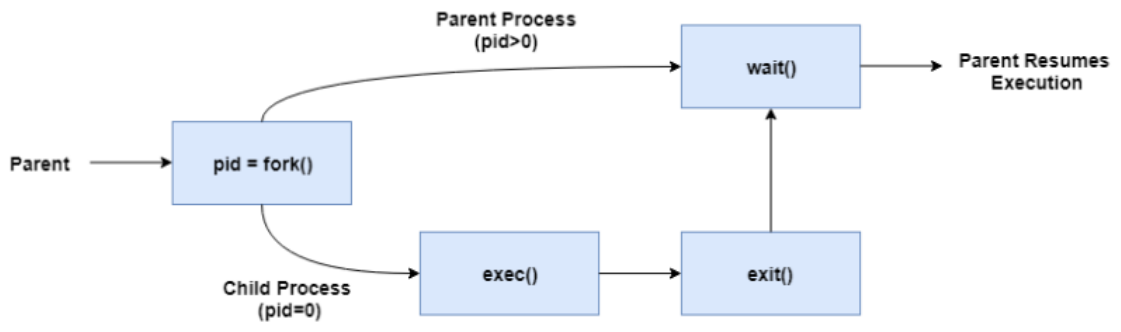
1. Process Creation

Processes need to be created in the system for different operations. This can be done by the following events

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

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.

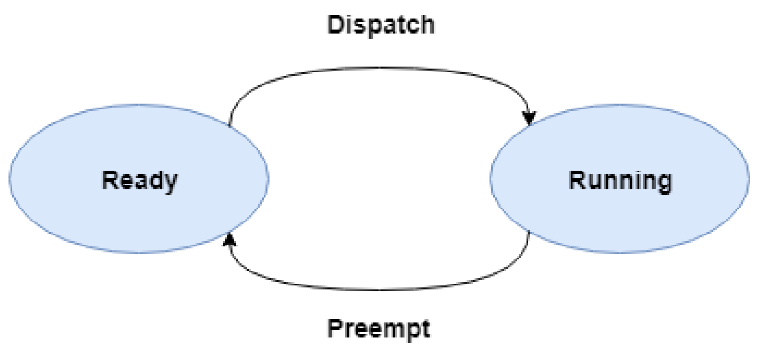
A diagram that demonstrates process creation using fork() is as follows –



Process Creation Using fork()

2. Process Preemption
An interrupt mechanism is used in preemption that suspends the process executing currently and the next process to execute is determined by the short-term scheduler. Preemption makes sure that all processes get some CPU time for execution.

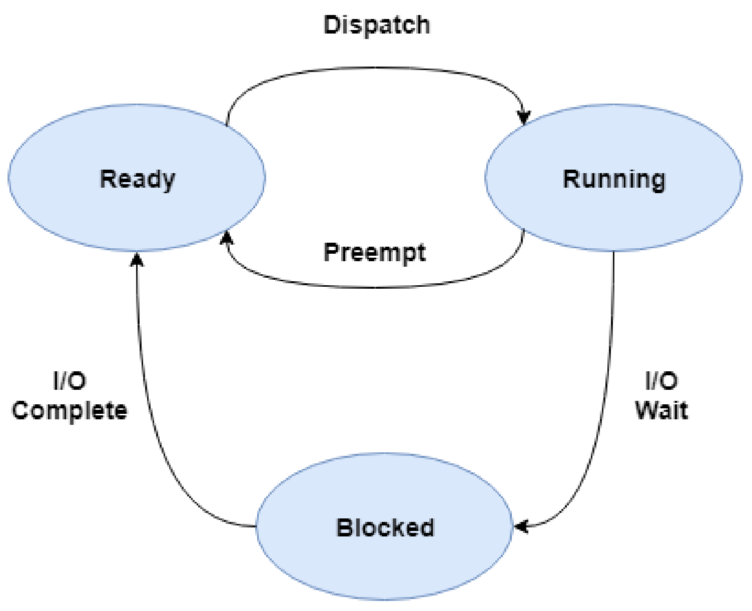
A diagram that demonstrates process preemption is as follows –



Process Preemption

3. Process Blocking
The process is blocked if it is waiting for some event to occur. This event may be I/O as the I/O events are executed in the main memory and don't require the processor. After the event is complete, the process again goes to the ready state.

A diagram that demonstrates process blocking is as follows –



Process Blocking

4. Process Termination
After the process has completed the execution of its last instruction, it is terminated. The resources held by a process are released after it is terminated.

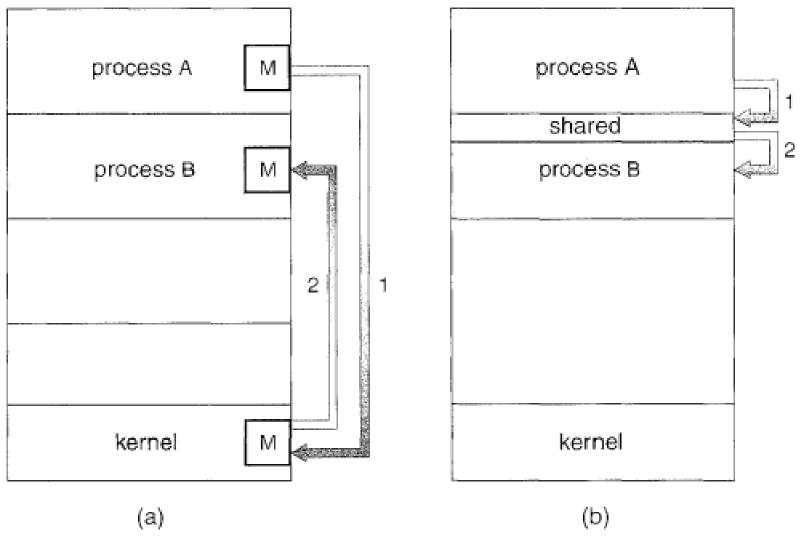
A child process can be terminated by its parent process if its task is no longer relevant. The child process sends its status information to the parent process before it terminates. Also, when a parent process is terminated, its child processes are terminated as well as the child processes cannot run if the parent processes are terminated.

❖ INTERPROCESS COMMUNICATION

Processes executing concurrently in the operating system may be either independent processes or cooperating processes. A process is independent if it cannot affect or be affected by the other processes executing in the system. Any process that does not share data with any other process is independent. A process is cooperating if it can affect or be affected by the other processes executing in the system. Clearly, any process that shares data with other processes is a cooperating process.

There are several reasons for providing an environment that allows process cooperation:
Information sharing. Since several users may be interested in the same piece of information (for instance, a shared file), we must provide an environment to allow concurrent access to such information.
Computation speedup. If we want a particular task to run faster, we must break it into subtasks, each of which will be executing in parallel with the others. Notice that such a speedup can be achieved only if the computer has multiple processing elements (such as CPUs or I/O channels).
Modularity. We may want to construct the system in a modular fashion, dividing the system functions into separate processes or threads.
Convenience. Even an individual user may work on many tasks at the same time. For instance, a user may be editing, printing, and compiling in parallel.

Cooperating processes require an interprocess communication (IPC) mechanism that will allow them to exchange data and information. There are two fundamental models of interprocess communication:
(1) shared memory and
(2) message passing.
In the shared-memory model, a region of memory that is shared by cooperating processes is established. Processes can then exchange information by reading and writing data to the shared region. In the message passing model, communication takes place by means of messages exchanged between the cooperating processes. The two communications models are contrasted in Figure.



1. **Shared Memory:**

Shared memory is an IPC mechanism that involves establishing a region of memory that multiple cooperating processes can access. These processes can read and write data to this shared memory region, allowing them to exchange information. Here's how shared memory works:

- **Memory Space Sharing:** Processes share a common memory area, referred to as shared memory space. This memory is allocated and managed by the operating system.
- **Data Exchange:** Processes can communicate by reading from and writing to the shared memory region. Changes made by one process can be immediately visible to other processes that access the same region.
- **Advantages:** Shared memory communication is generally faster than other IPC mechanisms because processes can directly access and modify shared data. It's especially efficient for large data exchanges.
- **Disadvantages:** Synchronization is essential to prevent race conditions where multiple processes access and modify shared data simultaneously. Semaphore, mutex, and other synchronization mechanisms are often used in conjunction with shared memory.

2. **Message Passing:**

Message passing is another IPC mechanism that involves processes exchanging messages through communication channels provided by the operating system. These messages can contain data, requests, or signals, allowing processes to communicate in a controlled manner. Here's how message passing works:

- **Communication Channels:** Processes communicate via predefined channels or mailboxes established by the operating system.
- **Explicit Communication:** Processes send messages to specific destinations or channels and receive messages from specific sources or channels.
- **Synchronization:** Message passing inherently involves synchronization because processes might have to wait for a message to arrive before continuing their execution.
- **Advantages:** Message passing provides isolation between processes since they can only exchange messages through designated communication channels. This isolation reduces the potential for conflicts and race conditions.

- **Disadvantages:** Message passing can have higher overhead compared to shared memory due to the involvement of the operating system in message routing and handling.

In summary, both shared memory and message passing are essential IPC mechanisms that facilitate communication and cooperation between processes. Shared memory allows processes to exchange data by sharing a common memory region, while message passing involves processes exchanging messages through predefined communication channels. The choice between these mechanisms depends on factors such as performance requirements, data size, synchronization needs, and the desired level of isolation between processes.

❖ COMMUNICATION IN CLIENT-SERVER SYSTEMS

Client/Server communication involves two components, namely a client and a server. They are usually multiple clients in communication with a single server. The clients send requests to the server and the server responds to the client requests.

There are three main methods to client/server communication. These are given as follows

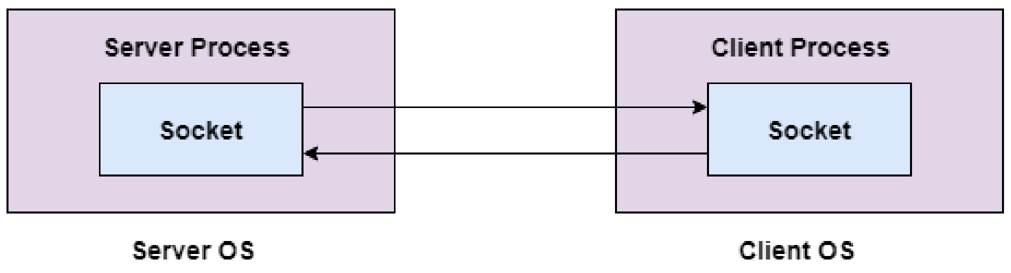
1.) Socket
2.) Remote Procedure Calls
3.) Pipes

1.) Sockets

Sockets facilitate communication between two processes on the same machine or different machines. They are used in a client/server framework and consist of the IP address and port number. Many application protocols use sockets for data connection and data transfer between a client and a server.

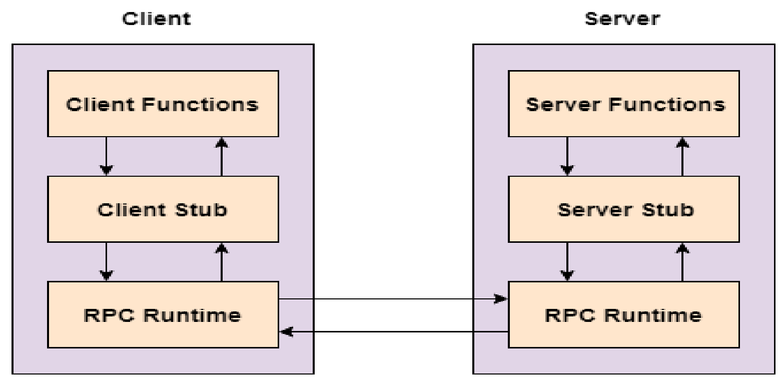
Socket communication is quite low-level as sockets only transfer an unstructured byte stream across processes. The structure on the byte stream is imposed by the client and server applications.

A diagram that illustrates sockets is as follows



2.) Remote Procedure Calls

These are interprocess communication techniques that are used for client-server based applications. A remote procedure call is also known as a subroutine call or a function call.
A client has a request that the RPC translates and sends to the server. This request may be a procedure or a function call to a remote server. When the server receives the request, it sends the required response back to the client



3.) Pipes
These are interprocess communication methods that contain two end points. Data is entered from one end of the pipe by a process and consumed from the other end by the other process.

The two different types of pipes are ordinary pipes and named pipes. Ordinary pipes only allow one way communication. For two way communication, two pipes are required. Ordinary pipes have a parent child relationship between the processes as the pipes can only be accessed by processes that created or inherited them.

Named pipes are more powerful than ordinary pipes and allow two way communication. These pipes exist even after the processes using them have terminated. They need to be explicitly deleted when not required anymore.

A diagram that demonstrates pipes are given as follows

Parent Process
Child Process

PART-2 – MULTITHREADED PROGRAMMING

Multithreading is a technique that allows multiple tasks to be executed concurrently within a single process. This is done by dividing the process into multiple threads, each of which has its own stack and registers. The threads share the same memory space and resources, but they can run independently of each other.

