NEW L J INSTITUTE OF ENGINEERING & TECHNOLOGY

SEMESTER- 4

BRANCH- CSE/CSE[AIML]

SUBJECT NAME: OPERATING SYSTEM

SUBJECT CODE: 3140702

GTU SYLLABUS

Sr. No.	Content	% Weightage
1	**Introduction:** Computer system overview, Architecture, Goals & Structures of O.S, Basic functions, Interaction of O.S. & hardware architecture, System calls, Batch, multiprogramming. Multitasking, time sharing, parallel, distributed & realtime O.S.	10
2	**Process and Threads Management:** Process Concept, Process states, Process control, Threads, Uni-processor Scheduling: Types of scheduling: Preemptive, Non preemptive, Scheduling algorithms: FCFS, SJF, RR, Priority, Thread Scheduling, Real Time Scheduling. System calls like ps, fork, join, exec family, wait.	15
3	**Concurrency:** Principles of Concurrency, Mutual Exclusion: S/W approaches, H/W Support, Semaphores, Pipes, Message Passing, Signals, Monitors.	8
4	**Inter Process Communication:** Race Conditions, Critical Section, Mutual Exclusion, Hardware Solution, Strict Alternation, Peterson's Solution, The Producer Consumer Problem, Semaphores, Event Counters, Monitors, Message Passing, Classical IPC Problems: Reader's & Writer Problem, Dinning Philosopher Problem etc., Scheduling, Scheduling Algorithms.	15
5	**Deadlock:** Principles of Deadlock, Starvation, Deadlock Prevention, Deadlock Avoidance, Deadlock Detection, System calls	8
6	**Memory Management:** Memory Management requirements, Memory partitioning: Fixed and Variable Partitioning, Memory Allocation: Allocation Strategies (First Fit, Best Fit, and Worst Fit), Swapping, Paging and Fragmentation. Demand Paging, Security Issues. Virtual Memory: Concepts, VM management, Page Replacement Policies (FIFO, LRU, Optimal, Other Strategies), Thrashing.	15
7	**I/O Management & Disk scheduling:** I/O Devices, Organization of I/O functions, Operating System Design issues, I/O Buffering, Disk Scheduling (FCFS, SCAN, C-SCAN, SSTF), RAID, Disk Cache.	10
8	**Security & Protection:** Security Environment, Design Principles Of Security, User Authentication, Protection Mechanism : Protection Domain, Access Control List	7
9	**Unix/Linux Operating System:** Development Of Unix/Linux, Role & Function Of Kernel, System Calls, Elementary Linux command & Shell Programming, Directory Structure, System Administration Case study: Linux, Windows Operating System	7
10	**Virtualization Concepts:** Virtual machines; supporting multiple operating systems simultaneously on a single hardware platform; running one operating system on top of another. True or pure virtualization.	5

UNIT 3: CONCURRENCY
UNIT4:INTER PROCESS COMMUNICATION

Unit 3 Unit 4	**Concurrency:** Principles of Concurrency, Mutual Exclusion: S/W approaches, H/W Support, Semaphores, Pipes, Message Passing, Signals, Monitors. **Inter Process Communication:** Race Conditions, Critical Section, Mutual Exclusion, Hardware Solution, Strict Alternation, Peterson's Solution, The Producer Consumer Problem, Semaphores, Event Counters, Monitors, Message Passing, Classical IPC Problems: Reader's & Writer Problem, Dinning Philosopher Problem etc., Scheduling, Scheduling Algorithms.	
	TOPIC 1: Principles of Concurrency	
	Defination: • It refers to the execution of multiple instruction sequences at the same time. It occurs in an operating system when multiple process threads are executing concurrently. These threads can interact with one another via shared memory or message passing. • Concurrency results in resource sharing, which causes issues like deadlocks and resource scarcity. It aids with techniques such as process coordination, memory allocation, and execution schedule to maximize throughput.	
	Question: List four design issues for which the concept of concurrency is relevant. **solution:** **Principles of Concurrency** The principles of concurrency in operating systems are designed to ensure that multiple processes or threads can execute efficiently and effectively, without interfering with each other or causing deadlock. • **Interleaving** − Interleaving refers to the interleaved execution of multiple processes or threads. The operating system uses a scheduler to determine which process or thread to execute at any given time. Interleaving allows for efficient use of CPU resources and ensures that all processes or threads get a fair share of CPU time. • **Synchronization** − Synchronization refers to the coordination of multiple processes or threads to ensure that they do not interfere with each other. This is done through the use of synchronization primitives such as locks, semaphores, and monitors. These primitives allow processes or threads to coordinate access to shared resources such as memory and I/O devices. • **Mutual exclusion** − Mutual exclusion refers to the principle of ensuring that only one process or thread can access a shared resource at a time. This is typically implemented using locks or semaphores to ensure that multiple processes or threads do not access a shared resource simultaneously. • **Deadlock avoidance** − Deadlock is a situation in which two or more processes or threads are waiting for each other to release a resource, resulting in a deadlock. Operating systems use various techniques such as resource allocation graphs and deadlock prevention algorithms to avoid deadlock. • **Process or thread coordination** − Processes or threads may need to coordinate their activities to achieve a common goal. This is typically achieved using synchronization primitives such as semaphores or message passing mechanisms such as pipes or sockets. • **Resource allocation** − Operating systems must allocate resources such as memory, CPU time, and I/O devices to multiple processes or threads in a fair and efficient manner. This is typically achieved using scheduling algorithms such as round-robin, priority-based, or real-time scheduling. **Advantages of concurrency in Operating System**	

Concurrency provides several advantages in operating systems, including −

- **Improved performance** − Concurrency allows multiple tasks to be executed simultaneously, improving the overall performance of the system. By using multiple processors or threads, tasks can be executed in parallel, reducing the overall processing time.
- **Resource utilization** − Concurrency allows better utilization of system resources, such as CPU, memory, and I/O devices. By allowing multiple tasks to run simultaneously, the system can make better use of its available resources.
- **Responsiveness** − Concurrency can improve system responsiveness by allowing multiple tasks to be executed concurrently. This is particularly important in real-time systems and interactive applications, such as gaming and multimedia.
- **Scalability** − Concurrency can improve the scalability of the system by allowing it to handle an increasing number of tasks and users without degrading performance.
- **Fault tolerance** − Concurrency can improve the fault tolerance of the system by allowing tasks to be executed independently. If one task fails, it does not affect the execution of other tasks.

