

Inter Process Communication

- Inter process communication is the mechanism provided by the operating system that allows processes to communicate with each other.
- Processes executing concurrently in the operating system.
- It is used for exchanging data between multiple threads in one or more processes or programs. The Processes may be running on single or multiple computers connected by a network.

- It is a set of programming interface which allow a programmer to coordinate activities among various program processes which can run concurrently in an operating system.
- This allows a specific program to handle many user requests at the same time.
- Since every single user request may result in multiple processes running in the operating system, the process may require to communicate with each other.
- Each IPC protocol approach has its own advantage and limitation, so it is not unusual for a single program to use all of the IPC methods.



- Processes is of 2 Types.

1. **Independent Processes** : They cannot affect or be affected by the other processes executing in the system.

2. **Cooperating Processes** : They can affect or be affected by the other processes executing in the system.

- Any process that shares data with other processes is a cooperating process.
- It requires an IPC mechanism that will allow them to exchange data and information.

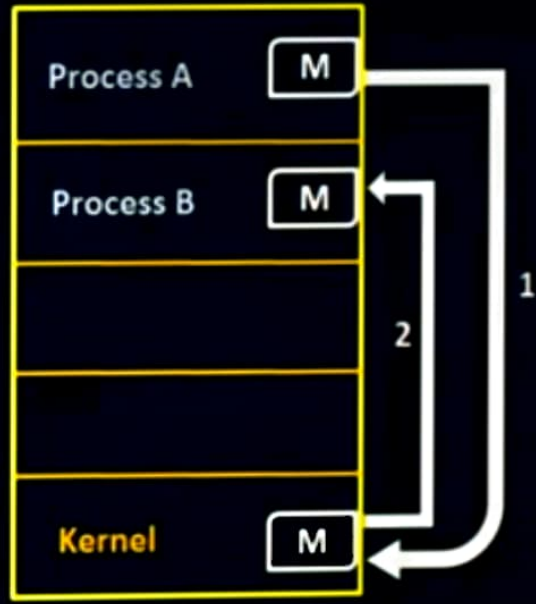
- There are Two fundamental models of IPC.

1. Shared Memory

2. Message Passing



(a)



(b)

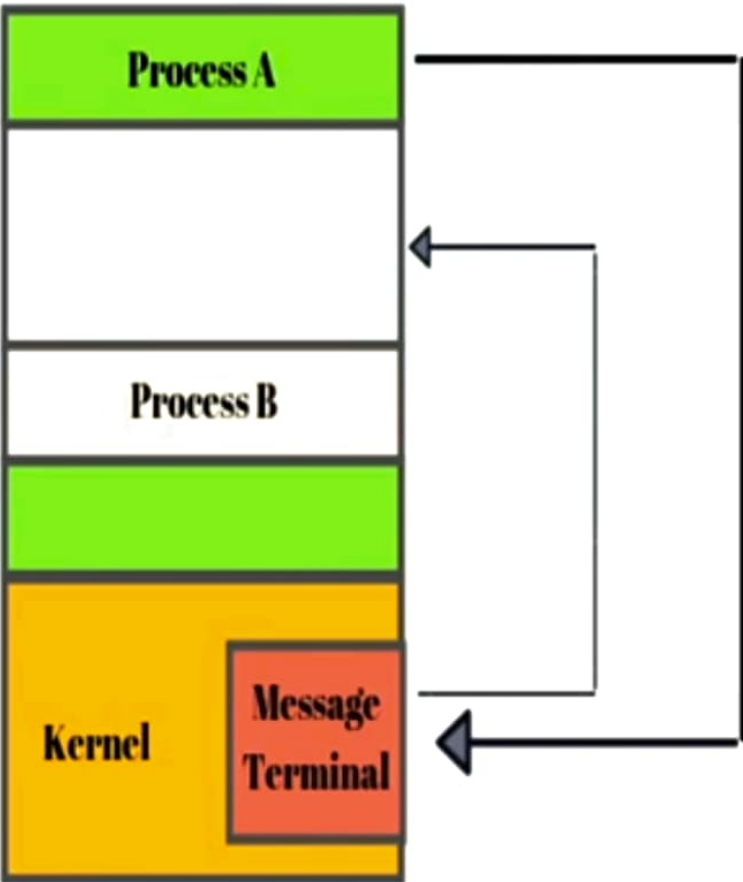
Fig: Communications models, (a) Shared memory, (b) Message Passing.