In an operating system, multithreading is implemented by the kernel. The kernel creates and manages the threads, and it also schedules them to run on the CPU. There are two main types of multithreading:

User-level threading: This is where the threads are created and managed by the application itself. The kernel is not involved in the threading process.
Kernel-level threading:This is where the threads are created and managed by the kernel. The application does not have direct control over the threads.

- **Multithreading has several benefits, including:**

Improved performance: Multithreading can improve the performance of a program by allowing multiple tasks to be executed concurrently. This can be especially beneficial for programs that have to wait for I/O operations to complete.
Increased responsiveness: Multithreading can also improve the responsiveness of a program by allowing the user interface to remain responsive even when the program is performing time-consuming tasks.
Better resource utilization: Multithreading can help to improve the utilization of CPU resources by allowing multiple threads to run on the CPU simultaneously.

- **Here are some examples of multithreading in operating systems:**
  - A web browser might use multiple threads to download a web page, render the page, and handle user input.
  - A word processor might use multiple threads to spell check a document, format the document, and print the document.
  - A game might use multiple threads to render the game graphics, update the game physics, and handle user input.

Multithreading is a powerful technique that can be used to improve the performance and responsiveness of programs. However, it is important to use multithreading carefully, as it can also introduce some challenges, such as:

Synchronization: It is important to synchronize the threads so that they do not access shared resources at the same time. This can be done using locks and mutexes.
Deadlocks: Deadlocks can occur when two or more threads are waiting for each other to release a resource. This can be avoided by using proper synchronization techniques.
Race conditions: Race conditions can occur when two or more threads are accessing the same resource at the same time and the outcome of the program depends on the order in which the threads access the resource. This can be avoided by using proper synchronization techniques.

❖ MULTITHREADING MODELS

Multithreading allows the execution of multiple parts of a program at the same time. These parts are known as threads and are lightweight processes available within the process. Therefore, multithreading leads to maximum utilization of the CPU by multitasking.
Multi threading-It is a process of multiple threads executes at same time.

There are two main threading models in process management: user-level threads and kernel-level threads.

User-level threads: In this model, the operating system does not directly support threads. Instead, threads are managed by a user-level thread library, which is part of the application. The library manages the threads and schedules them on available processors. The advantages of user-level threads include greater flexibility and portability, as the application has more control over thread management. However, the disadvantage is that user-level threads are not as efficient as kernel-level threads, as they rely on the application to manage thread scheduling.

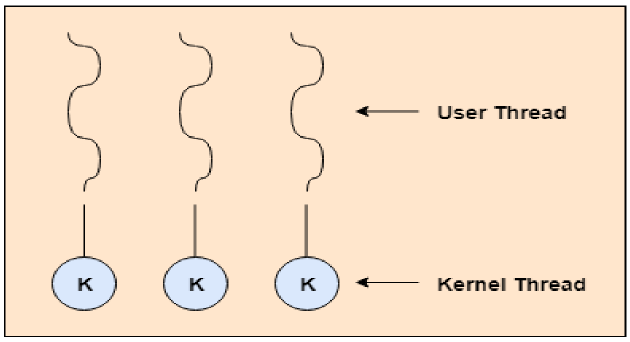
Kernel-level threads: In this model, the operating system directly supports threads as part of the kernel. Each thread is a separate entity that can be scheduled and executed independently by the operating system. The advantages of kernel-level threads include better performance and scalability, as the operating system can schedule threads more efficiently. However, the disadvantage is that kernel-level threads are less flexible and portable than user-level threads, as they are managed by the operating system.

The main models for multithreading are one to one model, many to one model and many to many model. Details about these are given as follows −

I. One to One Model(Kernel-level Threads)

The one to one model maps each of the user threads to a kernel thread. This means that many threads can run in parallel on multiprocessors and other threads can run when one thread makes a blocking system call.

A disadvantage of the one to one model is that the creation of a user thread requires a corresponding kernel thread. Since a lot of kernel threads burden the system, there is restriction on the number of threads in the system.



One to One Model

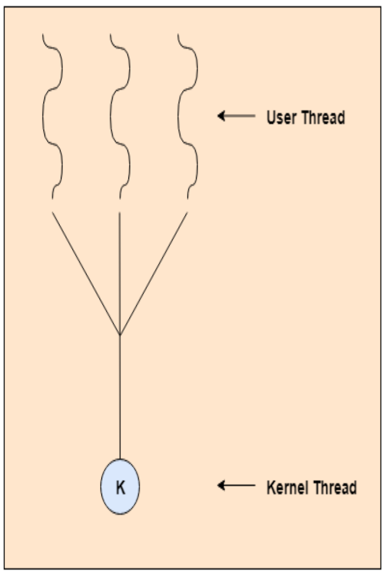
A diagram that demonstrates the one to one model is given as above

II. Many to One Model(User-level Threads)

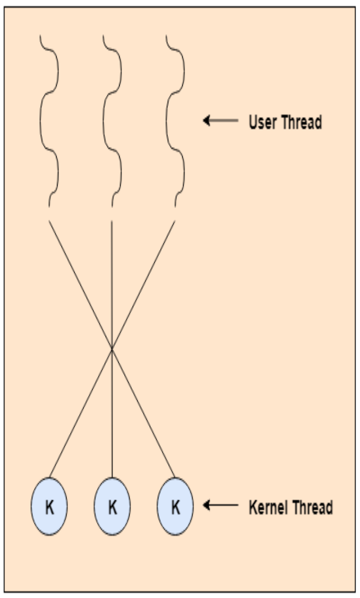
The many to one model maps many of the user threads to a single kernel thread. This model is quite efficient as the user space manages the thread management.

A disadvantage of the many to one model is that a thread blocking system call blocks the entire process. Also, multiple threads cannot run in parallel as only one thread can access the kernel at a time.

A diagram that demonstrates the many to one model is given as follows



Many to One Model



Many to Many Model

III. Many to Many Model

The many to many model maps many of the user threads to a equal number or lesser kernel threads. The number of kernel threads depends on the application or machine.

The many to many does not have the disadvantages of the one to one model or the many to one model. There can be as many user threads as required and their corresponding kernel threads can run in parallel on a multiprocessor.

A diagram that demonstrates the many to many model is given as follows –

❖ THREAD LIBRARIES

Thread libraries, also known as threading libraries or thread packages, are software libraries that provide a set of functions and tools for programmers to create, manage, and synchronize threads in their applications. These libraries abstract the underlying operating system's thread management mechanisms and provide a higher-level interface for developers to work with threads. There are two main types of thread libraries:

1. User-Level Thread Libraries:

- User-level thread libraries are implemented entirely in user space and do not require kernel support for thread management.
- These libraries provide their own thread creation, scheduling, and synchronization mechanisms.
- They are more lightweight in terms of context switching and memory overhead compared to kernel-level threads.
- However, user-level threads are limited by the fact that if one thread blocks, the entire process is blocked.

Advantages

- Thread switching does not require Kernel mode privileges.
- User level thread can run on any operating system.
- Scheduling can be application specific in the user level thread.
- User level threads are fast to create and manage.

Disadvantages

- In a typical operating system, most system calls are blocking.
- Multithreaded application cannot take advantage of multiprocessing.

Examples of user-level thread libraries include:

- POSIX Threads (Pthreads) for C/C++
- Windows User-Mode Thread API
- GNU Portable Threads (GNU Pth)
- Java Green Threads (pre-Java 1.3)

2. Kernel-Level Thread Libraries:

- Kernel-level thread libraries are implemented and managed by the operating system's kernel.
- Each thread is treated as a separate entity by the operating system and is scheduled independently.
- Kernel-level threads offer true parallelism as they can be scheduled on multiple processors or cores.
- However, creating and managing kernel-level threads can involve more overhead compared to user-level threads.

Advantages

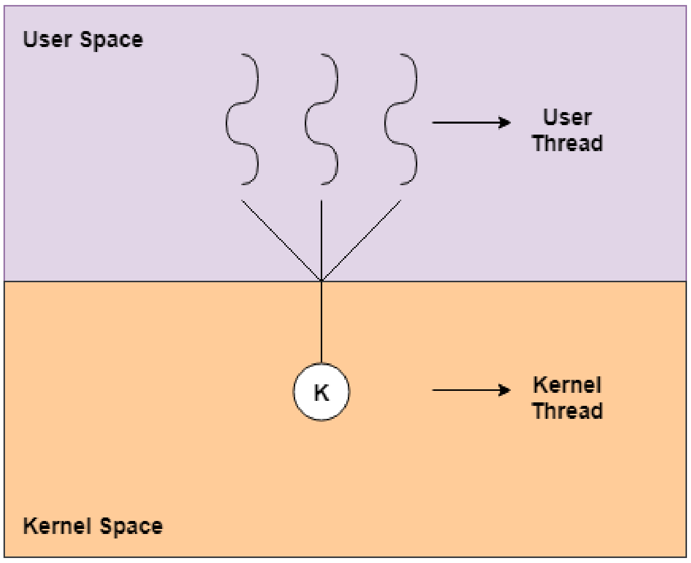
- Kernel can simultaneously schedule multiple threads from the same process on multiple processes.
- If one thread in a process is blocked, the Kernel can schedule another thread of the same process.
- Kernel routines themselves can multithreaded.

Disadvantages

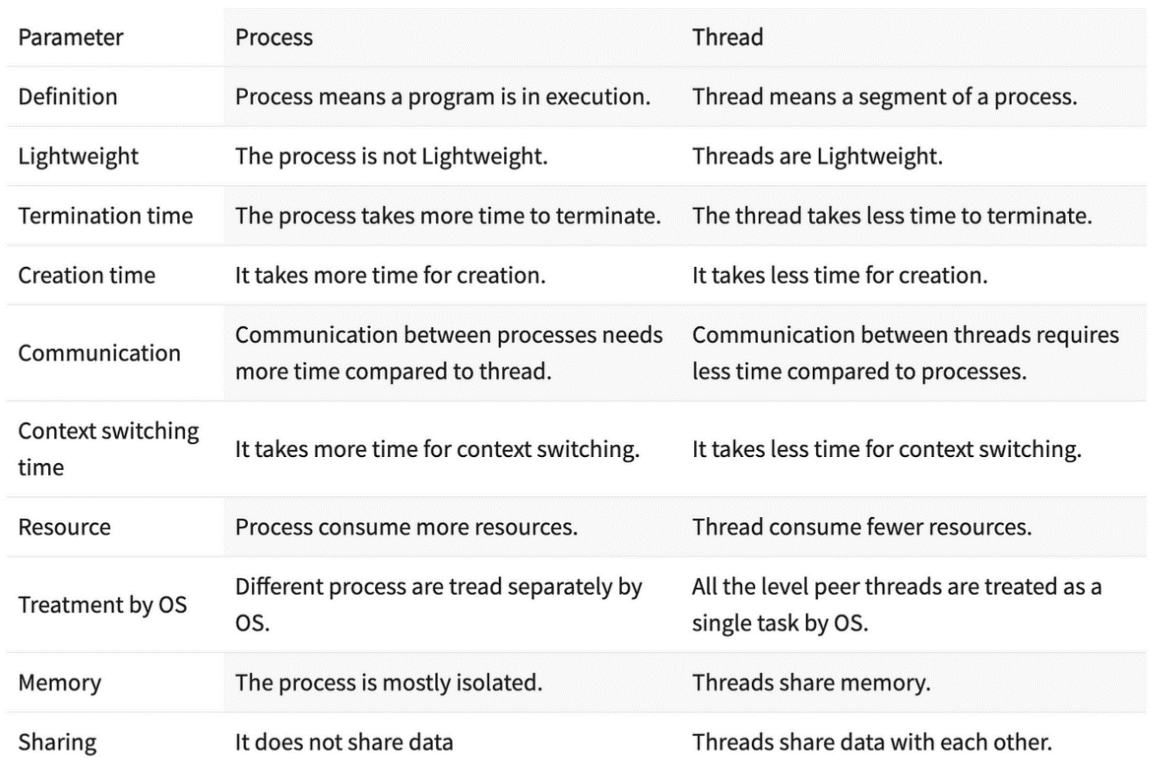
- Kernel threads are generally slower to create and manage than the user threads.
- Transfer of control from one thread to another within same process requires a mode switch to the Kernel.

Examples of kernel-level thread libraries include:
- Windows Kernel-Mode Thread API
- POSIX Threads (Pthreads) with kernel-level thread support
- Windows NT Native API for thread management

The choice between user-level and kernel-level thread libraries depends on factors such as the desired level of parallelism, the application's performance requirements, and the features provided by the underlying operating system. Different operating systems and programming languages may have their own preferred or supported thread libraries. Additionally, modern hardware advancements, like multicore processors and hardware-level thread management, have influenced the evolution of thread libraries to better harness the available computing power.