Question: Define Mutual Exclusion, Critical Section and Race Condition.

solution:

1.Race Condition

It occurs when two or more operations occur in an undefined manner. When two or more processes are reading or writing some shared data and the final result depends on who runs precisely

Example : shared =5

PROCESS P1	PROCESS P2
Int x=shared; x++ Sleep(1); Shared=x	Int x=shared; y++ Sleep(1); Shared=y

- The two process p1 and p2 run one by one
- After completion the value of shared in p1 is 6 and p2 is 4
- If p1 get pre-empt by any condition and process p2 execute first and then p1 resume the value of shared become 4
- Similarly as p2 preempt and p1 run the value of shared become 6 in p2

It show the race condituion

Question:Explain Race Condition regarding banking problem.

solution:

It should be avoided because they can cause fine error in applications

Example : in banking system .after depositing the amount ,it update by bank employee balance=balance+amount.at the same time, some amount is transfer then it became balance=balance-amount. These two tasks are in a race to write variable balance. In this the process that updates last determines the final value of balance

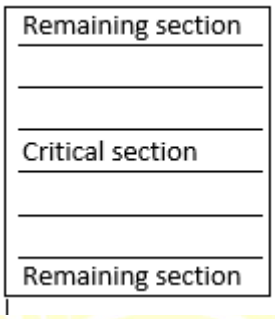
Design and management issue for concurrency are as follow:

1. Track of various processes is kept by operating system.

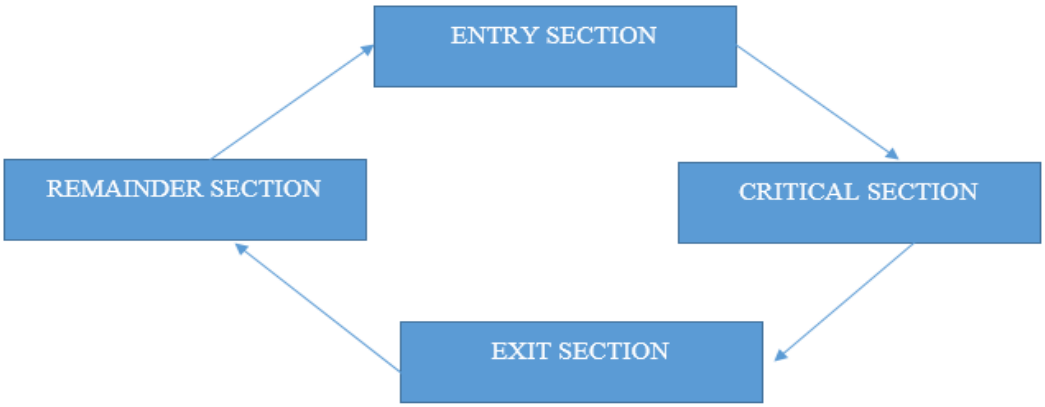
2. OS allocates and deallocate software and hardware resources to active process
3. OS must protect user data and physical resources from un authorized process
4. Process execution speed do not depends upon other process execution speed.

2.Critical Section Problem

The critical section is a code segment where the shared variables can be accessed. An atomic action is required in a critical section i.e. only one process can execute in its critical section at a time. All the other processes have to wait to execute in their critical sections.



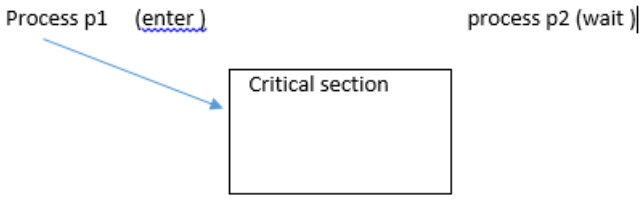
It ensure only one process at a time is allowed to be operating in critical section.
When process is accessing shared modificiable data ,it is said to be in critical section
The structure are as follow:

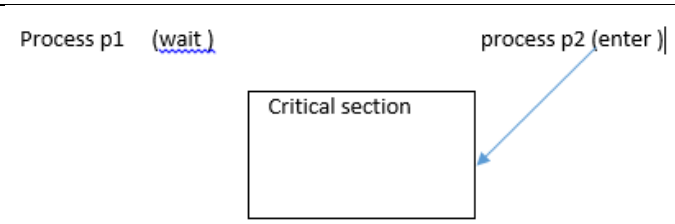


- Entry Section: it is block of code executed in preparation for enetering critical section
- Exit Section: the code executed upon leaving the critical section.
- Remainder Section: Rest of the code is in section
-

SYNCRONIZATION MECHANISMS

1. MUTUAL EXCLUSION
   If p1 execute critical section at same time p2 want to access critical section , mutual exclusion doesn't allow the critical section to be shared at the same time

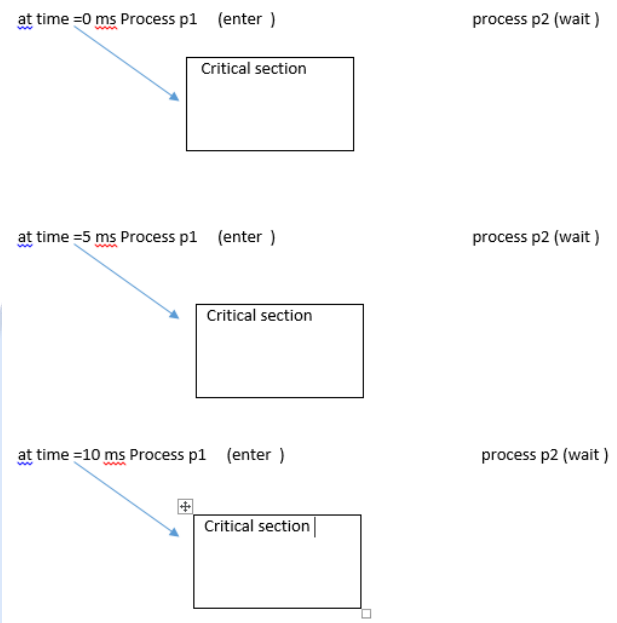




2. PROGRESS

When one process again and again use critical section and the process p2 still wait for critical section and led to starvation

Progress means each process get critical section simultaneously.



3.BOUNDED WAIT

if p1 enter critical section and didn't execute the exit code , for infinite time execute the critical section , this phenomena is knows as bounded wait

Two general approaches are used to handle critical sections:

Preemptive kernels: A preemptive kernel allows a process to be preempted while it is running in kernel mode.

Non preemptive kernels: A non preemptive kernel does not allow a process running in kernel mode to be preempted; a kernel-mode process will run until it exists in kernel mode, blocks, or voluntarily yields control of the CPU. A non preemptive kernel is essentially free from race conditions on kernel data structures, as only one process is active in the kernel at a time.

3.Critical Region

- In an operating system, a critical region refers to a section of code or a data structure that must be accessed exclusively by one method or thread at a time. Critical regions are utilized to prevent concurrent entry to shared sources, along with variables, information structures, or devices, that allow you to maintain information integrity and keep away from race conditions.
- The critical section cannot be executed by more than one process at the same time; operating system faces the difficulties in allowing and disallowing the processes from entering the critical section.
- The critical section problem is used to design a set of protocols which can ensure that the Race condition among the processes will never arise.

Question: explain the term Mutual Exclusion

Solution:

3.Mutual Exclusion:

Mutual Exclusion is a property of process synchronization that states that "no two processes can exist in the critical section at any given point of time". The term was first coined by Dijkstra. Any process synchronization technique being used must satisfy the property of mutual exclusion, without which it would not be possible to get rid of a race condition.