Shared Memory:

- 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.
- A shared memory region resides in the address space of the process creating the shared memory segment.
- Other processes that wish to communicate using the shared memory segment must attach it to their address space.

Message Passing :

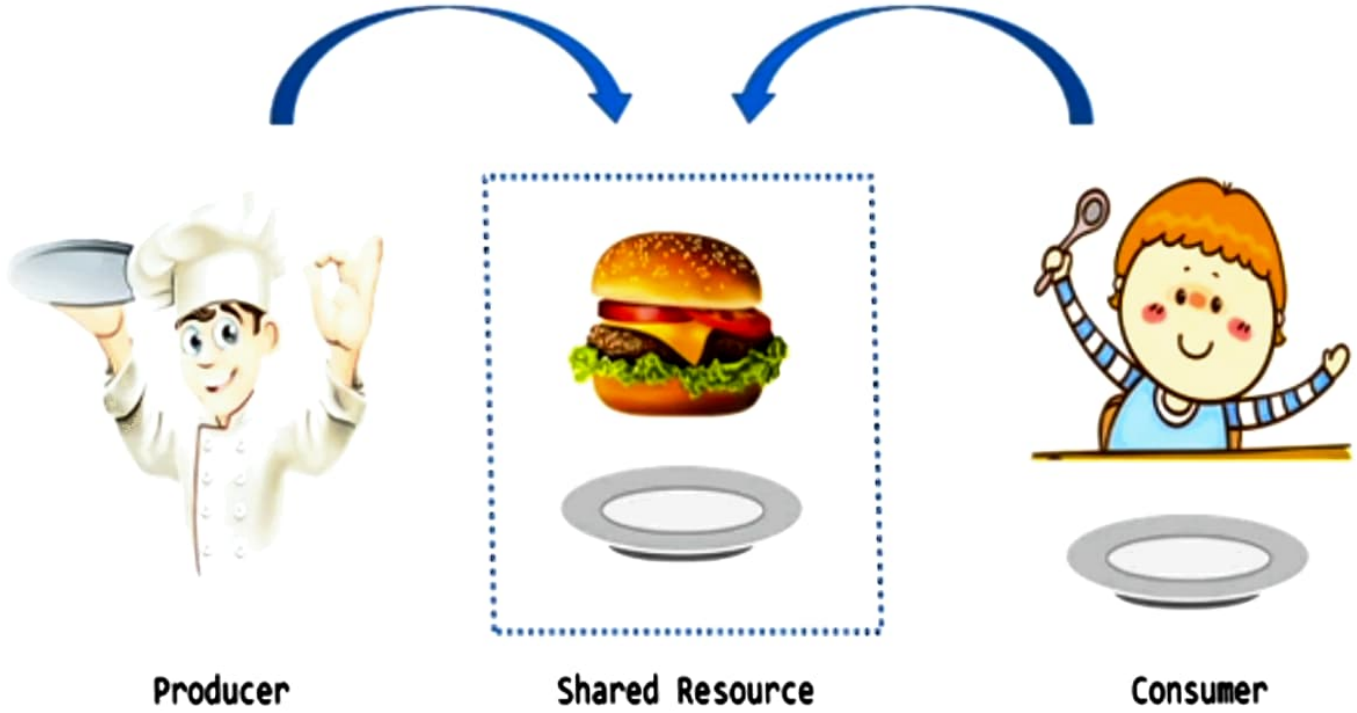
- Communication takes place by means of messages exchanged between the cooperating processes.
- It provides a mechanism to allow processes to communicate and to synchronize their actions without sharing the same address space.
- It is particularly useful in a distributed environment, where communicating processes may reside on different computers connected by a network.