No.	Parameters	User Level Thread	Kernel Level Thread
1.	Implemented by	User threads are implemented by users.	Kernel threads are implemented by Operating System (OS).
2.	Recognize	Operating System doesn't recognize user level threads.	Kernel threads are recognized by Operating System.
3.	Implementation	Implementation of User threads is easy.	Implementation of Kernel thread is complicated.
4.	Context switch time	Context switch time is less.	Context switch time is more.
5.	Hardware support	Context switch requires no hardware support.	Hardware support is needed.
6.	Blocking operation	If one user level thread performs blocking operation then entire process will be blocked.	If one kernel thread perform blocking operation then another thread can continue execution.
7.	Multithreading	Multithread applications cannot take advantage of multiprocessing.	Kernels can be multithreaded.
8.	Creation and Management	User level threads can be created and managed more quickly.	Kernel level threads take more time to create and manage.
9.	Operating System	Any operating system can support user-level threads.	Kernel level threads are operating system-specific.



Parameter	Process	Thread
Definition	Process means a program is in execution.	Thread means a segment of a process.
Lightweight	The process is not Lightweight.	Threads are Lightweight.
Termination time	The process takes more time to terminate.	The thread takes less time to terminate.
Creation time	It takes more time for creation.	It takes less time for creation.
Communication	Communication between processes needs more time compared to thread.	Communication between threads requires less time compared to processes.
Context switching time	It takes more time for context switching.	It takes less time for context switching.
Resource	Process consume more resources.	Thread consume fewer resources.
Treatment by OS	Different process are tread separately by OS.	All the level peer threads are treated as a single task by OS.
Memory	The process is mostly isolated.	Threads share memory.
Sharing	It does not share data	Threads share data with each other.

❖ THREAD ISSUES

Threading in programming introduces several potential issues and challenges that developers need to be aware of and address to ensure correct and efficient operation of their multithreaded applications. Here are some common threading issues:

1. Race Conditions:

Race conditions occur when two or more threads access shared resources concurrently, and at least one of them modifies the resource.
This can lead to unpredictable behavior, incorrect results, and data corruption.
To avoid race conditions, synchronization mechanisms like locks, semaphores, and mutexes are used to ensure that only one thread can access a shared resource at a time.

2. Deadlocks:

Deadlocks occur when two or more threads are unable to proceed because each is waiting for a resource that another thread holds.This can cause the application to hang indefinitely.
Preventing deadlocks involves careful resource management and proper use of synchronization mechanisms.

3. **Starvation**:

Starvation happens when a thread is perpetually denied access to resources or CPU time.
This can occur if a thread with lower priority is always preempted by higher-priority threads.
Proper scheduling and priority management can help avoid starvation.

4. Thread Synchronization:

Coordinating the activities of multiple threads requires synchronization mechanisms to ensure that threads do not interfere with each other's operations.Overuse or incorrect use of synchronization can lead to contention and performance degradation.

5. Priority Inversion:
Priority inversion occurs when a lower-priority thread holds a resource that a higher-priority thread needs, effectively blocking the higher-priority thread.This can happen when preemptive scheduling is not properly managed or when priority inheritance protocols are not in place.

6. Thread Scalability:
Scalability issues arise when the performance of a multithreaded application degrades as more threads are added.Contentions for shared resources and synchronization bottlenecks can lead to poor scalability.

7. Memory Consistency:
Memory consistency issues occur when threads access shared memory in an unpredictable order due to optimizations performed by modern processors.Memory barriers or synchronization mechanisms are used to ensure proper memory ordering.

8. Thread Safety:
Ensuring thread safety involves designing classes and methods that can be safely accessed by multiple threads without causing data corruption or inconsistent results.Proper use of locks, atomic operations, and immutable data structures can help achieve thread safety.

9. Interrupts and Signals:
Interrupts and signals can disrupt the normal flow of threads and require careful handling to ensure that threads respond appropriately to these events.

10. Resource Management:
Managing resources like memory, file handles, and network connections among multiple threads can lead to leaks, contention, and inefficiencies if not handled properly.

To address these threading issues, developers need to follow best practices for multithreaded programming, such as using appropriate synchronization mechanisms, avoiding unnecessary shared resources, designing for thread safety, and thoroughly testing their applications under various conditions. Additionally, the use of debugging tools and static code analysis can help identify potential threading problems before they cause issues in production environments.

❖ PROCESS SCHEDULING

What is Process Scheduling in Operating Systems?

The process manager's activity is process scheduling, which involves removing the running process from the CPU and selecting another process based on a specific strategy.

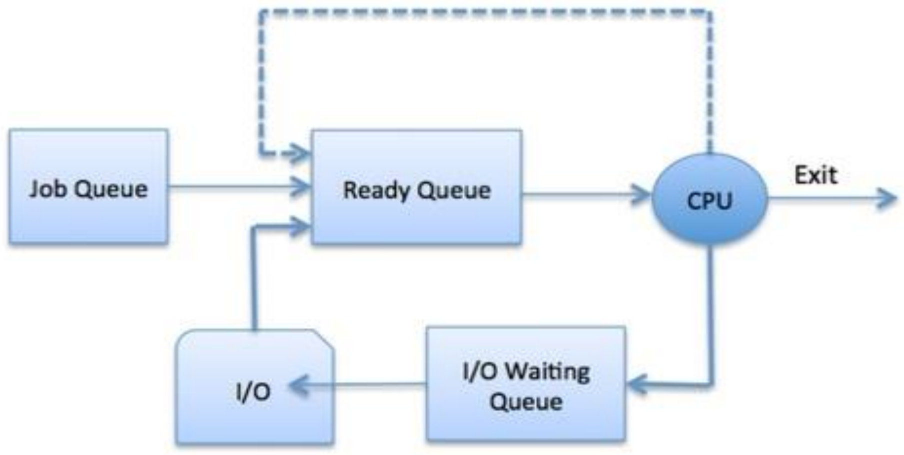
Multiprogramming OS's process scheduling is critical. Multiple processes could be loaded into the executable memory at the same time in such operating systems, and the loaded processes share the CPU utilising temporal multiplexing.

Process Scheduling Queues

All PCBs (Process Scheduling Blocks) are kept in Process Scheduling Queues by the OS. Each processing state has its own queue in the OS, and PCBs from all processes in the same execution state are put in the very same queue. A process's PCB is unlinked from its present queue and then moved to its next state queue when its status changes.

The following major process scheduling queues are maintained by the Operating System:

- Job queue – It contains all of the system's processes.
- Ready queue – This queue maintains a list of all processes in the main memory that are ready to run. This queue is always filled with new processes.
- Device queue – This queue is made up of processes that are stalled owing to the lack of an I/O device.



Each queue can be managed by the OS using distinct policies (FIFO, Priority, Round Robin, etc.). The OS scheduler governs how tasks are moved between the ready and run queues, each of which can only have one item per processor core on a system; it has been integrated with the CPU in the preceding figure.

Two-State Process Model

The running and non-running states of a two-state process paradigm are detailed below.

- **Running**

Whenever a new process is formed, it is immediately put into the operating state of the system.

- **Not currently running**

Processes that are not now executed are queued and will be executed when their turn comes. Each queue entry is a pointer to a certain process. Linked lists are used to implement queues.

What are Scheduling Queues?

- The Job Queue stores all processes that are entered into the system.
- The Ready Queue holds processes in the ready state.
- Device Queues hold processes that are waiting for any device to become available. For each I/O device, there are separate device queues.

The ready queue is where a new process is initially placed. It sits in the ready queue, waiting to be chosen for execution or dispatched. One of the following occurrences can happen once the process has been assigned to the CPU and is running:

- The process could send out an I/O request before being placed in the I/O queue.
- The procedure could start a new one and then wait for it to finish.
- As a result of an interrupt, the process could be forcibly removed from the CPU and returned to the ready queue.

The process finally moves from the waiting to ready state in the first two circumstances and then returns to the ready queue. This cycle is repeated until a process is terminated, at which point it is withdrawn from all queues, and its PCB and resources are reallocated.

Types of Schedulers

Schedulers are specialised computer programmes that manage process scheduling in a variety of ways. Their primary responsibility is to choose which jobs to submit into the system and which processes to run. Click here to learn more on Process Schedulers in Operating System. There are three types of schedulers:

- **Long-Term**

A job scheduler is another name for it. A long-term scheduler determines which programmes are accepted for processing into the system. It picks processes from the queue and then loads them into memory so they can be executed. For CPU scheduling, a process loads into memory.

- **Short-Term**

CPU scheduler is another name for it. Its major goal is to improve system performance according to the set of criteria defined. It refers to the transition from the process's ready state to the running stage. The CPU scheduler happens to choose a process from those that are ready to run and then allocates the CPU to it.

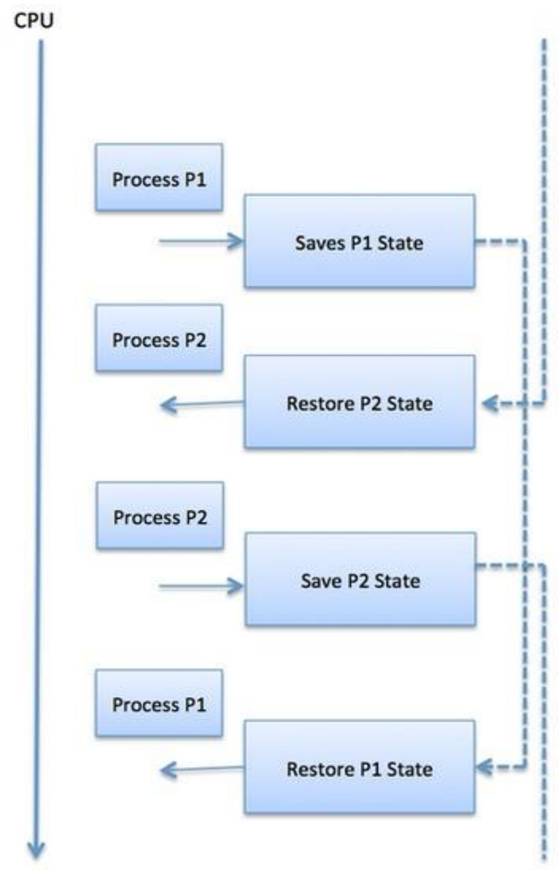
- **Medium-Term**

It includes medium-term scheduling. Swapping clears the memory of the processes. The degree of multiprogramming is reduced. The switched out processes are handled by the medium-term scheduler.

Context Switch

A context switch is a mechanism in the Process Control block that stores and restores the state or context of a CPU so that a process execution can be resumed from the very same point at a later time. A context switcher makes use of this technique to allow numerous programmes to share a single CPU. Thus, context switching is one of the most important elements of a multitasking operating system.

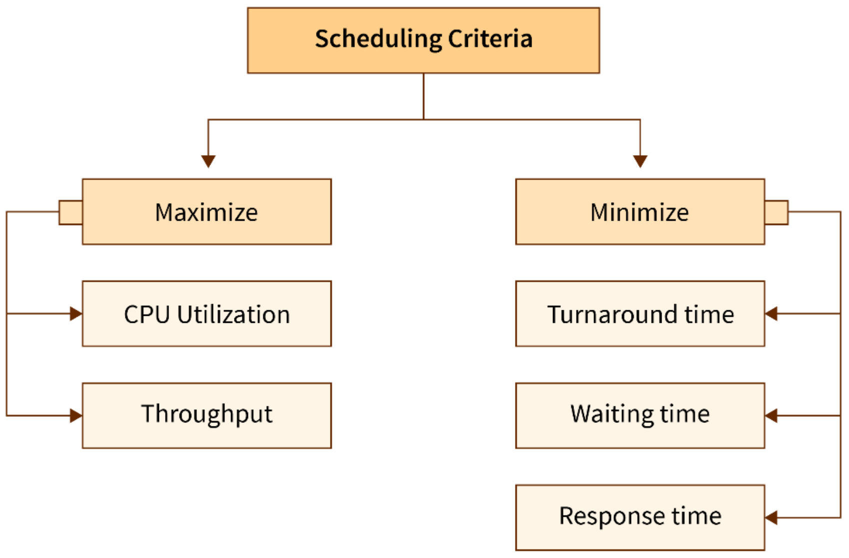
The state of the currently running process is recorded in the PCB when the scheduler changes the CPU from one process to another. The state for the following operation is then loaded from its PCB and used to set the registers, PC, and so on. The second process can then begin its execution.



Because register and memory states must be preserved as well as recovered, context switches can be computationally demanding. Some hardware systems use multiple sets of processor registers in order to save context switching time. The following information is saved for further use when the process is switched.

- Program counter
- Base and limit register value
- Scheduling information
- Currently used register
- I/O State information
- Changed state
- Accounting information

❖ What are the Scheduling Criteria in OS?



The criteria include the following:

1. **CPU utilization:** The main objective of any CPU scheduling algorithm is to keep the CPU as busy as possible. Theoretically, CPU utilization can range from 0 to 100 but in a real-time system, it varies from 40 to 90 percent depending on the load upon the system.

2. **Throughput:** A measure of the work done by the CPU is the number of processes being executed and completed per unit of time. This is called throughput. The throughput may vary depending on the length or duration of the processes.