The need for mutual exclusion comes with concurrency. There are several kinds of concurrent execution:

- nterrupt handlers
- Interleaved, preemptively scheduled processes/threads
- Multiprocessor clusters, with shared memory
- Distributed systems

Mutual exclusion methods are used in concurrent programming to avoid the simultaneous use of a common resource, such as a global variable, by pieces of computer code called critical sections •

The requirement of mutual exclusion is that when process P1 is accessing a shared resource R1, another process should not be able to access resource R1 until process P1 has finished its operation with resource R1.

Examples of such resources include files, I/O devices such as printers, and shared data structures.

Conditions Required for Mutual Exclusion

- According to the following four criteria, mutual exclusion is applicable:
- When using shared resources, it is important to ensure mutual exclusion between various processes. There cannot be two processes running simultaneously in either of their critical sections.
- It is not advisable to make assumptions about the relative speeds of the unstable processes.
- For access to the critical section, a process that is outside of it must not obstruct another process.
- Its critical section must be accessible by multiple processes in a finite amount of time; multiple processes should never be kept waiting in an infinite loop.

Approaches To Implementing Mutual Exclusion

1. Software method: Leave the responsibility to the processes themselves. These methods are usually highly error-prone and carry high overheads.

2. Hardware method: Special-purpose machine instructions are used for accessing shared resources. This method is faster but cannot provide a complete solution. Hardware solutions cannot give guarantee the absence of deadlock and starvation.

3. Programming language method: Provide support through the operating system or through the programming language.

Requirements of Mutual Exclusion

1. At any time, only one process is allowed to enter its critical section.
2. The solution is implemented purely in software on a machine.
3. A process remains inside its critical section for a bounded time only.
4. No assumption can be made about the relative speeds of asynchronous concurrent processes.
5. A process cannot prevent any other process from entering into a critical section.

6. A process must not be indefinitely postponed from entering its critical section.

4.LOCK

This is the simplest synchronization mechanism. This is a Software Mechanism implemented in User mode. This is a busy waiting solution which can be used for more than two processes.

In this mechanism, a Lock variable **lock**is used. Two values of lock can be possible, either 0 or 1. Lock value 0 means that the critical section is vacant while the lock value 1 means that it is occupied.

A process which wants to get into the critical section first checks the value of the lock variable. If it is 0 then it sets the value of lock as 1 and enters into the critical section, otherwise it waits.

The pseudo code of the mechanism looks like following.

```
Entry Section →
While (lock! = 0);
Lock = 1;
//Critical Section
Exit Section →
Lock =0;
```

- If we look at the Pseudo Code, we find that there are three sections in the code. Entry Section, Critical Section and the exit section.
- Initially the value of lock variable is 0. The process which needs to get into the critical section, enters into the entry section and checks the condition provided in the while loop.
- The process will wait infinitely until the value of lock is 1 (that is implied by while loop). Since, at the very first time critical section is vacant hence the process will enter the critical section by setting the lock variable as 1.
- When the process exits from the critical section, then in the exit section, it reassigns the value of lock as 0.

Every Synchronization mechanism is judged on the basis of four conditions.

- Mutual Exclusion
- Progress
- Bounded Waiting
- Portability
- Out of the four parameters, Mutual Exclusion and Progress must be provided by any solution.

5.MUTEX

Mutex is a specific kind of binary semaphore that is used to provide a locking mechanism. It stands for Mutual Exclusion Object. Mutex is mainly used to provide mutual exclusion to a specific portion of the code so that the process can execute and work with a particular section of the code at a particular time.

Mutex uses a priority inheritance mechanism to avoid priority inversion issues. The priority inheritance mechanism keeps higher-priority processes in the blocked state for the minimum possible time. However, this cannot avoid the priority inversion problem, but it can reduce its effect up to an extent.

Advantages of Mutex

- No race condition arises, as only one process is in the critical section at a time.
- Data remains consistent and it helps in maintaining integrity.

- It's a simple locking mechanism that can be obtained by a process before entering into a critical section and released while leaving the critical section.

Swap() :

- Instead of directly setting lock to true in the swap function, key is set to true and then swapped with lock. First process will be executed, and in while(key), since key=true , swap will take place and hence lock=true and key=false. Again next iteration takes place while(key) but key=false , so while loop breaks and first process will enter in critical section.
- Now another process will try to enter in Critical section, so again key=true and hence while(key) loop will run and swap takes place so, lock=true and key=true (since lock=true in first process). Again on next iteration while(key) is true so this will keep on executing and another process will not be able to enter in critical section.
- Therefore Mutual exclusion is ensured. Again, out of the critical section, lock is changed to false, so any process finding it gets t enter the critical section. Progress is ensured. However, again bounded waiting is not ensured for the very same reason.

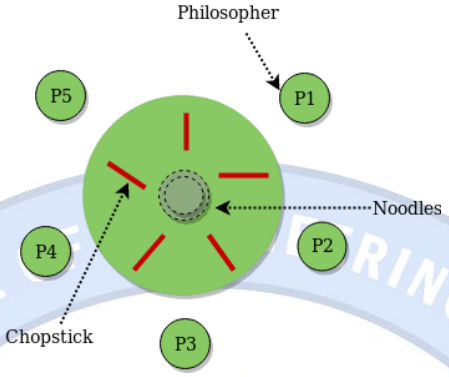
```
// Shared variable lock initialized to false
// and individual key initialized to false;

boolean lock;
Individual key;

void swap(boolean &a, boolean &b){
   boolean temp = a;
   a = b;
   b = temp;
}

while (1){
   key = true;
   while(key)
      swap(lock,key);
   critical section
   lock = false;
   remainder section
}
```

Question: Explain Dining philosopher problem and its solution using semaphore.

solution:

2.DINNING PHILPSOPHER PROBLEM USING SEMAPHORE

The Dining Philosopher Problem states that K philosophers are seated around a circular table with one chopstick between each pair of philosophers. There is one chopstick between each philosopher. A philosopher may eat if he can pick up the two chopsticks adjacent to him. One chopstick may be picked up by any one of its adjacent followers but not both.



1. There are three states of the philosopher: **THINKING, HUNGRY, and EATING**.
2. Here there are two semaphores: Mutex and a semaphore array for the philosophers.
3. Mutex is used such that no two philosophers may access the pickup or put it down at the same time.
4. The array is used to control the behavior of each philosopher. But, semaphores can result in deadlock due to programming errors.

The Dining Philosopher Problem is a classic synchronization problem in computer science that involves

1. multiple processes (philosophers) sharing a limited set of resources (forks) in order to perform a task (eating).
2. In order to avoid deadlock or starvation, a solution must be implemented that ensures that each philosopher can access the resources they need to perform their task without interference from other philosophers.

solution

to the Dining Philosopher Problem uses semaphores, a synchronization mechanism that can be used to control access to shared resources. In this solution, each fork is represented by a semaphore, and a philosopher must acquire both the semaphore for the fork to their left and the semaphore for the fork to their right before they can begin eating. If a philosopher cannot acquire both semaphores, they must wait until they become available.