Producer Consumer Problem

- A producer process produces information that is consumed by a consumer process.
- Eg: A compiler may produce assembly code, which is consumed by an assembler. The assembler in turn may produce object modules which are consumed by a loader.
- The solution to the producer consumer problem is shared memory and semaphores.

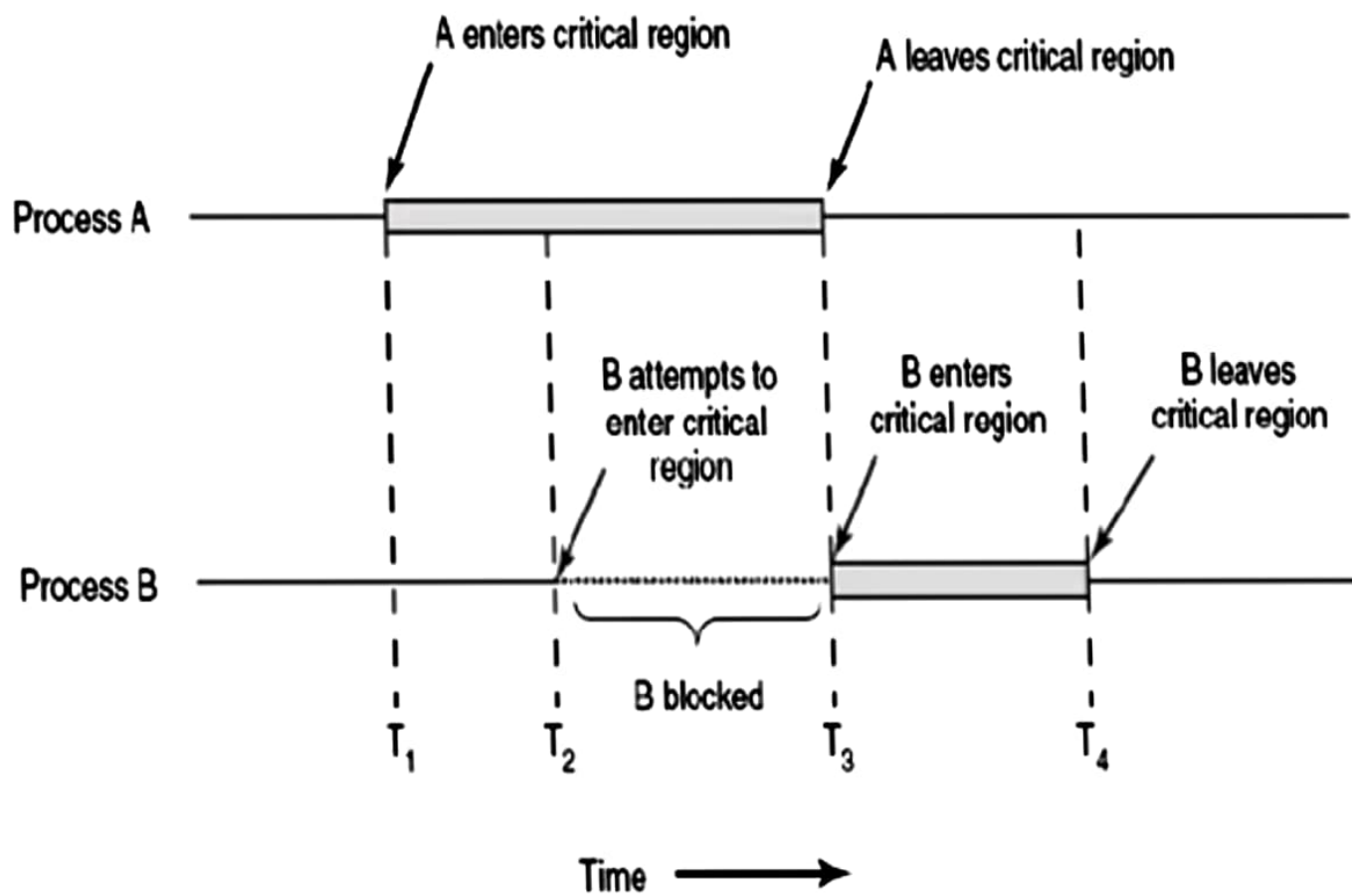
Mutually Exclusive Access

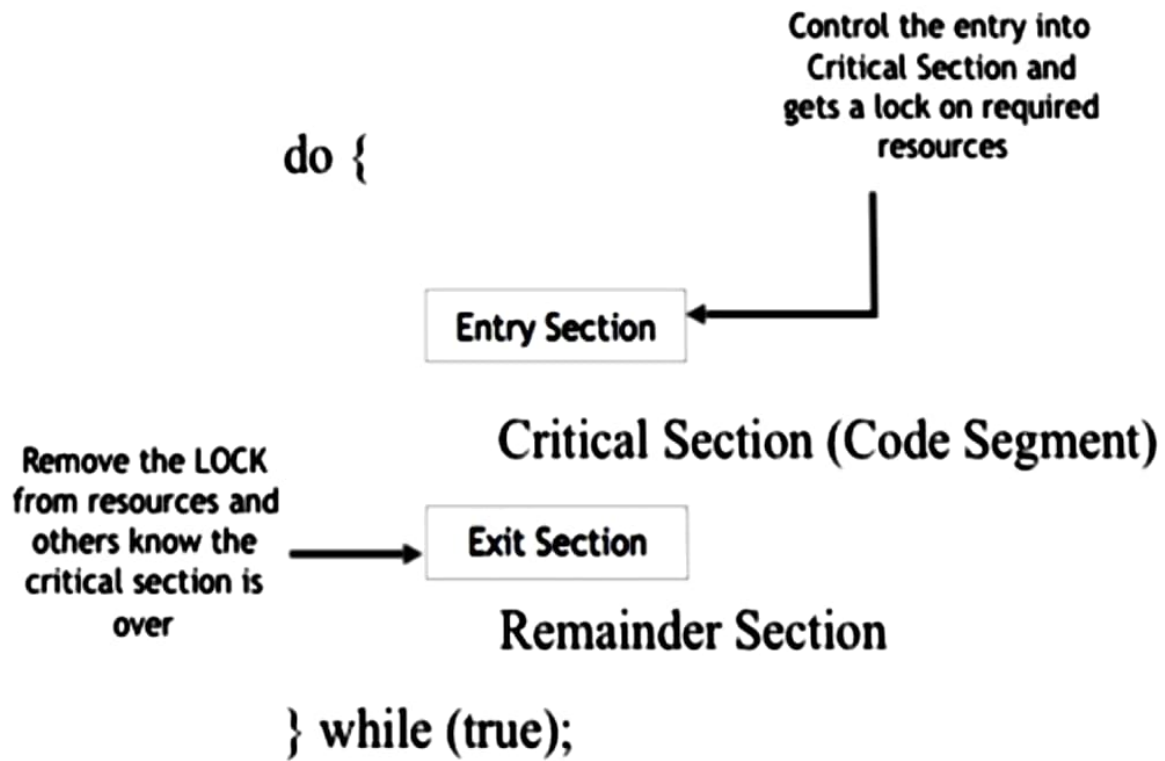


- Two processes share a common, fixed-sized buffer.
- Producer and consumer will access the same shared memory.
- Producer and consumer have to be synchronized, otherwise problems will arise.
- Two problems
 1. Producer is fast – No limit on buffer size
 2. Consumer is fast - Limit on buffer size

Critical section (critical region)

- The part of program where the shared memory is accessed.
- Only one process have to access shared memory at a time.
- Lock system is implemented.
- How one process can pass information to another.
- How to make sure two or more processes do not get into the critical region at the same time.