3. **Turnaround time:** For a particular process, an important criterion is how long it takes to execute that process. The time elapsed from the time of submission of a process to the time of completion is known as the turnaround time. Turn-around time is the sum of times spent waiting to get into memory, waiting in the ready queue, executing in CPU, and waiting for I/O.
   The formula to calculate Turn Around Time = Completion Time – Arrival Time.

4. **Waiting time:** A scheduling algorithm does not affect the time required to complete the process once it starts execution. It only affects the waiting time of a process i.e. time spent by a process waiting in the ready queue.
   The formula for calculating Waiting Time = Turnaround Time – Burst Time.

5. **Response time:** In an interactive system, turn-around time is not the best criterion. A process may produce some output fairly early and continue computing new results while previous results are being output to the user. Thus another criterion is the time taken from submission of the process of the request until the first response is produced. This measure is called response time.
   The formula to calculate Response Time = CPU Allocation Time(when the CPU was allocated for the first) – Arrival Time

6. **Completion time:** The completion time is the time when the process stops executing, which means that the process has completed its burst time and is completely executed.

7. **Priority:** If the operating system assigns priorities to processes, the scheduling mechanism should favor the higher-priority processes.

8. **Predictability:** A given process always should run in about the same amount of time under a similar system load.

❖ Operating System CPU Scheduling algorithms

A Process Scheduler schedules different processes to be assigned to the CPU based on particular scheduling algorithms. There are six popular process scheduling algorithms.

1. First-Come, First-Served (FCFS) Scheduling
2. Shortest-Job-Next (SJN) Scheduling
3. Priority Scheduling
4. Shortest Remaining Time
5. Round Robin(RR) Scheduling
6. Multiple-Level Queues Scheduling

CPU scheduling algorithms can be categorized into three main categories based on their behavior and characteristics:

1. Preemptive Scheduling:In preemptive scheduling, a running process can be forcibly removed from the CPU before its time slice (quantum) is completed. This is usually done to allocate CPU time to a higher-priority process or to ensure fair distribution of CPU resources. Preemptive scheduling provides better responsiveness and can handle time-critical tasks efficiently. Common preemptive scheduling algorithms include Round Robin, Priority Scheduling, and Multilevel Feedback Queue Scheduling.

2. Non-Preemptive Scheduling: In non-preemptive scheduling, a running process continues to execute on the CPU until it voluntarily releases the CPU or completes its execution. This approach can lead to longer response times for higher-priority processes if a lower-priority process is running for an extended period. Non-preemptive scheduling is simpler to implement compared to preemptive scheduling. Shortest Job Next (SJN), First-Come, First-Served (FCFS), and Priority Scheduling (without preemption) are examples of non-preemptive scheduling algorithms.

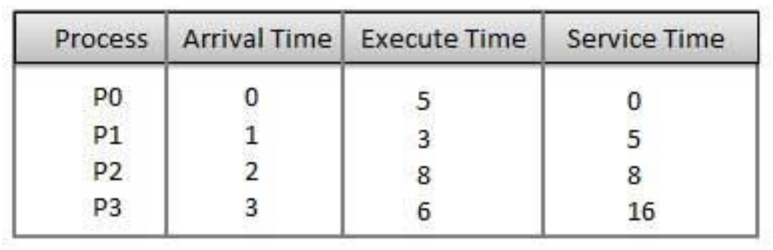
3. Cooperative Scheduling:In cooperative scheduling, also known as voluntary scheduling, a running process yields the CPU to other processes. The process gives up the CPU only when it explicitly decides to do so, such as when it is waiting for I/O or has completed a certain task. Cooperative scheduling requires the processes to be well-behaved and voluntarily cooperate in yielding CPU time, which can be a challenge to ensure in practice. Cooperative scheduling is often used in user-level thread libraries and can be efficient for certain workloads, but it relies heavily on process cooperation and can lead to responsiveness issues if processes do not yield the CPU appropriately.

The choice of scheduling category and specific algorithm depends on factors such as the nature of the workload, the desired system responsiveness, and the trade-offs between various scheduling objectives.

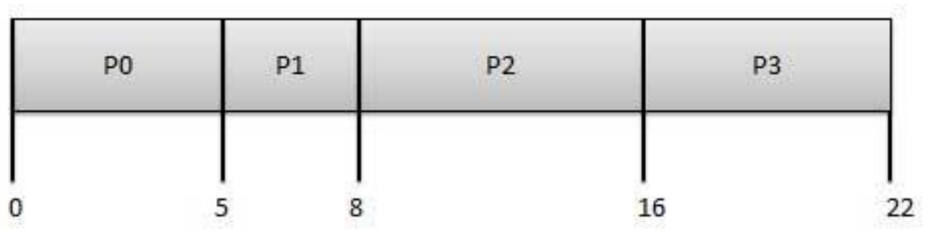
These algorithms are either non-preemptive or preemptive. Non-preemptive algorithms are designed so that once a process enters the running state, it cannot be preempted until it completes its allotted time, whereas the preemptive scheduling is based on priority where a scheduler may preempt a low priority running process anytime when a high priority process enters into a ready state.

1. **First Come First Serve (FCFS)**

- Jobs are executed on first come, first serve basis.
- It is a non-preemptive, pre-emptive scheduling algorithm.
- Easy to understand and implement.
- Its implementation is based on FIFO queue.
- Poor in performance as average wait time is high.



Process	Arrival Time	Execute Time	Service Time
P0	0	5	0
P1	1	3	5
P2	2	8	8
P3	3	6	16



Wait time of each process is as follows

Process	Wait Time : Service Time - Arrival Time
P0	0 - 0 = 0
P1	5 - 1 = 4
P2	8 - 2 = 6
P3	16 - 3 = 13

Average Wait Time: (0+4+6+13) / 4 = 5.75

2. **Shortest Job Next (SJN)**
   - This is also known as shortest job first, or SJF
   - This is a non-preemptive, pre-emptive scheduling algorithm.
   - Best approach to minimize waiting time.
   - Easy to implement in Batch systems where required CPU time is known in advance.
   - Impossible to implement in interactive systems where required CPU time is not known.
   - The processer should know in advance how much time process will take.

Given: Table of processes, and their Arrival time, Execution time.

Process	Arrival Time	Execution Time	Service Time
P0	0	5	0
P1	1	3	5
P2	2	8	14
P3	3	6	8

Waiting time of each process is as follows −

Process	Waiting Time
P0	0 - 0 = 0
P1	5 - 1 = 4
P2	14 - 2 = 12
P3	8 - 3 = 5

Average Wait Time: (0 + 4 + 12 + 5)/4 = 21 / 4 = 5.25

3. Priority Based Scheduling

Priority scheduling is a non-preemptive algorithm and one of the most common scheduling algorithms in batch systems.Each process is assigned a priority. Process with highest priority is to be executed first and so on.Processes with same priority are executed on first come first served basis.

Priority can be decided based on memory requirements, time requirements or any other resource requirement.

Given: Table of processes, and their Arrival time, Execution time, and priority. Here we are considering 1 is the lowest priority.

Process	Arrival Time	Execution Time	Priority	Service Time
P0	0	5	1	0
P1	1	3	2	11
P2	2	8	1	14
P3	3	6	3	5

Waiting time of each process is as follows –

Process	Waiting Time
P0	0 – 0 = 0
P1	11 – 1 = 10
P2	14 – 2 = 12
P3	5 – 3 = 2

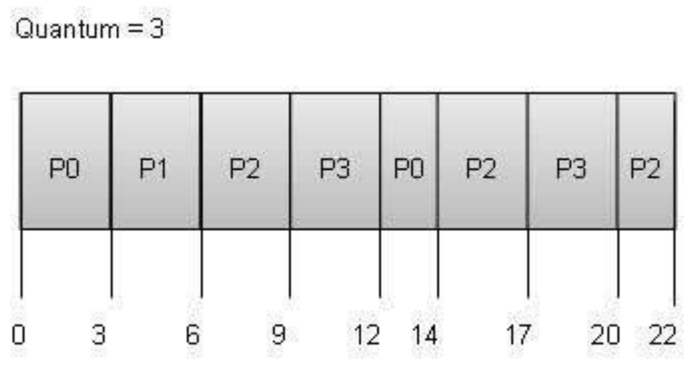
Average Wait Time: (0 + 10 + 12 + 2)/4 = 24 / 4 = 6

4. Shortest Remaining Time (SRT)

This scheduling algorithm is the preemptive version of the SJN algorithm. The OS allocates the processor to the job that is closest to completion. Though there is a chance of this job being preempted, if another job is ready with a shorter time to completion. It is used in batch environments where short jobs are given preference and cannot be implemented in interactive systems where required CPU time is unknown.

5. Round Robin (RR)

This scheduling algorithm is a preemptive process scheduling algorithm where each process is provided a fixed time to execute. This fixed time is called a quantum.It uses context switching to save states of preempted processes. Once a process is done executing for a given time period, it is preempted and another process executes for a given time period.



Wait time of each process is as follows –

Process	Wait Time : Service Time - Arrival Time
P0	(0 - 0) + (12 - 3) = 9
P1	(3 - 1) = 2
P2	(6 - 2) + (14 - 9) + (20 - 17) = 12
P3	(9 - 3) + (17 - 12) = 11

Average Wait Time: (9+2+12+11) / 4 = 8.5

6.Multiple-Level Queues Scheduling

- Multiple-level queues are not an independent scheduling algorithm. They make use of other existing algorithms to group and schedule jobs with common characteristics.
- Multiple queues are maintained for processes with common characteristics.
- Each queue can have its own scheduling algorithms.
- Priorities are assigned to each queue.

For example, CPU-bound jobs can be scheduled in one queue and all I/O-bound jobs in another queue. The Process Scheduler then alternately selects jobs from each queue and assigns them to the CPU based on the algorithm assigned to the queue.

Purpose of a Scheduling algorithm

The purpose of a scheduling algorithm is to provide:

- Maximum CPU utilization
- Minimum turnaround time
- Fair allocation of CPU
- Maximum throughput
- Minimum waiting time
- Minimum response time

Algorithm	Description	Advantages	Disadvantages

FCFS	Schedules the processes in the order in which they arrive.	Simple to implement.	Not very efficient for systems with a lot of short processes.
SJF	Schedules the process with the shortest predicted CPU burst time.	Very efficient for systems with a lot of short processes.	Can be difficult to implement.
SRTF	Similar to SJF, but it schedules the process with the shortest remaining CPU burst time.	Even more efficient than SJF.	Can be even more difficult to implement.
Priority scheduling	Schedules the processes with the highest priority.	Can be used to give certain processes priority over others.	Can be unfair to processes with low priority.
Round robin	Schedules the processes in a circular queue. Each process is given a time slice to run.	Simple to implement.	Can be inefficient for systems with a lot of short processes.
Multilevel queue scheduling	Divides the processes into multiple queues, and each queue has its own scheduling algorithm. The processes are moved between queues based on their priority.	Can achieve a balance between efficiency and fairness.	Can be more complex to implement than other algorithms.

❖ MULTIPLE PROCESSOR SCHEDULING ALGORITHM

Multiple processor scheduling or multiprocessor scheduling focuses on designing the system's scheduling function, which consists of more than one processor. Multiple CPUs share the load (load sharing) in multiprocessor scheduling so that various processes run simultaneously. In general, multiprocessor scheduling is complex as compared to single processor scheduling. In the multiprocessor scheduling, there are many processors, and they are identical, and we can run any process at any time.

The multiple CPUs in the system are in close communication, which shares a common bus, memory, and other peripheral devices. So we can say that the system is tightly coupled. These systems are used when we want to process a bulk amount of data, and these systems are mainly used in satellite, weather forecasting, etc.

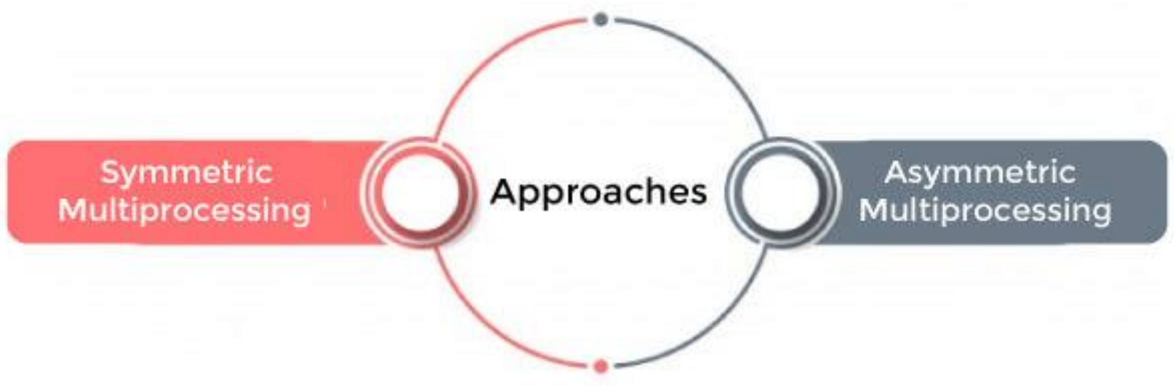
There are cases when the processors are identical, i.e., homogenous, in terms of their functionality in multiple-processor scheduling. We can use any processor available to run any process in the queue.