The steps for the Dining Philosopher Problem solution using semaphores are as follows

1. Initialize the semaphores for each fork to 1 (indicating that they are available).
2. Initialize a binary semaphore (mutex) to 1 to ensure that only one philosopher can attempt to pick up a fork at a time.
3. For each philosopher process, create a separate thread that executes the following code:
   - While true:
   - Think for a random amount of time.
   - Acquire the mutex semaphore to ensure that only one philosopher can attempt to pick up a fork at a time.
   - Attempt to acquire the semaphore for the fork to the left.
   - If successful, attempt to acquire the semaphore for the fork to the right.
   - If both forks are acquired successfully, eat for a random amount of time and then release both semaphores.

- If not successful in acquiring both forks, release the semaphore for the fork to the left (if acquired) and then release the mutex semaphore and go back to thinking.

4. Run the philosopher threads concurrently.

By using semaphores to control access to the forks, the Dining Philosopher Problem can be solved in a way that avoids deadlock and starvation. The use of the mutex semaphore ensures that only one philosopher can attempt to pick up a fork at a time, while the use of the fork semaphores ensures that a philosopher can only eat if both forks are available.

Some of the solutions include the following:

1. The maximum number of philosophers at the table should not exceed four; in this case, philosopher P3 will have access to chopstick C4; he will then begin eating, and when he is finished, he will put down both chopsticks C3 and C4; as a result, semaphore C3 and C4 will now be incremented to 1.
2. Now that philosopher P2, who was holding chopstick C2, will also have chopstick C3, he will do the same when finished eating, allowing other philosophers to eat.
3. While a philosopher in an odd position should select the right chopstick first, a philosopher in an even position should select the left chopstick and then the right chopstick.
4. A philosopher should only be permitted to choose their chopsticks if both of the available chopsticks (the left and the right) are available at the same time.
5. All four of the initial philosophers (P0, P1, P2, and P3) should choose the left chopstick before choosing the right, while P4 should choose the right chopstick before choosing the left. As a result, P4 will be forced to hold his right chopstick first because his right chopstick, C0, is already being held by philosopher P0 and its value is set to 0. As a result, P4 will become trapped in an infinite loop and chopstick C4 will remain empty.
6. As a result, philosopher P3 has both the left C3 and the right C4. chopstick available, therefore it will start eating and will put down both chopsticks once finishes and let others eat which removes the problem of deadlock.

ALGORITHM

```
Void Philosopher
 {
 while(1)
  {
   take_chopstick[i];
   take_chopstick[ (i+1) % 5] ;
   . .
   . EATING THE NOODLE
   .
.  put_chopstick[i] );
.  put_chopstick[ (i+1) % 5] ;
.  .
.  . THINKING
.  }
.}
```

- Suppose Philosopher P0 wants to eat, it will enter in Philosopher() function, and execute take_chopstick[i]; by doing this it holds C0 chopstick after that it execute take_chopstick[(i+1) % 5]; by doing this it holds C1 chopstick(since i =0, therefore (0 + 1) % 5 = 1)
- Similarly suppose now Philosopher P1 wants to eat, it will enter in Philosopher() function, and execute take_chopstick[i]; by doing this it holds C1 chopstick after that it execute take_chopstick[(i+1) % 5]; by doing this it holds C2 chopstick(since i =1, therefore (1 + 1) % 5 = 2)
- But Practically Chopstick C1 is not available as it has already been taken by philosopher P0, hence the above code generates problems and produces race condition.

```
void Philosopher
 {
 while(1)
  {
   Wait( take_chopstickC[i] );
   Wait( take_chopstickC[(i+1) % 5] ) ;
   . .
   . EATING THE NOODLE
   .
.  Signal( put_chopstickC[i] );
.  Signal( put_chopstickC[ (i+1) % 5] ) ;
.  .
.  . THINKING
.  }
.}
```

- the above code, first wait operation is performed on take_chopstickC[i] and take_chopstickC [(i+1) % 5]. This shows philosopher i have picked up the chopsticks from its left and right. The eating function is performed after that.
- On completion of eating by philosopher i the, signal operation is performed on take_chopstickC[i] and take_chopstickC [(i+1) % 5]. This shows that the philosopher i have eaten and put down both the left and right chopsticks. Finally, the philosopher starts thinking again.
- Let's understand how the above code is giving a solution to the dining philosopher problem?
- Let value of i = 0(initial value), Suppose Philosopher P0 wants to eat, it will enter in Philosopher() function, and execute **Wait(take_chopstickC[i]);** by doing this it holds **C0 chopstick** and reduces semaphore C0 to 0**,** after that it execute **Wait(take_chopstickC[(i+1) % 5]);** by doing this it holds **C1 chopstick**(since i =0, therefore (0 + 1) % 5 = 1) and reduces semaphore C1 to 0
- Similarly, suppose now Philosopher P1 wants to eat, it will enter in Philosopher() function, and execute **Wait(take_chopstickC[i]);** by doing this it will try to hold **C1 chopstick** but will not be able to do that**,** since the value of semaphore C1 has already been set to 0 by philosopher P0, therefore it will enter into an infinite loop because of which philosopher P1 will not be able to pick chopstick C1 whereas if Philosopher P2 wants to eat, it will enter in Philosopher() function, and

execute **Wait(take_chopstickC[i]);** by doing this it holds **C2 chopstick** and reduces semaphore C2 to 0, after that, it executes **Wait(take_chopstickC[(i+1) % 5]);** by doing this it holds **C3 chopstick**(since i =2, therefore (2 + 1) % 5 = 3) and reduces semaphore C3 to 0.

- Hence the above code is providing a solution to the dining philosopher problem, A philosopher can only eat if both immediate left and right chopsticks of the philosopher are available else philosopher needs to wait. Also at one go two independent philosophers can eat simultaneously (i.e., philosopher **P0 and P2, P1 and P3 & P2 and P4** can eat simultaneously as all are the independent processes and they are following the above constraint of dining philosopher problem)

Question: What is Semaphore? Give the implementation of Readers-Writers Problem using semaphore

solution:

3.READER WRITER PROBLEM USING SEMAPHORE

Consider a situation where we have a file shared between many people.

- If one of the person tries editing the file, no other person should be reading or writing at the same time, otherwise changes will not be visible to him/her.
- However, if some person is reading the file, then others may read it at the same time.

Precisely in OS, we call this situation the readers-writer, problem

Problem parameters: that

- One set of data is shared among a number of processes
- Once a writer is ready, it performs its write. Only one writer may write at a time
- If a process is writing, no other process can read it
- If at least one reader is reading, no other process can write
- Readers may not write and only read

Solution when Reader has the Priority over Writer

There are four types of cases that could happen here.