Mutual Exclusion

- Mechanisms that ensure that only one person or process is doing certain things at one time (others are excluded).
- Good solution requirements of **mutual exclusion**
- No two processes may be simultaneously inside their critical section Independent of CPU speed.
- No process running outside its critical section should block any process Bounded waiting

Hardware Solution

- TSL instruction – help from the hardware
- Instruction test and set lock TSL RX, LOCK
- 1. Read the content at the memory address of lock into register RX.
- 2. Store a non-zero value at the memory address of lock
- The operations of reading the content of lock and storing into it are guaranteed to be indivisible.

- How to use Test and Set Lock instruction for solving race condition?
- When lock = 0, No process will enter.
- Any process may set lock = 1 using TSL instruction and go to its critical section.
- When the process finishes its critical section, set lock = 0 using the original move instruction.

Race Condition

- A situation where two or more processes are reading or writing some shared data and the final result depends on who runs precisely when, is called race condition.
- A section of code, or collection of operations, in which only one thread may be executing at a given time because the code needs to access a shared resource. (e.g. code segment for printing to a printer)

- X=10

Fun()	p1	p2
{	10	11
Read x;	11	11 – 12(correct)
X=x+1;		
Write x;		
}		

- A solution for the race condition should have following conditions:
- 1. No two processes may be simultaneously inside their critical section (mutual exclusion).
- 2. No process running outside its critical region may block other processes.
- 3. No process should have to wait forever to enter critical region.
- 4. No assumptions may be made about speeds or the number of CPUs.

Strict Alternation

- Turn Variable or Strict Alternation Approach is the software mechanism implemented at user mode.
- It is a busy waiting solution which can be implemented only for two processes. In this approach, A turn variable is used which is actually a lock.
- This approach can only be used for only two processes. In general, let the two processes be P0 and P1.
- They share a variable called turn variable. The pseudo code of the program can be given as following.

Process P0

while (turn != 0);

Entry Section

Critical Section

turn = 1

Exit Section

Process P1

while (turn != 1);

Entry Section

Critical Section

turn = 0

Exit Section

Peterson Solution

- Peterson's solution is widely used solution to critical section problems. This algorithm was developed by a computer scientist Peterson that's why it is named as a Peterson's solution.
- In this solution, when a process is executing in a critical state, then the other process only executes the rest of the code, and the opposite can happen. This method also helps to make sure that only a single process runs in the critical section at a specific time.

FLAG

P1	False
P2	True
P3	True
.	
.	
Pn	False

- Assume there are N processes (P_1, P_2, \dots, P_N) and every process at some point of time requires to enter the Critical Section
- A `FLAG[]` array of size N is maintained which is by default false. So, whenever a process requires to enter the critical section, it has to set its flag as true. For example, If P_i wants to enter it will set `FLAG[i]=TRUE`.
- Another variable called `TURN` indicates the process number which is currently waiting to enter into the CS.

Semaphores

- Semaphores are integer variables that are used to solve the critical section problem by using two atomic operations, wait and signal are used for process synchronization.
- Conditions
 1. Mutual Exclusion
 2. Bounded Wait
 3. Progress
 4. Interference

1. 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 ($S \leq 0$);`

`S=S-1;`

`}`

- Signal
- The signal operation increments the value of its argument S.

signal(S)

{

S=S+1;

}

```
do {  
    waiting(mutex);  
  
    // critical section  
  
    signal(mutex);  
  
    // remainder section  
}while (TRUE);
```

- **Types of Semaphores**

- **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.

- **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.