Multiprocessor systems may be heterogeneous (different kinds of CPUs) or homogenous (the same CPU). There may be special scheduling constraints, such as devices connected via a private bus to only one CPU.

There is no policy or rule which can be declared as the best scheduling solution to a system with a single processor. Similarly, there is no best scheduling solution for a system with multiple processors as well.

Approaches to Multiple Processor Scheduling

There are two approaches to multiple processor scheduling in the operating system: Symmetric Multiprocessing and Asymmetric Multiprocessing.



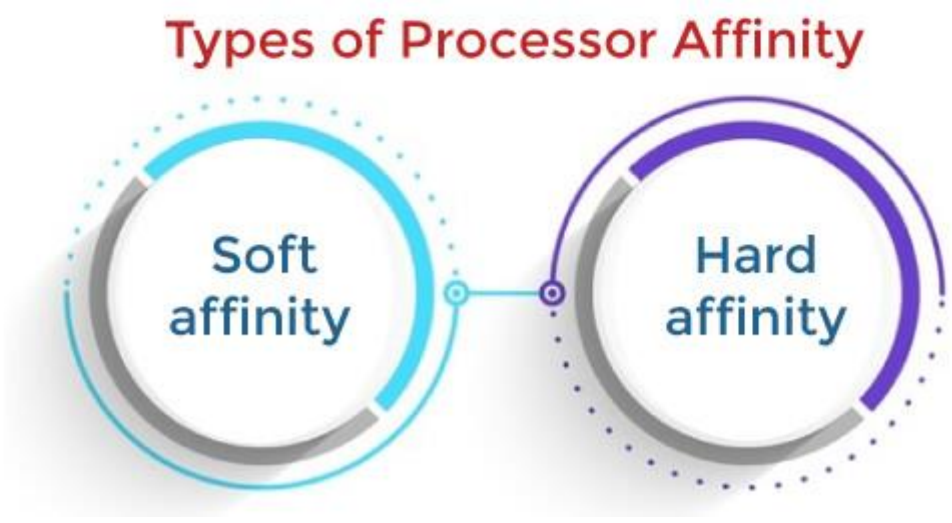
1.Symmetric Multiprocessing: It is used where each processor is self-scheduling. All processes may be in a common ready queue, or each processor may have its private queue for ready processes. The scheduling proceeds further by having the scheduler for each processor examine the ready queue and select a process to execute.

2.Asymmetric Multiprocessing: It is used when all the scheduling decisions and I/O processing are handled by a single processor called the Master Server. The other processors execute only the user code. This is simple and reduces the need for data sharing, and this entire scenario is called Asymmetric Multiprocessing.

Processor Affinity

Processor Affinity means a process has an affinity for the processor on which it is currently running. When a process runs on a specific processor, there are certain effects on the cache memory. The data most recently accessed by the process populate the cache for the processor.

As a result, successive memory access by the process is often satisfied in the cache memory. Now, suppose the process migrates to another processor. In that case, the contents of the cache memory must be invalidated for the first processor, and the cache for the second processor must be repopulated. Because of the high cost of invalidating and repopulating caches, most SMP(symmetric multiprocessing) systems try to avoid migrating processes from one processor to another and keep a process running on the same processor. This is known as



processor affinity. There are two types of processor affinity, such as:

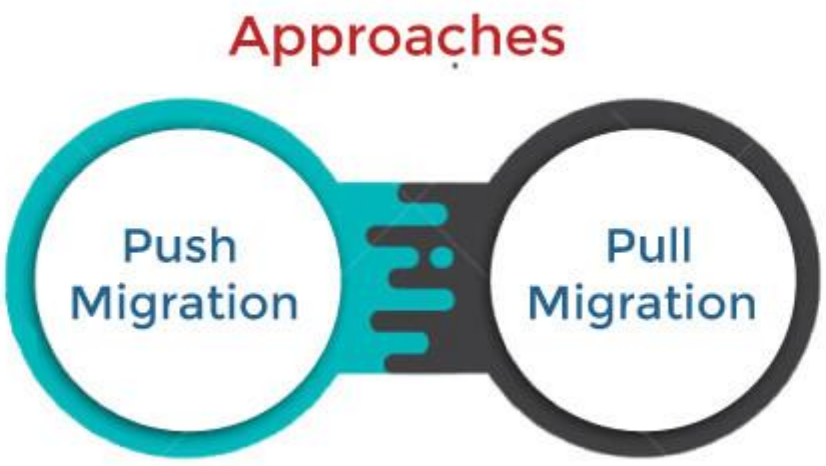
1.Soft Affinity: When an operating system has a policy of keeping a process running on the same processor but not guaranteeing it will do so, this situation is called soft affinity.

2.Hard Affinity: Hard Affinity allows a process to specify a subset of processors on which it may run. Some Linux systems implement soft affinity and provide system calls like sched_setaffinity() that also support hard affinity.

Load Balancing

Load Balancing is the phenomenon that keeps the workload evenly distributed across all processors in an SMP system. Load balancing is necessary only on systems where each processor has its own private queue of a process that is eligible to execute.

Load balancing is unnecessary because it immediately extracts a runnable process from the common run queue once a processor becomes idle. On SMP (symmetric multiprocessing), it is important to keep the workload balanced among all processors to utilize the benefits of having more than one processor fully. One or more processors will sit idle while other processors have high workloads along with lists of processors awaiting the CPU. There are two general approaches to load balancing:



1.Push Migration: In push migration, a task routinely checks the load on each processor. If it finds an imbalance, it evenly distributes the load on each processor by moving the processes from overloaded to idle or less busy processors.

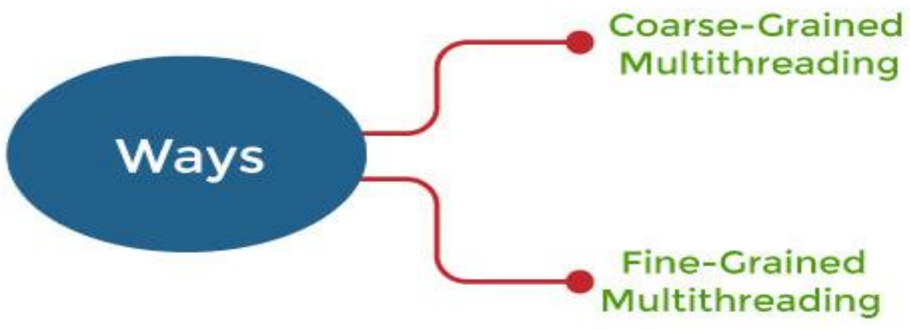
2.Pull Migration: Pull Migration occurs when an idle processor pulls a waiting task from a busy processor for its execution.

<u>Multi-core Processors</u>

In multi-core processors, multiple processor cores are placed on the same physical chip. Each core has a register set to maintain its architectural state and thus appears to the operating system as a separate physical processor. SMP systems that use multi-core processors are faster and consume less power than systems in which each processor has its own physical chip.

However, multi-core processors may complicate the scheduling problems. When the processor accesses memory, it spends a significant amount of time waiting for the data to become available. This situation is called a Memory stall. It occurs for various reasons, such as cache miss, which is accessing the data that is not in the cache memory.

In such cases, the processor can spend upto 50% of its time waiting for data to become available from memory. To solve this problem, recent hardware designs have implemented multithreaded processor cores in which two or more hardware threads are assigned to each core. Therefore if one thread stalls while waiting for the memory, the core can switch to another thread. There are two ways to multithread a processor:



1.Coarse-Grained Multithreading: A thread executes on a processor until a long latency event such as a memory stall occurs in coarse-grained multithreading. Because of the delay caused by the long latency event, the processor must switch to another thread to begin execution. The cost of switching between threads is high as the instruction pipeline must be terminated before the other thread can begin execution on the processor core. Once this new thread begins execution, it begins filling the pipeline with its instructions.

2.Fine-Grained Multithreading: This multithreading switches between threads at a much finer level, mainly at the boundary of an instruction cycle. The architectural design of fine-grained systems includes logic for thread switching, and as a result, the cost of switching between threads is small.

Virtualization and Threading

In this type of multiple processor scheduling, even a single CPU system acts as a multiple processor system. In a system with virtualization, the virtualization presents one or more virtual CPUs to each of the virtual machines running on the system. It then schedules the use of physical CPUs among the virtual machines.

- Most virtualized environments have one host operating system and many guest operating systems, and the host operating system creates and manages the virtual machines.
- Each virtual machine has a guest operating system installed, and applications run within that guest.
- Each guest operating system may be assigned for specific use cases, applications, or users, including time-sharing or real-time operation.
- Any guest operating-system scheduling algorithm that assumes a certain amount of progress in a given amount of time will be negatively impacted by the virtualization.
- A time-sharing operating system tries to allot 100 milliseconds to each time slice to give users a reasonable response time. A given 100 millisecond time slice may take much more than 100 milliseconds of virtual CPU time. Depending on how busy the system is, the time slice may take a second or more, which results in a very poor response time for users logged into that virtual machine.
- The net effect of such scheduling layering is that individual virtualized operating systems receive only a portion of the available CPU cycles, even though they believe they are receiving all cycles and scheduling all of those cycles. The time-of-day clocks in virtual machines are often incorrect because timers take no longer to trigger than they would on dedicated CPUs.
- Virtualizations can thus undo the good scheduling algorithm efforts of the operating systems within virtual machines.

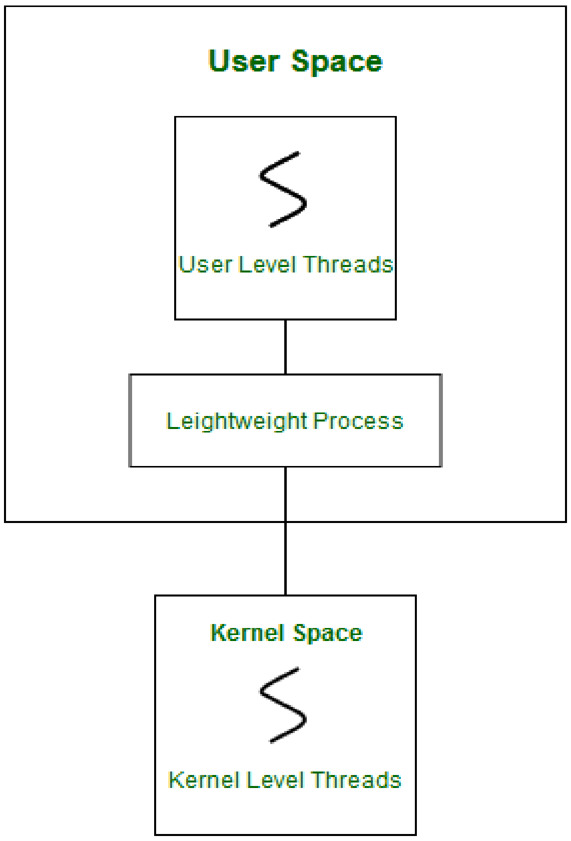
❖ THREAD SCHEDULING

Scheduling of threads involves two boundary scheduling,

- Scheduling of user level threads (ULT) to kernel level threads (KLT) via lightweight process (LWP) by the application developer.
- Scheduling of kernel level threads by the system scheduler to perform different unique os functions.

1. Lightweight Process (LWP) :

Light-weight process are threads in the user space that acts as an interface for the ULT to access the physical CPU resources. Thread library schedules which thread of a process to run on which LWP and how long. The number of LWP created by the thread library depends on the type of application. In the case of an I/O bound application, the number of LWP depends on the number of user-level threads. This is because when an LWP is blocked on an I/O operation, then to invoke the other ULT the thread library needs to create and schedule another LWP. Thus, in an I/O bound application, the number of LWP is equal to the number of the ULT. In the case of a CPU bound application, it depends only on the application. Each LWP is attached to a separate kernel-level thread.



In real-time, the first boundary of thread scheduling is beyond specifying the scheduling policy and the priority. It requires two controls to be specified for the User level threads: Contention scope, and Allocation domain. These are explained as following below.

1. Contention Scope :

The word contention here refers to the competition or fight among the User level threads to access the kernel resources. Thus, this control defines the extent to which contention takes place. It is defined by the application developer using the thread library. Depending upon the extent of contention it is classified as Process Contention Scope and System Contention Scope.

2.Process Contention Scope (PCS) –

The contention takes place among threads within a same process. The thread library schedules the high-prioritized PCS thread to access the resources via available LWPs (priority as specified by the application developer during thread creation).

int Pthread_attr_setscope(pthread_attr_t *attr, int scope)

The first parameter denotes to which thread within the process the scope is defined. The second parameter defines the scope of contention for the thread pointed. It takes two values.