Case	Process 1	Process 2	Allowed/Not Allowed
Case 1	Writing	Writing	Not Allowed
Case 2	Writing	Reading	Not Allowed
Case 3	Reading	Writing	Not Allowed
Case 4	Reading	Reading	Allowed

Here priority means, no reader should wait if the share is currently opened for reading.

Three variables are used: mutex, wrt, readcnt to implement solution

1. semaphore mutex, wrt; // semaphore mutex is used to ensure mutual exclusion when readcnt is updated i.e. when any reader enters or exit from the critical section and semaphore wrt is used by both readers and writers
2. int readcnt; // readcnt tells the number of processes performing read in the critical section, initially 0

Functions for semaphore :

– wait() : decrements the semaphore value.

– signal() : increments the semaphore value.

Writer process:
1. Writer requests the entry to critical section.
2. If allowed i.e. wait() gives a true value, it enters and performs the write. If not allowed, it keeps on waiting.
3. It exits the critical section.

```
do {
   // writer requests for critical section
   wait(wrt);

   // performs the write

   // leaves the critical section
   signal(wrt);

} while(true);
```

Reader process:
1. Reader requests the entry to critical section.
2. If allowed:
   - it increments the count of number of readers inside the critical section. If this reader is the first reader entering, it locks the wrt semaphore to restrict the entry of writers if any reader is inside.
   - It then, signals mutex as any other reader is allowed to enter while others are already reading.
   - After performing reading, it exits the critical section. When exiting, it checks if no more reader is inside, it signals the semaphore "wrt" as now, writer can enter the critical section.
3. If not allowed, it keeps on waiting.

```
do {
  // Reader wants to enter the critical section
  wait(mutex);

  // The number of readers has now increased by 1
  readcnt++;

  // there is atleast one reader in the critical section
  // this ensure no writer can enter if there is even one reader
  // thus we give preference to readers here
```

```
  if (readcnt==1)
    wait(wrt);

  // other readers can enter while this current reader is inside
  // the critical section
  signal(mutex);

  // current reader performs reading here
  wait(mutex);   // a reader wants to leave

  readcnt--;

  // that is, no reader is left in the critical section,
  if (readcnt == 0)
     signal(wrt);        // writers can enter

  signal(mutex); // reader leaves

} while(true);
```

Thus, the semaphore 'wrt' is queued on both readers and writers in a manner such that preference is given to readers if writers are also there.

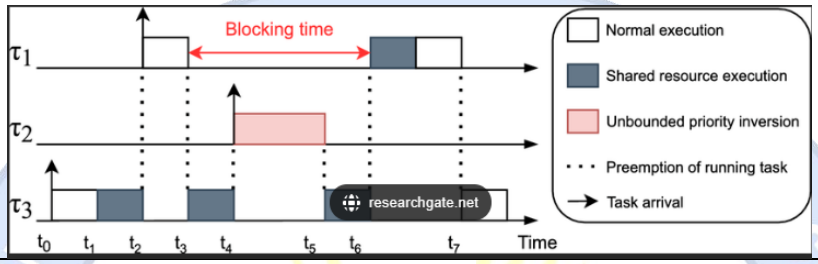
Question: What is the priority inversion problem in inter process communication? How to Solve it with semaphore?

solution:

Priority inversion

- In Operating System, one of the important concepts is Task Scheduling. There are several Scheduling methods such as First Come First Serve, Round Robin, Priority-based scheduling, etc. Each scheduling method has its pros and cons. As you might have guessed, Priority Inversion comes under Priority-based Scheduling. Basically, it's a problem which arises sometimes when Priority-based scheduling is used by OS. In Priority-based scheduling, different tasks are given different priorities so that higher priority tasks can intervene in lower priority tasks if possible.
- So, in priority-based scheduling, if a lower priority task (L) is running and if a higher priority task (H) also needs to run, the lower priority task (L) would be preempted by a higher priority task (H). Now, suppose both lower and higher priority tasks need to share a common resource (say access to the same file or device) to achieve their respective work. In this case, since there are resource sharing and task synchronization is needed, several methods/techniques can be used for handling such scenarios. For sake of our topic on Priority Inversion, let us mention a synchronization method say mutex. Just to recap on the mutex, a task acquires mutex before entering the critical section (CS) and releases mutex after exiting the critical section (CS). While running in CS, tasks access this common resource. Now, say both L and H share a common Critical Section (CS) i.e. the same mutex is needed for this CS.

- Coming to our discussion of priority inversion, let us examine some scenarios.
  1) L is running but not in CS; H needs to run; H preempts L; H starts running; H relinquishes or releases control; L resumes and starts running
  2) L is running in CS; H needs to run but not in CS; H preempts L; H starts running; H relinquishes control; L resumes and starts running.
  3) L is running in CS; H also needs to run in CS; H waits for L to come out of CS; L comes out of CS; H enters CS and starts running



Monitor :

- Monitors are a higher-level synchronization construct that simplifies process synchronization by providing a high-level abstraction for data access and synchronization. Monitors are implemented as programming language constructs, typically in object-oriented languages, and provide mutual exclusion, condition variables, and data encapsulation in a single construct.
- A monitor is essentially a module that encapsulates a shared resource and provides access to that resource through a set of procedures. The procedures provided by a monitor ensure that only one process can access the shared resource at any given time, and that processes waiting for the resource are suspended until it becomes available.
- Monitors are used to simplify the implementation of concurrent programs by providing a higher-level abstraction that hides the details of synchronization. Monitors provide a structured way of sharing data and synchronization information, and eliminate the need for complex synchronization primitives such as semaphores and locks.
- The key advantage of using monitors for process synchronization is that they provide a simple, high-level abstraction that can be used to implement complex concurrent systems. Monitors also ensure that synchronization is encapsulated within the module, making it easier to reason about the correctness of the system.
- monitors have some limitations. For example, they can be less efficient than lower-level synchronization primitives such as semaphores and locks, as they may involve additional overhead due to their higher-level abstraction. Additionally, monitors may not be suitable for all types of synchronization problems, and in some cases, lower-level primitives may be required for optimal performance.
- The monitor is one of the ways to achieve Process synchronization. The monitor is supported by programming languages to achieve mutual exclusion between processes. For example Java Synchronized methods. Java provides wait() and notify() constructs.
- It is the collection of condition variables and procedures combined together in a special kind of module or a package.
- The processes running outside the monitor can't access the internal variable of the monitor but can call procedures of the monitor.

- Only one process at a time can execute code inside monitors.

```
Monitor Demo // Name of the Monitor
{
variables;
condition variables;
procedure p1 {.....}
procedure p2 {.....}
}
```

Condition Variables: Two different operations are performed on the condition variables of the monitor.

Wait.

signal.