Advantages of Semaphores

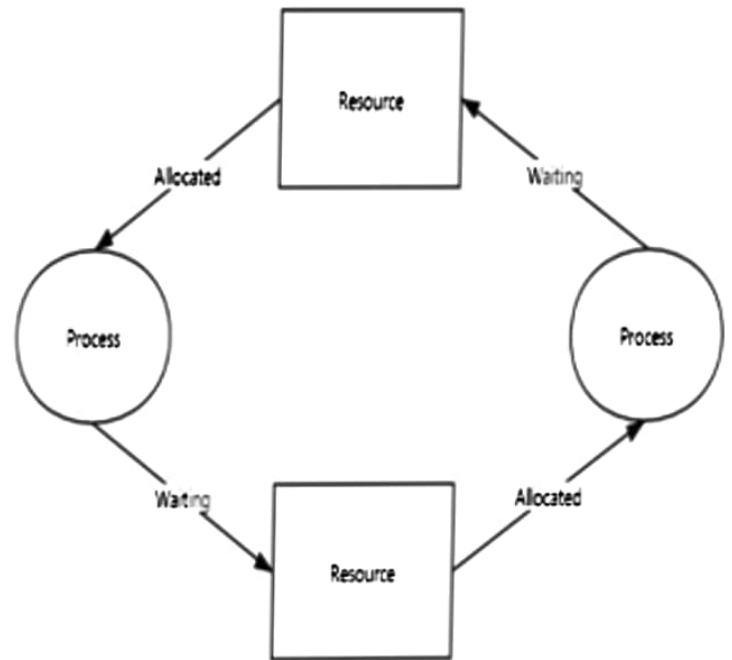
- **Semaphores allow only one process into the critical section. They follow the mutual exclusion principle strictly and are much more efficient than some other methods of synchronization.**
- **There is no resource wastage because of busy waiting in semaphores as processor time is not wasted unnecessarily 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

- 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.

Deadlock

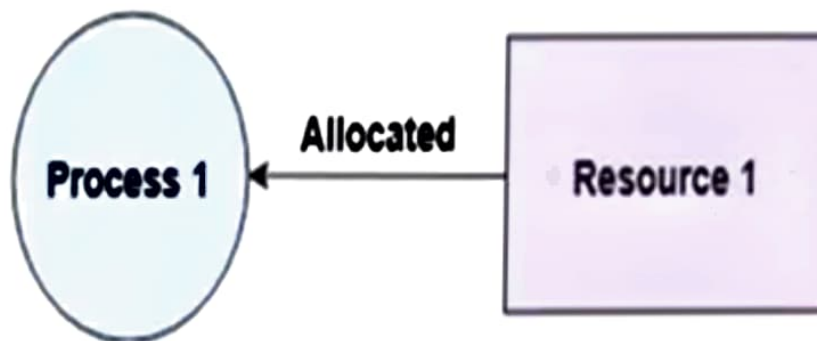
A deadlock happens in operating system when two or more processes need some resource to complete their execution that is held by the other process.



Four Conditions

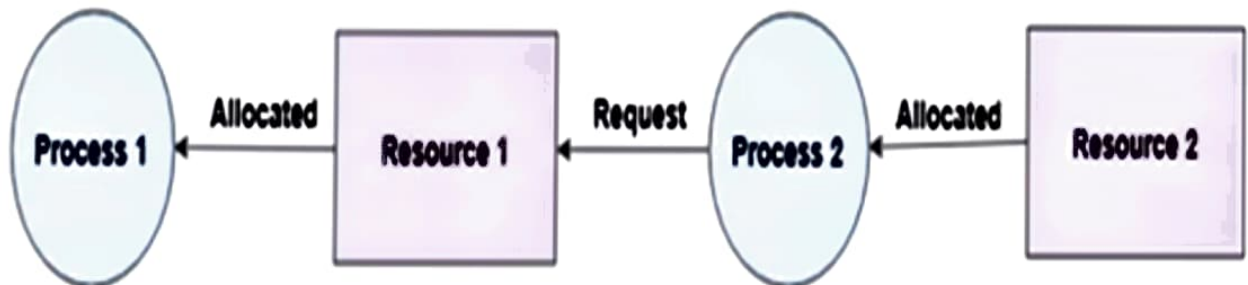
1. Mutual Exclusion

There should be a resource that can only be held by one process at a time. In the diagram below, there is a single instance of Resource 1 and it is held by Process 1 only.