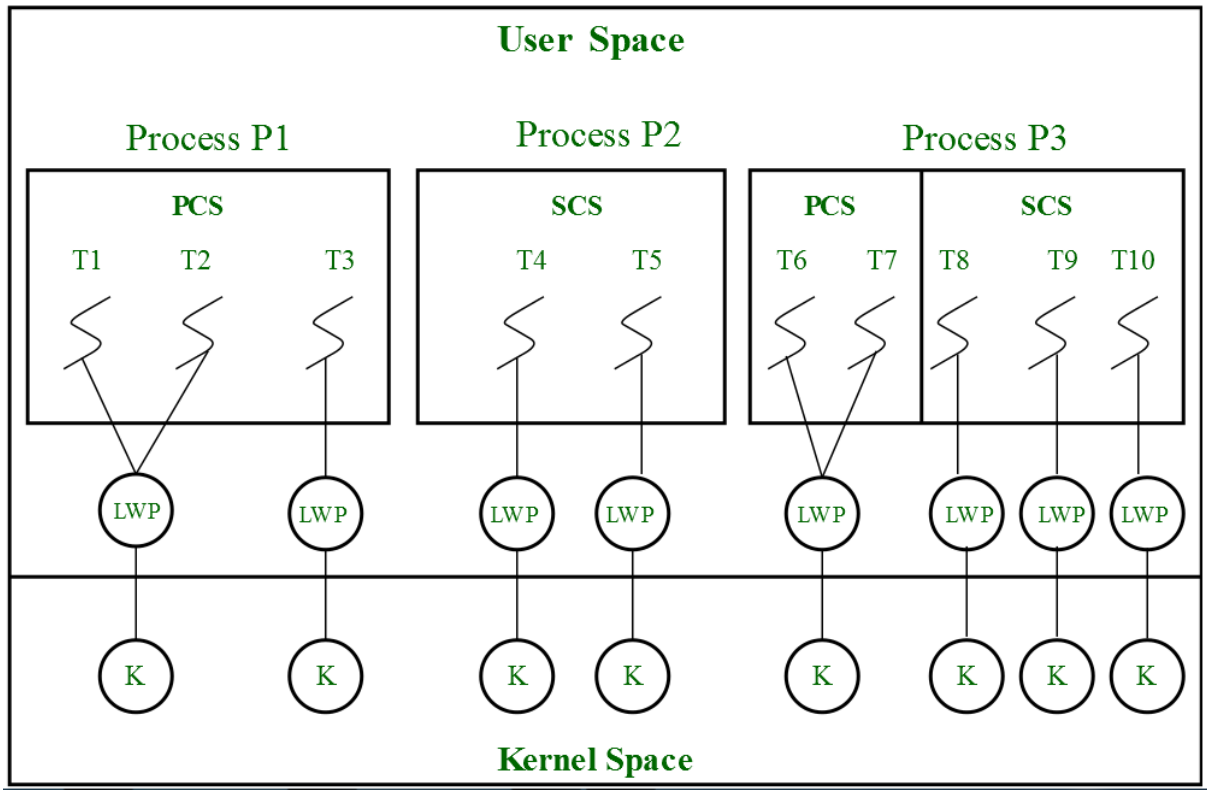
PTHREAD_SCOPE_SYSTEM

PTHREAD_SCOPE_PROCESS If the scope value specified is not supported by the system, then the function returns ENOTSUP.

Allocation Domain :

The allocation domain is a set of one or more resources for which a thread is competing. In a multicore system, there may be one or more allocation domains where each consists of one or more cores. One ULT can be a part of one or more allocation domain. Due to this high complexity in dealing with hardware and software architectural interfaces, this control is not specified. But by default, the multicore system will have an interface that affects the allocation domain of a thread.

Consider a scenario, an operating system with three process P1, P2, P3 and 10 user level threads (T1 to T10) with a single allocation domain. 100% of CPU resources will be distributed among all the three processes. The amount of CPU resources allocated to each process and to each thread depends on the contention scope, scheduling policy and priority of each thread defined by the application developer using thread library and also depends on the system scheduler. These User level threads are of a different contention scope.



In this case, the contention for allocation domain takes place as follows,

Process P1:

All PCS threads T1, T2, T3 of Process P1 will compete among themselves. The PCS threads of the same process can share one or more LWP. T1 and T2 share an LWP and T3 are allocated to a separate LWP. Between T1 and T2 allocation of kernel resources via LWP is based on preemptive priority scheduling by the thread library. A Thread with a high priority will preempt low priority threads. Whereas, thread T1 of process p1 cannot preempt thread T3 of process p3 even if the priority of T1 is greater than the priority of T3. If the priority is equal, then the allocation of ULT to available LWPs is based on the scheduling policy of threads by the system scheduler(not by thread library, in this case).

Process P2:

Both SCS threads T4 and T5 of process P2 will compete with processes P1 as a whole and with SCS threads T8, T9, T10 of process P3. The system scheduler will schedule the kernel resources among P1, T4, T5, T8, T9, T10, and PCS threads (T6, T7) of process P3 considering each as a separate process. Here, the Thread library has no control of scheduling the ULT to the kernel resources.

Process P3:

Combination of PCS and SCS threads. Consider if the system scheduler allocates 50% of CPU resources to process P3, then 25% of resources is for process scoped threads and the remaining 25% for system scoped threads. The PCS threads T6 and T7 will be allocated to access the 25% resources based on the priority by the thread library. The SCS threads T8, T9, T10 will divide the 25% resources among themselves and access the kernel resources via separate LWP and KLT. The SCS scheduling is by the system scheduler.

Advantages of PCS over SCS :

- If all threads are PCS, then context switching, synchronization, scheduling everything takes place within the userspace. This reduces system calls and achieves better performance.
- PCS is cheaper than SCS.
- PCS threads share one or more available LWPs. For every SCS thread, a separate LWP is associated.For every system call, a separate KLT is created.
- The number of KLT and LWPs created highly depends on the number of SCS threads created. This increases the kernel complexity of handling scheduling and synchronization. Thereby, results in a limitation over SCS thread creation, stating that, the number of SCS threads to be smaller than the number of PCS threads.
- If the system has more than one allocation domain, then scheduling and synchronization of resources becomes more tedious. Issues arise when an SCS thread is a part of more than one allocation domain, the system has to handle n number of interfaces.
- The second boundary of thread scheduling involves CPU scheduling by the system scheduler. The scheduler considers each kernel-level thread as a separate process and provides access to the kernel resources.

PART-3 INTERPROCESS COMMUNICATION

Synchronization in operating systems refers to the coordination of multiple processes or threads to ensure that they access shared resources in a controlled and orderly manner. The goal of synchronization is to prevent conflicts, race conditions, and data inconsistencies that can arise when multiple processes or threads access shared resources concurrently.

- **Common shared resources that require synchronization include:**

Memory: When multiple processes or threads share memory, it's important to ensure that they don't overwrite each other's data inadvertently.

Files: When multiple processes or threads read from or write to the same file, synchronization is necessary to maintain data integrity.

I/O Devices: Shared input/output devices such as printers or network interfaces need synchronization to prevent data corruption and ensure proper functioning.

Data Structures: When multiple processes or threads manipulate data structures (like linked lists, queues, or databases), synchronization is needed to avoid inconsistencies.

- **Synchronization Mechanisms:**

Mutex (Mutual Exclusion): A mutex is a synchronization primitive that allows only one process or thread to access a shared resource at a time. When a process or thread wants to access the resource, it must first acquire the mutex. If the mutex is locked by another entity, the requesting entity is blocked until the mutex is released.

Semaphore: A semaphore is a synchronization mechanism that maintains a count. It allows a specified number of processes or threads to access a shared resource concurrently. Semaphores can be used to implement various synchronization scenarios, like limiting the number of concurrent accesses.

Locks: Locks, also known as locks or critical sections, are synchronization mechanisms used to protect a section of code that accesses shared resources. They prevent multiple processes or threads from entering the protected section simultaneously.

Condition Variables: Condition variables are used to signal and wait for specific conditions to be met. They are often used in conjunction with locks to implement more complex synchronization scenarios, like producer-consumer problems.

Barriers: Barriers are synchronization points where multiple processes or threads wait until all of them have reached the same point before continuing execution. They are useful for scenarios where all entities need to synchronize before proceeding.

Read-Write Locks: These locks differentiate between read-only and write accesses. Multiple processes or threads can hold a read lock simultaneously, but only one can hold a write lock. This is useful in scenarios where multiple reads are safe but writes need exclusive access.

Synchronization is a critical aspect of multi-threaded and multi-process programming to ensure data consistency and avoid race conditions. However, improper synchronization can lead to deadlocks (where processes are stuck waiting for each other) or performance bottlenecks. Choosing the appropriate synchronization mechanism depends on the specific problem and requirements of the application.

The Critical-Section Problem

We begin our consideration of process synchronization by discussing the so called criticalsection problem. Consider a system consisting of n processes {P0, P1, ..., Pn−1}. Each processhas a segment of code, called a critical section, in which the process may be changing common variables, updating a table, writing a file, and so on. The important feature of the system is that, when one process is executing in its critical section, no other process is allowed to execute in its critical section. That is, no two processes are executing in their critical sections at the same time. The critical-section problem is to design a protocol that the processes can use to cooperate. Each process must request permission to enter its critical section.

The section of code implementing this request is the entry section. The critical section may be followed by an exit section. The remaining code is the remainder section. The general structure of a typical process Pi is shown in Figure. The entry section and exit section are enclosed in boxes to highlight these important segments of code.

```
do {

    entry section

        critical section

    exit section

        remainder section

} while (true);
```

General structure of a typical process P_i.

A solution to the critical-section problem must satisfy the following three requirements:

1.Mutual exclusion. If process Pi is executing in its critical section, then no other processes can be executing in their critical sections.

2. Progress. If no process is executing in its critical section and some processes wish to enter their critical sections, then only those processes that are not executing in their remainder sections can participate in deciding which will enter its critical section next, and this selection cannot be postponed indefinitely.

3. Bounded waiting. There exists a bound, or limit, on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.

Two general approaches are used to handle critical sections in operating systems:

1) Preemptive kernels and

2) non-Preemptive kernels.

A preemptive (preemption is the act of temporarily interrupting a task being carried out by a computer system) kernel allows a process to be preempted while it is running in kernel mode. A nonpreemptive kernel does not allow a process running in kernel mode to preempted; a kernel-mode process will run until it exits kernel mode, blocks, or voluntarily yields control of the CPU.

1.Preemptive Kernel:

A preemptive kernel is one in which the operating system can interrupt a running task or process and switch to another task without the cooperation of the currently executing task. In other words, the kernel has the ability to forcefully stop a running task and allocate the CPU to another task. This approach ensures that no single task can monopolize the CPU and that tasks can be switched in and out rapidly.

- **Advantages of preemptive kernels:**

Improved responsiveness: Critical tasks can be executed without waiting for a long-running task to finish.

Fairness: No single task can block the CPU for an extended period, preventing starvation of other tasks.

Real-time capabilities: Preemptive kernels are more suitable for real-time systems where timely execution of tasks is essential.

- **Disadvantages of preemptive kernels:**

Higher overhead: Frequent task switches introduce context switching overhead, which can affect performance.

Complexity: Managing concurrent execution requires more complex synchronization mechanisms to prevent data corruption.

2.Non-preemptive Kernel:

A non-preemptive (or cooperative) kernel, also known as a cooperative multitasking system, relies on tasks or processes voluntarily yielding control of the CPU to allow other tasks to execute. In this model, a running task must explicitly relinquish control, usually by making system calls or waiting for specific events. If a task does not yield, it can potentially block other tasks from running.

- **Advantages of non-preemptive kernels:**

Simplicity: The absence of frequent task switches simplifies the management of tasks.

Lower overhead: Context switches are less frequent, reducing the associated overhead.

- **Disadvantages of non-preemptive kernels:**

Lack of fairness: A misbehaving or long-running task can monopolize the CPU, leading to poor system responsiveness.

Limited multitasking: The system's ability to handle multiple tasks efficiently is constrained by the willingness of tasks to yield control.

❖ MUTUAL EXCLUSION WITH BUSY WAITING

Mutual exclusion with busy waiting is a synchronization technique that ensures that only one process at a time can access a shared resource. It works by having each process repeatedly check if the resource is available. If it is, the process acquires the resource and uses it. If it is not, the process busy waits, which means that it continuously checks if the resource is available.

Busy waiting is a simple and efficient way to implement mutual exclusion, but it can waste CPU time if the resource is not available for a long time.

Here is an example of how mutual exclusion with busy waiting can be implemented:

```
shared int resource = 0;

void process1() {
  while (resource != 0) {
  }
  // Acquire resource
  resource = 1;
  // Use resource
  // Release resource
  resource = 0;
}

void process2() {
  while (resource != 0) {
  }
  // Acquire resource
  resource = 1;
  // Use resource
  // Release resource
  resource = 0;
}
```

In this example, the resource variable is shared between the two processes. The process1() function checks if the resource is available. If it is, the function acquires the resource and uses it. If it is not, the function busy waits until the resource becomes available. The process2() function does the same thing.