Wait Operation

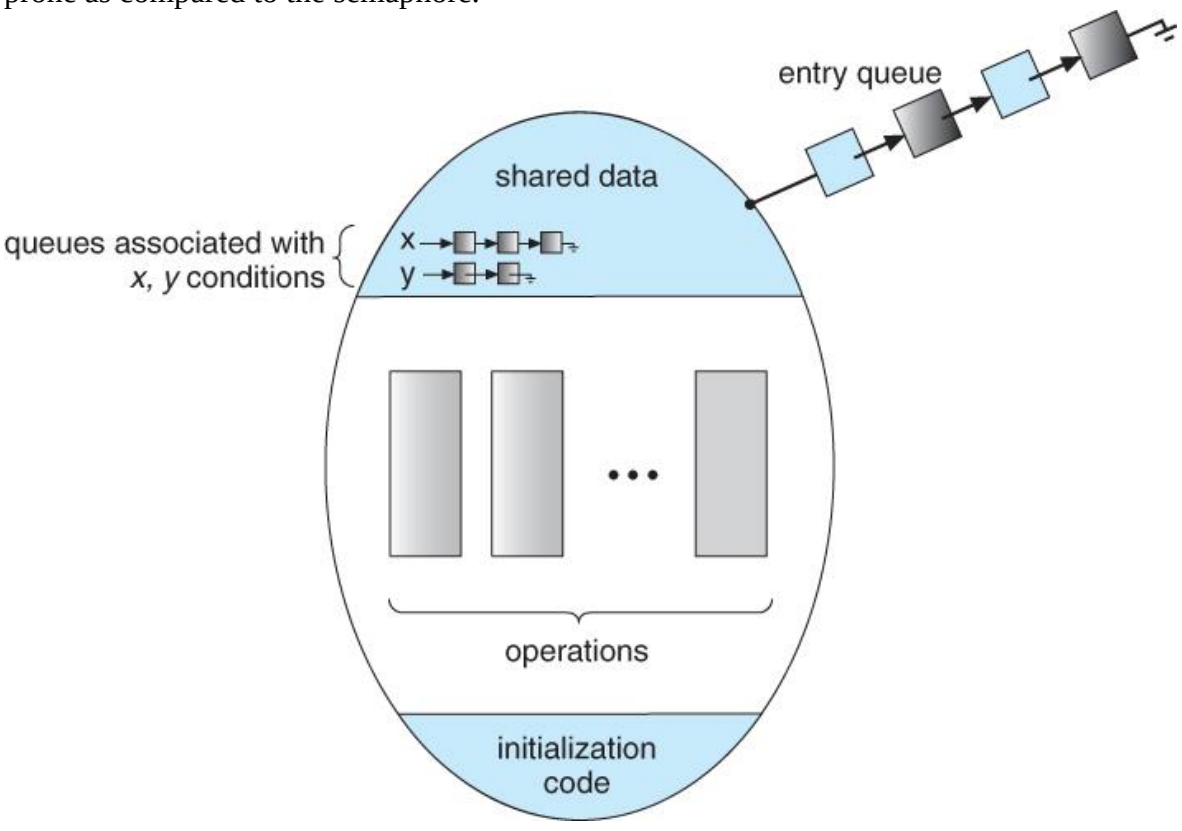
a.wait(): - The process that performs wait operation on the condition variables are suspended and locate the suspended process in a block queue of that condition variable.

Signal Operation

a.signal() : - If a signal operation is performed by the process on the condition variable, then a chance is provided to one of the blocked processes.

Advantages of Monitor

It makes the parallel programming easy, and if monitors are used, then there is less error-prone as compared to the semaphore.



Difference between Monitors and Semaphore

Monitors	Semaphore
We can use condition variables only in the monitors.	In semaphore, we can use condition variables anywhere in the program, but we cannot use conditions variables in a semaphore.
In monitors, wait always block the caller.	In semaphore, wait does not always block the caller.
The monitors are comprised of the shared variables and the procedures which operate the shared variable.	The semaphore S value means the number of shared resources that are present in the system.
Condition variables are present in the monitor.	Condition variables are not present in the semaphore.

Question: **Write Solution to Dining-Philosopher problem using monitor.**

- We demonstrate monitor ideas by proposing a **deadlock-free solution** to the Dining-Philosophers problem. The monitor is used to control access to state variables and condition variables. It only notifies when to enter and exit the segment.
- This approach imposes the limitation that a philosopher may only take up her forks if both the forks are available.
- To code this solution, we must distinguish between three situations where we could see a philosopher. We introduce the following data structure for this purpose:
- **Thinking**: When the philosopher does not want to use either fork.
- **Hungry**: When a philosopher wishes to use the forks, i.e., he wants to go to the critical section.
- **Eating:** When the philosopher has both forks, i.e., he has entered the critical section.
- Philosopher i may set the variable state[i] = **Eating**, Only if her two neighbours are not eating,i.e., (state[(i+4)%5]!= **Eating**) and (state[(i+1)%5]!= **Eating**).

```
// Dining-Philosophers Solution Using Monitors
monitor DiningPhilosophers
{
   status state[5];
   condition self[5];

   // Picking up forks
   void Pick(int i)
   {
      // indicating HUNGRY state
      state[i] = HUNGRY;

      // setting state to EATING in test()
      // only if the left and right neighbors if current       //philosopher are not EATING
      test(i);
```

```
        // if unable to eat,then waiting to be signaled
        if (state[i] != EATING)
            self[i].wait;
    }

   // Putting down the forks
    void Put(int i)
    {

        // indicating THINKING state
        state[i] = THINKING;

        // if right neighbor R=(i+1)%5 is HUNGRY and
        // both of R's neighbors are not EATING,
        // then setting the R's state to EATING and waking it up            // by signaling R's C
V
        test((i + 1) % 5);
        test((i + 4) % 5);
    }

    void test(int i)
    {

        if (state[(i + 1) % 5] != EATING && state[(i + 4) % 5] != EATING && state[i] == H
UNGRY)
        {

            // indicating EATING state
            state[i] = EATING;

            // signal() has no effect during Pick(),but is //important to wake up waiting HUNG
RY philosophers during Put()
            self[i].signal();
        }
    }

    initialization_code()
    {

        // Execution of Pick(), Put() and test() are all mutually //exclusive, i.e. only one at a ti
me can be executed
        for (int i = 0; i < 5; i++)
            state[i] = THINKING;
        // Verifying that this monitor-based solution is deadlock //free and mutually exclusive
in that no 2 neighbors can eat //simultaneously
    }
} // end of monitor
```

Question: Explain producer consumer problem and its solution using monitor.