This implementation of mutual exclusion with busy waiting ensures that only one process at a time can access the resource variable. However, it can waste CPU time if the resource is not available for a long time.There are other ways to implement mutual exclusion, such as using semaphores or mutexes. These techniques are more efficient than busy waiting, but they are also more complex to implement

❖ SLEEP AND WAKEUP

Sleep and Wakeup are two system calls that are used to implement mutual exclusion in operating systems.

Sleep is a system call that causes the calling process to block until it is woken up by another process.

Wakeup is a system call that wakes up one or more blocked processes.

Sleep and Wakeup are often used to implement the producer-consumer problem, which is a classic problem in concurrent programming. In the producer-consumer problem, there is a shared buffer that is used to store data. The producer process puts data into the buffer, and the consumer process takes data out of the buffer.

To ensure that the producer and consumer processes do not access the buffer at the same time, we can use Sleep and Wakeup. The producer process will sleep if the buffer is full, and the consumer process will sleep if the buffer is empty. The producer process will wake up the consumer process when it puts data into the buffer, and the consumer process will wake up the producer process when it takes data out of the buffer.

Sleep and Wakeup can also be used to implement other synchronization problems, such as the readers-writers problem and the dining philosophers problem.

- **Sleep**

The `sleep()` system call takes a time interval as an argument. The calling process will block until the time interval has passed. If the time interval is zero, the calling process will block indefinitely.

The `sleep()` system call is typically implemented using a timer. The timer is set to the specified time interval, and the calling process is blocked. When the timer expires, the calling process is unblocked and can continue running.

- **Wakeup**

The `wakeup()` system call wakes up one or more blocked processes. The calling process does not block.

The `wakeup()` system call is typically implemented by decrementing the count of blocked processes on a semaphore or mutex. When the count of blocked processes reaches zero, the first blocked process is unblocked.

Types of Sleep

There are two types of sleep:

Absolute sleep is a sleep that is specified by an absolute time. The calling process will sleep until the specified time has passed.

Relative sleep is a sleep that is specified by a relative time. The calling process will sleep for the specified amount of time, starting from the current time.

Types of Wakeup

There are also two types of wakeup:

- Broadcast wakeup wakes up all blocked processes.
- Selective wakeup wakes up a specific blocked process.

➢ **Advantages of Sleep and Wakeup**

Sleep and Wakeup are a simple and efficient way to implement mutual exclusion. They are also relatively easy to understand and use.

➢ **Disadvantages of Sleep and Wakeup**

Sleep and Wakeup can waste CPU time if the processes are blocked for a long time. They can also be difficult to use correctly, and it is possible to deadlock the system

➢ Sleep and Wakeup are a powerful tool for implementing mutual exclusion in operating systems. However, they should be used with care to avoid wasting CPU time and deadlocking the system.

❖ SEMAPHORES

Semaphores are integer variables that are used to solve the critical section problem by using two atomic operations, wait and signal that are used for process synchronization.

Semaphores are used to coordinate the access to shared resources in a multi-process or multi-threaded environment, preventing race conditions and ensuring that critical sections of code are executed in a controlled manner. They help maintain the order and synchronization of processes or threads that are trying to access the same resource concurrently. Here's how semaphores work and how they can be used:

➢ **Working of Semaphores:**

Initialization: A semaphore is initialized with an integer value, usually indicating the number of available instances of a resource. For binary semaphores (mutexes), the initial value is often set to 1.

Wait (P) Operation: When a process or thread wants to access a shared resource, it first performs a "wait" operation on the semaphore associated with that resource. If the semaphore's value is greater than zero, the process is allowed to proceed, and the semaphore's value is decremented. If the semaphore's value is zero or negative, the process is blocked (suspended) until the semaphore's value becomes non-negative.

Signal (V) Operation: After a process or thread finishes using the shared resource, it performs a "signal" operation on the semaphore. This operation increments the semaphore's value. If there are other processes or threads waiting on the semaphore, one of them is allowed to proceed.

The definitions of wait and signal are as follows −

- **Wait**

The wait operation decrements the value of its argument S, if it is positive. If S is negative or zero, then no operation is performed.

```
wait(S)
{
  while (S<=0);


  S--;
}
```

- **Signal**

The signal operation increments the value of its argument S.

```
signal(S)
{
  S++;
}
```

➢ **Types of Semaphores**

There are two main types of semaphores i.e. counting semaphores and binary semaphores. Details about these are given as follows −

1.Counting Semaphores

These are integer value semaphores and have an unrestricted value domain. These semaphores are used to coordinate the resource access, where the semaphore count is the number of available resources. If the resources are added, semaphore count automatically incremented and if the resources are removed, the count is decremented.

2.Binary Semaphores

The binary semaphores are like counting semaphores but their value is restricted to 0 and 1. The wait operation only works when the semaphore is 1 and the signal operation succeeds when semaphore is 0. It is sometimes easier to implement binary semaphores than counting semaphores.

➢ **Using Semaphores:**

Mutual Exclusion (Binary Semaphore): One common use of semaphores is to enforce mutual exclusion, ensuring that only one process or thread can access a critical section of code at a time. This is typically achieved using a binary semaphore (mutex). When a process wants to enter a critical section, it performs a wait operation on the mutex. If the mutex is available (its value is 1), the process enters the critical section by locking the mutex (setting its value to 0). When the process is done, it releases the mutex by performing a signal operation, allowing other processes to enter.

Resource Management (Counting Semaphore): Counting semaphores are used to manage resources that have multiple instances. For example, consider a scenario where a limited number of database connections are available. A counting semaphore can be used to keep track of the available connections. When a process wants to use a connection, it performs a wait operation on the semaphore. If there are available connections, it can proceed. When the process is done, it signals the semaphore, releasing the connection for other processes to use.

Producer-Consumer Problem: Semaphores can be used to solve synchronization problems like the producer-consumer problem. In this scenario, multiple producer threads are adding data to a shared buffer, while multiple consumer threads are removing data from the buffer. Semaphores are used to ensure that producers and consumers don't access the buffer simultaneously, avoiding issues like overwriting data or accessing empty spaces.

Reader-Writer Problem: Semaphores can also be used to solve the reader-writer problem, where multiple threads need to read a shared resource, but only one thread can write to it at a time. Semaphores help control the access patterns to maintain data consistency.

It's important to note that while semaphores are powerful synchronization tools, they require careful design and management to avoid potential issues like deadlocks (where processes wait indefinitely) or resource contention. Modern programming languages and libraries often provide higher-level abstractions that help manage concurrency and synchronization more safely, but understanding the underlying principles of semaphores remains valuable for handling complex synchronization scenarios.

➢ Advantages of Semaphores

Some of the advantages of semaphores are as follows −

- Semaphores allow only one process into the critical section. They follow the mutual exclusion principle strictly and are much more efficient than some other methods of synchronization.
- There is no resource wastage because of busy waiting in semaphores as processor time is not wasted unnecessarily to check if a condition is fulfilled to allow a process to access the critical section.
- Semaphores are implemented in the machine independent code of the microkernel. So they are machine independent.

➢ Disadvantages of Semaphores

Some of the disadvantages of semaphores are as follows −

- Semaphores are complicated so the wait and signal operations must be implemented in the correct order to prevent deadlocks.
- Semaphores are impractical for last scale use as their use leads to loss of modularity. This happens because the wait and signal operations prevent the creation of a structured layout for the system.
- Semaphores may lead to a priority inversion where low priority processes may access the critical section first and high priority processes later.

❖ MUTEXES

The terms mutex and semaphore have always created chaos among scholars. These are among the important concepts of an operating system and play a vital role in achieving process synchronization in a multiprogramming environment.

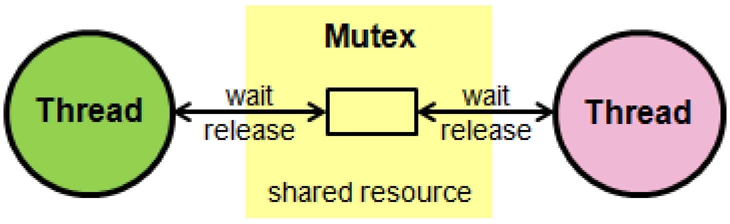
Before moving on to mutex, first, let's get familiar with a few terminologies:

1.Process Synchronization- In a multiprogramming environment, where multiple processes are present, all the processes need to be synchronized so that incorrect output is not produced.

2.Race Condition- When two or more processes try to access the same shared variable, there is a possibility that it may produce an incorrect output which depends on the order in which access takes place.

3.Critical Section- A region that contains the shared variables.

A mutex is a binary variable used to provide a locking mechanism. It offers mutual exclusion to a section of code that restricts only one thread to work on a code section at a given time.



Working of Mutex

- Suppose a thread executes a code and has locked that region using a mutex.
- If the scheduler decides to perform context switching, all the threads ready to execute in the same region are unblocked.
- Out of all the threads available, only one will make it to the execution, but if it tries to access the piece of code already locked by the mutex, then this thread will go to sleep.
- Context switching will take place again and again, but no thread will be able to access the region that has been locked until the lock is released.
- Only the thread that has locked the region can unlock it as well.

This is how mutex ensures the mutual exclusion of threads in a multiprogramming environment.

Consider the given example:

```
int a;
mutex_lock  lock_a; //lock variable which should reside in shared memory
void thread()
{
  mutex_lock(&lock_a);
  //Some work in the thread
  &mutex_unlock(&lock_a);
}
```
Here, in the above example, we have defined a mutex lock for the thread function. Firstly, the thread will acquire a mutex lock when entering the critical section after completion of the work the thread will be unlocked and exit the critical section.

Deadlock in Mutex

In a multiprogramming environment where two or more processes are executed, deadlock can occur using mutex during process synchronization. Deadlock occurs when threads are holding each other locks and are waiting for each other to unlock first. The four conditions for deadlock are:

Mutual exclusion: There must exist at least one resource which can be used by only one resource.

No preemption: The thread must release the resource by itself. Another thread can't snatch it.

Hold and wait: There must exist a resource holding a resource and waiting for another resource to be released.

Circular wait: All the processes must wait in a circular pattern.

Let's try to understand deadlock in mutex through an example:

```
//thread1
void thread1()
{
   lock(&mutex1);
   lock(&mutex2);
   // Some work in the thread
   unlock(&mutex2);
   unlock(&mutex1);
}

//thread2
void thread2()
{
   lock(&mutex2);
   lock(&mutex1);
   // Some work in the thread
   unlock(&mutex1);
   unlock(&mutex2);
}
```
In the above example, there are two threads and two mutexes. This example will work smoothly if both the mutex lock the threads first, finish their job, and unlock them after completion.

But in a multiprogramming environment, both the thread function can start simultaneously, and thread1 may acquire mutex1, and thread2 acquires mutex2, respectively. Then both go into infinite wait for the other thread to unlock the mutexes first. Thus, creating a deadlock condition.

Advantages of Mutex

- Since there is only one thread present in the critical section at a given time, there are no race conditions, and the data always remains consistent.
- Only one thread will be able to access the critical section
- Solves the race condition problem

Disadvantages of Mutex

- If a thread is preempted or goes to sleep after obtaining a lock, then it may cause other threads to enter a state of starvation as they may not move forward.
- It cannot be locked or unlocked other than the one thread that has acquired it.
- It may lead to a busy waiting state, which wastes CPU time as the thread will check again and again for the lock to get released, hence wasting a lot of system calls.

❖ MONITOR

What is a monitor in OS?

Monitors are a programming language component that aids in the regulation of shared data access. The Monitor is a package that contains shared data structures, operations, and synchronization between concurrent procedure calls. Therefore, a monitor is also known as a synchronization tool. Java, C#, Visual Basic, Ada, and concurrent Euclid are among some of the languages that allow the use of monitors. Processes operating outside the monitor can't access the monitor's internal variables, but they can call the monitor's procedures.

For example, synchronization methods like the wait() and notify() constructs are available in the Java programming language.

Syntax of monitor in OS

Monitor in os has a simple syntax similar to how we define a class, it is as follows:

```
Monitor monitorName{
     variables_declaration;
     condition_variables;
    procedure p1{ ... };
    procedure p2{ ... };
    ...
    procedure pn{ ... };

    {
       initializing_code;
    }

}
```

Monitor in an operating system is simply a class containing variable_declarations, condition_variables, various procedures (functions), and an initializing_code block that is used for process synchronization.

Characteristics of Monitors in OS

A monitor in os has the following characteristics:

- We can only run one program at a time inside the monitor.
- Monitors in an operating system are defined as a group of methods and fields that are combined with a special type of package in the os.
- A program cannot access the monitor's internal variable if it is running outside the monitor. Although, a program can call the monitor's functions.
- Monitors were created to make synchronization problems less complicated.
- Monitors provide a high level of synchronization between processes.

Components of Monitor in an operating system

The monitor is made up of four primary parts:

Initialization: The code for initialization is included in the package, and we just need it once when creating the monitors.

Private Data: It is a feature of the monitor in an operating system to make the data private. It holds all of the monitor's secret data, which includes private functions that may only be utilized within the monitor. As a result, private fields and functions are not visible outside of the monitor.

Monitor Procedure: Procedures or functions that can be invoked from outside of the monitor are known as monitor procedures.

Monitor Entry Queue: Another important component of the monitor is the Monitor Entry Queue. It contains all of the threads, which are commonly referred to as procedures only.

Condition Variables

There are two sorts of operations we can perform on the monitor's condition variables:

1.Wait

2.Signal

Consider a condition variable (y) is declared in the monitor:

y.wait(): The activity/process that applies the wait operation on a condition variable will be suspended, and the suspended process is located in the condition variable's block queue.

y.signal(): If an activity/process applies the signal action on the condition variable, then one of the blocked activity/processes in the monitor is given a chance to execute.

Advantages of Monitor in OS

- Monitors offer the benefit of making concurrent or parallel programming easier and less error-prone than semaphore-based solutions.
- It helps in process synchronization in the operating system.
- Monitors have built-in mutual exclusion.
- Monitors are easier to set up than semaphores.
- Monitors may be able to correct for the timing faults that semaphores cause.

Disadvantages of Monitor in OS

- Monitors must be implemented with the programming language.
- Monitor increases the compiler's workload.
- The monitor requires to understand what operating system features are available for controlling crucial sections in the parallel procedures.

❖ What is message passing technique in OS?

Message Passing provides a mechanism to allow processes to communicate and to synchronize their actions without sharing the same address space.

For example − chat programs on World Wide Web.

Now let us discuss the message passing step by step.

Step 1 − Message passing provides two operations which are as follows −

- Send message
- Receive message
- Messages sent by a process can be either fixed or variable size.

Step 2 − For fixed size messages the system level implementation is straight forward. It makes the task of programming more difficult.

Step 3 − The variable sized messages require a more system level implementation but the programming task becomes simpler.

Step 4 − If process P1 and P2 want to communicate they need to send a message to and receive a message from each other that means here a communication link exists between them.

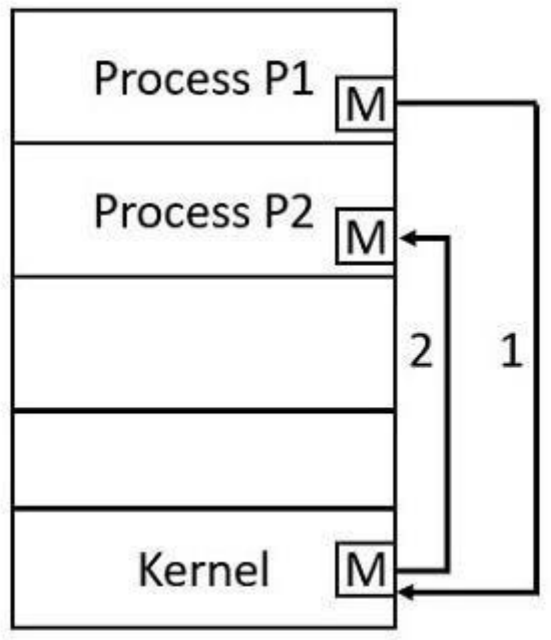
Step 5 − Methods for logically implementing a link and the send() and receive() operations**.**

Characteristics

The characteristics of Message passing model are as follows −

- Mainly the message passing is used for communication.
- It is used in distributed environments where the communicating processes are present on remote machines which are connected with the help of a network.

- Here no code is required because the message passing facility provides a mechanism for communication and synchronization of actions that are performed by the communicating processes.



Message Passing System

- Message passing is a time consuming process because it is implemented through kernel (system calls).
- It is useful for sharing small amounts of data so that conflicts need not occur.
- In message passing the communication is slower when compared to shared memory technique.

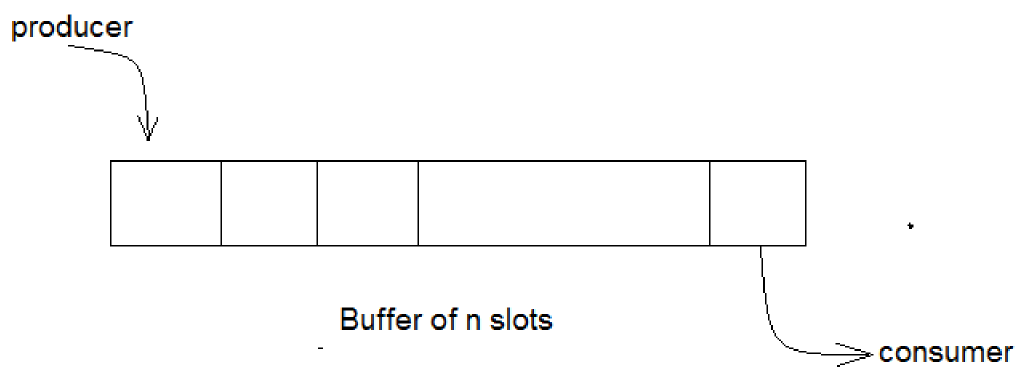
➢ <u>Classic Problems of Synchronization</u>

➢ Bounded Buffer Problem

Bounded buffer problem, which is also called producer consumer problem, is one of the classic problems of synchronization. Let's start by understanding the problem here, before moving on to the solution and program code.

➢ What is the Problem Statement?

There is a buffer of n slots and each slot is capable of storing one unit of data. There are two processes running, namely, producer and consumer, which are operating on the buffer.



➢ Bounded Buffer Problem

A producer tries to insert data into an empty slot of the buffer. A consumer tries to remove data from a filled slot in the buffer. As you might have guessed by now, those two processes won't produce the expected output if they are being executed concurrently.

There needs to be a way to make the producer and consumer work in an independent manner.

➢ Here's a Solution

One solution of this problem is to use semaphores. The semaphores which will be used here are:

- m, a **binary semaphore** which is used to acquire and release the lock.
- empty, a **counting semaphore** whose initial value is the number of slots in the buffer, since, initially all slots are empty.
- full, a **counting semaphore** whose initial value is 0.
- At any instant, the current value of empty represents the number of empty slots in the buffer and full represents the number of occupied slots in the buffer.

➢ The Producer Operation

The pseudocode of the producer function looks like this:

```
do
{
// wait until empty > 0 and then decrement 'empty'
wait(empty);
// acquire lock
wait(mutex);
/* perform the insert operation in a slot */
// release lock
signal(mutex);
// increment 'full'
signal(full);
}while(TRUE);
```

- it decrements the empty semaphore because, there will now be one less empty slot, since the producer is going to insert data in one of those slots.
- Then, it acquires lock on the buffer, so that the consumer cannot access the buffer until producer completes its operation.

• After performing the insert operation, the lock is released and the value of full IS incremented because the producer has just filled a slot in the buffer.

- **The Consumer Operation**

The pseudocode for the consumer function looks like this:

```
do
{
 // wait until full > 0 and then decrement 'full'
 wait(full);
 // acquire the lock
 wait(mutex);
 /* perform the remove operation in a slot */
 // release the lock
 signal(mutex);
 // increment 'empty'
 signal(empty);
}
while(TRUE);
```

- The consumer waits until there is at least one full slot in the buffer.
- Then it decrements the full semaphore because the number of occupied slots will be decreased by one, after the consumer completes its operation.
- After that, the consumer acquires lock on the buffer.
- Following that, the consumer completes the removal operation so that the data from one of the full slots is removed.Then, the consumer releases the lock.
- Finally, the empty semaphore is incremented by 1, because the consumer has just Removed data from an occupied slot, thus making it empty.

- **What is Readers Writer Problem?**

Readers writer problem is another example of a classic synchronization problem. There are many variants of this problem, one of which is examined below.

The Problem Statement

There is a shared resource which should be accessed by multiple processes. There are two types of processes in this context. They are reader and writer. Any number of readers can read from the shared resource simultaneously, but only one writer can write to the shared resource. When a writeris writing data to the resource, no other process can access the resource. A writer cannot write to the resource if there are non-zero number of readers accessing the resource at that time.

The Solution

From the above problem statement, it is evident that readers have higher priority than writer. If a writer wants to write to the resource, it must wait until there are no readers currently accessing that resource.

Here, we use one mutex m and a semaphore w. An integer variable read_count is used tomaintain the number of readers currently accessing the resource. The variable read_count is Initialized to 0. A value of 1 is given initially to m and w.Instead of having the process to acquire lock on the shared resource, we use the mutex m to make the process to acquire and release lock whenever it is updating the read_count variable.

The code for the writer process looks like this:

```
while(TRUE)
{
 wait(w);
 /* perform the write operation */
 signal(w);
}
```

And, the code for the reader process looks like this:

```
while(TRUE)
{
 //acquire lock
 wait(m);
 read_count++;
 if(read_count == 1)
 wait(w);
 //release lock
 signal(m);
 /* perform the reading operation */
 // acquire lock
 wait(m);
 read_count--;
 if(read_count == 0)
 signal(w);
 // release lock
 signal(m);
}
```

Here is the Code uncoded(explained)

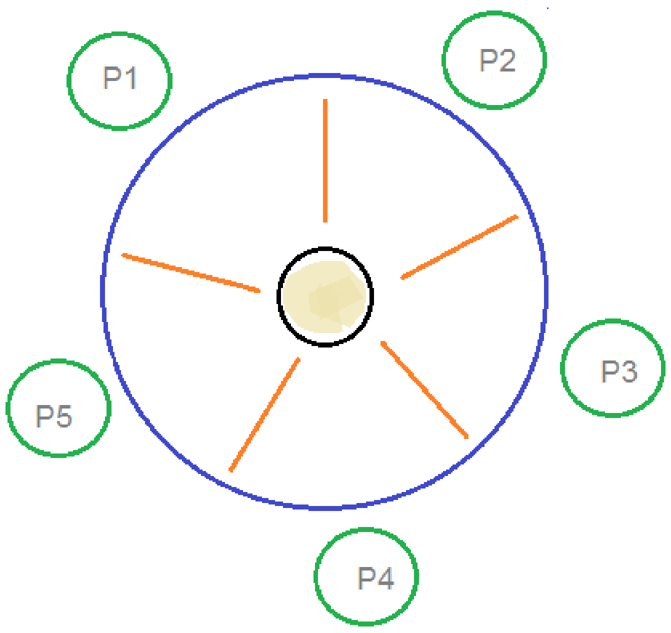
- As seen above in the code for the writer, the writer just waits on the w semaphore until it gets a chance to write to the resource.
- After performing the write operation, it increments w so that the next writer can access resource.
- On the other hand, in the code for the reader, the lock is acquired whenever the read_count is updated by a process.
- When a reader wants to access the resource, first it increments the read_count value, then accesses the resource and then decrements the read_count value.
- The semaphore w is used by the first reader which enters the critical section and the last reader which exits the critical section.
- The reason for this is, when the first readers enters the critical section, the writer is blocked from the resource. Only new readers can access the resource now.
- Similarly, when the last reader exits the critical section, it signals the writer using the w semaphore because there are zero readers now and a writer can have the chance to access the resource.

❖ Dining Philosophers Problem

The dining philosophers problem is another classic synchronization problem which is used to evaluate situations where there is a need of allocating multiple resources to multiple processes.

What is the Problem Statement?

Consider there are five philosophers sitting around a circular dining table. The dining table has five chopsticks and a bowl of rice in the middle as shown in the below figure



Dining Philosophers Problem

At any instant, a philosopher is either eating or thinking. When a philosopher wants to eat, he uses two chopsticks - one from their left and one from their right. When a philosopher wants to think, he keeps down both chopsticks at their original place.

➢ Here's the Solution

From the problem statement, it is clear that a philosopher can think for an indefinite amount of time. But when a philosopher starts eating, he has to stop at some point of time. The philosopher is in an endless cycle of thinking and eating.

An array of five semaphores, stick[5], for each of the five chopsticks.

The code for each philosopher looks like:

```
while(TRUE)

{

 wait(stick[i]);
 /*
 mod is used because if i=5, next
 chopstick is 1 (dining table is circular)
 */
 wait(stick[(i+1) % 5]);
 /* eat */
 signal(stick[i]);
 signal(stick[(i+1) % 5]);
 /* think */
}
```

When a philosopher wants to eat the rice, he will wait for the chopstick at his left and picks up that chopstick. Then he waits for the right chopstick to be available, and then picks it too. After eating, he puts both the chopsticks down.

But if all five philosophers are hungry simultaneously, and each of them pickup one chopstick, then a deadlock situation occurs because they will be waiting for another chopstick forever.

The possible solutions for this are:

• A philosopher must be allowed to pick up the chopsticks only if both the left and right chopsticks are available.

• Allow only four philosophers to sit at the table. That way, if all the four philosophers pick up four chopsticks, there will be one chopstick left on the table. So, one philosopher can start eating and eventually, two chopsticks will be available. In this way, deadlocks can be avoided