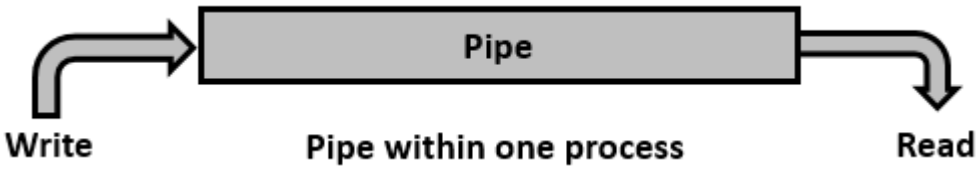
```
monitor ProducerConsumer
  condition full, empty;
  int count;

 procedure enter();
 {
  if (count == N) wait(full);       // if buffer is full, block
  put_item(widget);                 // put item in buffer
  count = count + 1;                // increment count of full slots
  if (count == 1)  signal(empty);   // if buffer was empty, wake consumer
 }

 procedure remove();
 {
  if (count == 0)  wait(empty);     // if buffer is empty, block
  remove_item(widget);              // remove item from buffer
  count = count - 1;                // decrement count of full slots
  if (count == N-1)  signal(full);  // if buffer was full, wake producer
 }

 count = 0;
 end monitor;

 Producer();
 {
  while (TRUE)
  {
    make_item(widget);              // make a new item
    ProducerConsumer.enter;         // call enter function in monitor
  }
 }

 Consumer();
 {
  while (TRUE)
  {
    ProducerConsumer.remove;        // call remove function in monitor
    consume_item;                   // consume an item
  }
 }
```

Question: Explain pipes and signal in reference with unix concurrency mechanism.

solution:

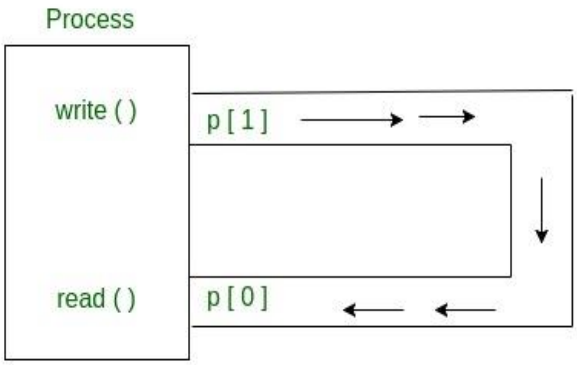
PIPES

- Pipe is a communication medium between two or more related or interrelated processes.
- It can be either within one process or a communication between the child and the parent processes.
- Communication can also be multi-level such as communication between the parent, the child and the grand-child, etc.
- Communication is achieved by one process writing into the pipe and other reading from the pipe.
- To achieve the pipe system call, create two files, one to write into the file and another to read from the file.
- Pipe mechanism can be viewed with a real-time scenario such as filling water with the pipe into some container, say a bucket, and someone retrieving it, say with a mug. The filling process is nothing but writing into the pipe and the reading process is nothing but retrieving from the pipe. This implies that one output (water) is input for the other (bucket).



Pipe within one process

- Pipes are useful for communication between related processes(inter-process communication).
- Pipe is one-way communication only i.e we can use a pipe such that One process write to the pipe, and the other process reads from the pipe. It opens a pipe, which is an area of main memory that is treated as a "virtual file".
- The pipe can be used by the creating process, as well as all its child processes, for reading and writing. One process can write to this "virtual file" or pipe and another related process can read from it.
- If a process tries to read before something is written to the pipe, the process is suspended until something is written.
- The pipe system call finds the first two available positions in the process's open file table and allocates them for the read and write ends of the pipe.



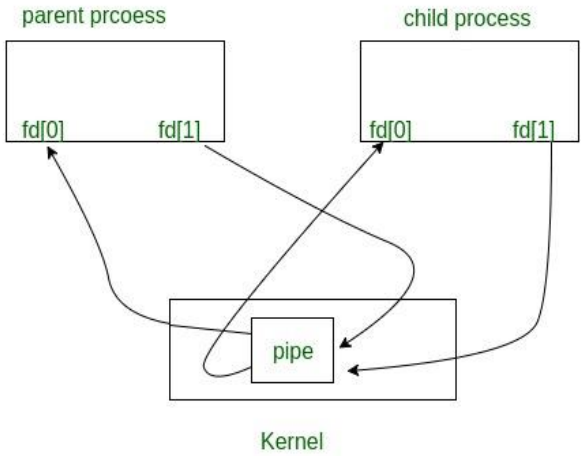
Syntax in C language:
int pipe(int fds[2]);

Parameters :
fd[0] will be the fd(file descriptor) for the read end of pipe.
fd[1] will be the fd for the write end of pipe.
Returns : 0 on Success.

-1 on error.

Parent and child sharing a pipe

When we useforkin any process, file descriptors remain open across child process and also parent process. If we call fork after creating a pipe, then the parent and child can communicate via the pipe.



advantages:

- Simplicity: Pipes are a simple and straightforward way for processes to communicate with each other.
- Efficiency: Pipes are an efficient way for processes to communicate, as they can transfer data quickly and with minimal overhead.
- Reliability: Pipes are a reliable way for processes to communicate, as they can detect errors in data transmission and ensure that data is delivered correctly.
- Flexibility: Pipes can be used to implement various communication protocols, including one-way and two-way communication.

Disadvantages:

- Limited capacity: Pipes have a limited capacity, which can limit the amount of data that can be transferred between processes at once.
- Unidirectional: In a unidirectional pipe, only one process can send data at a time, which can be a disadvantage in some situations.
- Synchronization: In a bidirectional pipe, processes must be synchronized to ensure that data is transmitted in the correct order.
- Limited scalability: Pipes are limited to communication between a small number of processes on the same computer, which can be a disadvantage in large-scale distributed systems.

SIGNALS

Signals are standardized messages sent to a running program to trigger specific behavior, such as quitting or error handling. They are a limited form of inter-process communication (IPC), typically used in Unix, Unix-like, and other POSIX-compliant operating systems.

A signal is an asynchronous notification sent to a process or to a specific thread within the same process to notify it of an event. Common uses of signals are to interrupt, suspend, terminate or kill a process.

Whenever a signal is raised (either programmatically or system generated signal), a default action is performed. What if you don't want to perform the default action but wish to perform your own actions on receiving the signal? Is this possible for all the signals? Yes, it is possible to handle the signal but not for all the signals. What if you want to ignore the signals, is this possible? Yes, it is possible to ignore the signal. Ignoring the signal implies neither performing the default action nor handling the signal. It is possible to ignore or handle almost all the signals. The signals which can't be either ignored or handled/caught are SIGSTOP and SIGKILL.

In summary, the actions performed for the signals are as follows −

Default Action
Handle the signal
Ignore the signal

How Signals Work as IPC

1. Sending a Signal: One process can send a signal to another process using system calls like kill(pid, signal), where pid is the Process ID of the recipient and signal is the signal to be sent.
2. Receiving a Signal: Upon receiving a signal, a process has several options:
   - Execute a default action (like terminating for SIGTERM).
   - Ignore the signal.
   - Catch the signal with a custom handler function.
   - Mask/block certain signals.
3. Synchronization: Signals can be used to synchronize actions between processes. For instance, a parent process can pause execution using the pause() system call and await a signal from the child process.

Advantages and Limitations

Advantages:

- Lightweight: Signals provide a quick and efficient way to notify processes.
- Built-in Mechanism: No need for external libraries or tools.
- Limitations:
- Limited Payload: Signals carry limited information—basically just the signal type.
- Delivery Uncertainty: There's no guaranteed order of delivery if multiple signals are sent rapidly.
- Risk of Lost Signals: If multiple instances of the same signal are sent before being handled, they might be collapsed into a single instance.

Use Cases

1. Process Management: Parent processes can monitor child processes and get notified when they terminate (SIGCHLD).
2. Resource Alerts: Processes can be alerted to take specific actions when system resources are low.

3. User Notifications: Processes can be notified of user actions, like pressing Ctrl+C (SIGINT).

Question: What are the advantages of inter-process communication? How communication takes place in a shared-memory environment?

solution:

IPC(INTER PROCESS COMMUNICATION)
Process executing concurrently in the operating system may be either independent process and cooperating process.
Any process share data with other process is cooperating .
A process can be of two types:

- Independent process.
- Co-operating process.

reasons for providing an environment that allow process cooperation:

1. Information sharing: several users may be interested in the same piece of information for example a shared file, we must provide an environment to allow concurrent access to such information.
2. Computational speed up: break tasks in different sub tasks, each of which will be executing in parallel with the other in multiple processing cores
3. Modularity:dividing the system function into separate processesor threads
4. Convenience :user may be work on many task at the same time for example, a user may be editing ,listening to music are compilling in parallel

An independent process is not affected by the execution of other processes while a co-operating process can be affected by other executing processes. Though one can think that those processes, which are running independently, will execute very efficiently, in reality, there are many situations when co-operative nature can be utilized for increasing computational speed, convenience, and modularity. Inter-process communication (IPC) is a mechanism that allows processes to communicate with each other and synchronize their actions. The communication between these processes can be seen as a method of co-operation between them. Processes can communicate with each other through both:

Question: Define and explain Message Passing.

solution:

1. Shared Memory
2. Message passing

Figure 1 below shows a basic structure of communication between processes via the shared memory method and via the message passing method.

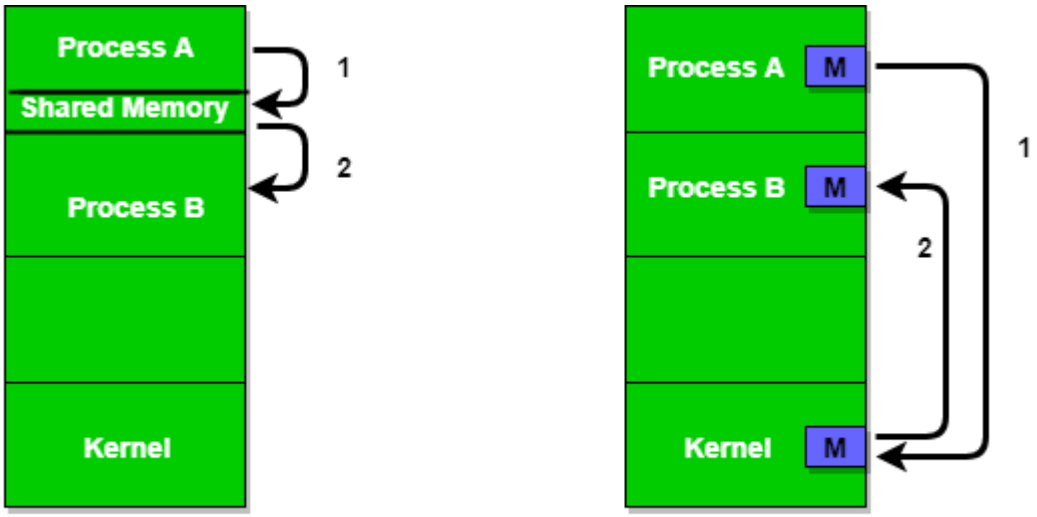


Figure 1 - Shared Memory and Message Passing

I)SHARED MEMORY

In shared memory model , a region of memory that is shared between by cooperating process

Process can exchange information by reading and writing data to the shared region

If one process write the other process can read the data from memory

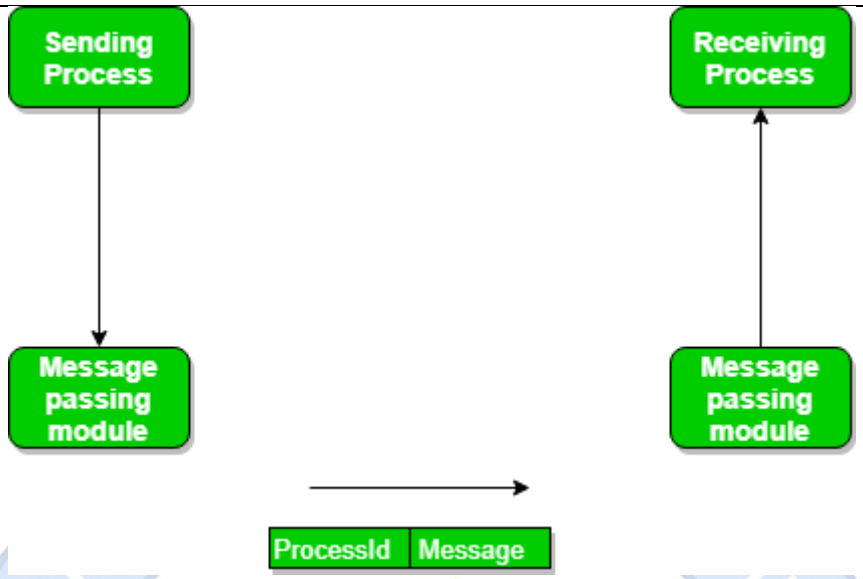
Two process p1 and p2

1) P1------write in memory ------p2 is blocked state -------p2 read
2) P2 ------write in memory-------p1 is blocked state-------p1 read

ii) Messaging Passing Method

Now, We will start our discussion of the communication between processes via message passing. In this method, processes communicate with each other without using any kind of shared memory. If two processes p1 and p2 want to communicate with each other, they proceed as follows:

- Establish a communication link (if a link already exists, no need to establish it again.)
- Start exchanging messages using basic primitives.
  We need at least two primitives:
  – send(message, destination) or send(message)
  – receive(message, host) or receive(message)
- The message size can be of fixed size or of variable size. If it is of fixed size, it is easy for an OS designer but complicated for a programmer and if it is of variable size then it is easy for a programmer but complicated for the OS designer. A standard message can have two parts: **header and body.**
  The **header part** is used for storing message type, destination id, source id, message length, and control information. The control information contains information like what to do if runs out of buffer space, sequence number, priority. Generally, message is sent using FIFO style.



Structure of message passing

	structure information			address	
Actual data or pointer	number of bytes	type	sequence number or ID	receiver	sender

Issue in message passing:

1. Sender and receiver identity
2. Acceptance system
3. Crash or link failure
4. Handling buffer
5. Order of delivery message