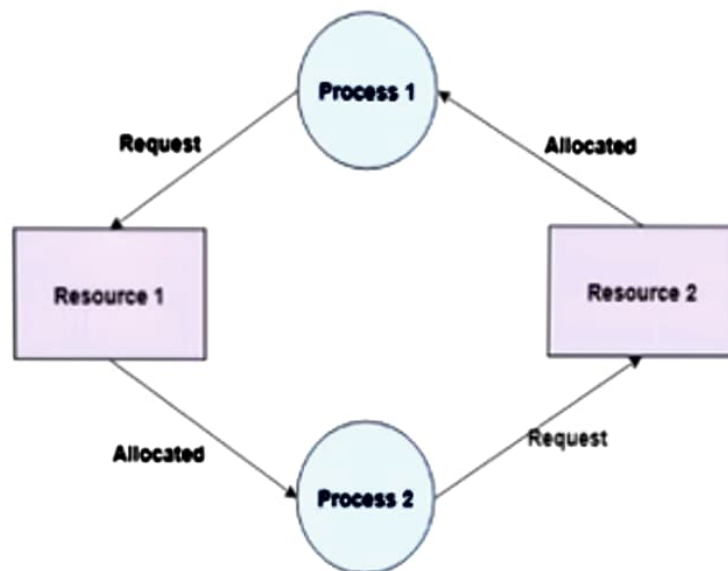
2. No Preemption

A resource cannot be preempted from a process by force. A process can only release a resource voluntarily. In the diagram below, Process 2 cannot preempt Resource 1 from Process 1. It will only be released when Process 1 relinquishes it voluntarily after its execution is complete



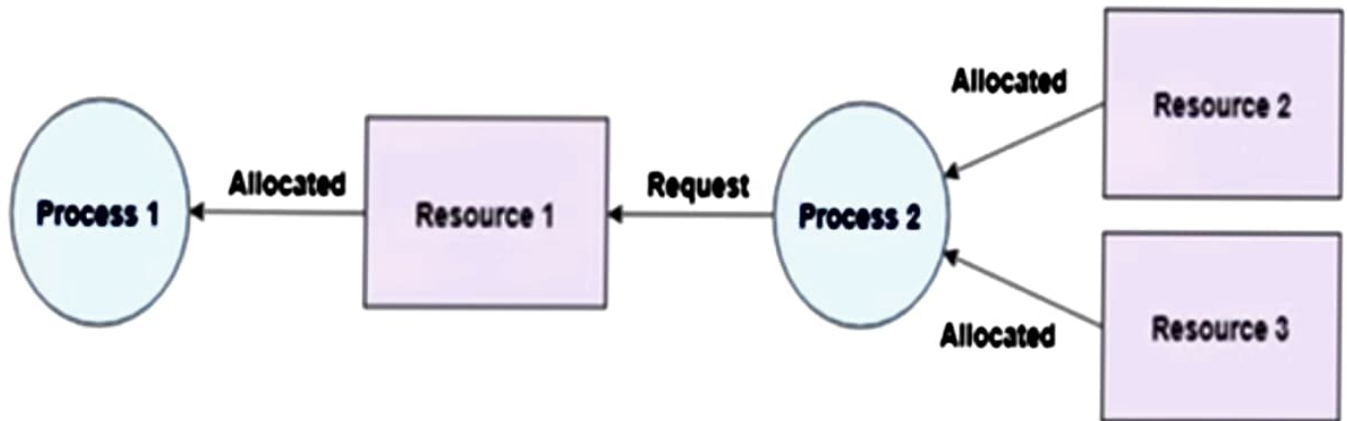
3. Circular Wait

A process is waiting for the resource held by the second process, which is waiting for the resource held by the third process and so on, till the last process is waiting for a resource held by the first process. This forms a circular chain. For example: Process 1 is allocated Resource 2 and it is requesting Resource 1. Similarly, Process 2 is allocated Resource 1 and it is requesting Resource 2. This forms a circular wait loop.



4. Hold and Wait

A process can hold multiple resources and still request more resources from other processes which are holding them. In the diagram given below, Process 2 holds Resource 2 and Resource 3 and is requesting the Resource 1 which is held by Process 1.






Methods Handle to Deadlock

1. Deadlock Prevention

Deadlock prevention algorithms ensure that at least one of the necessary conditions (Mutual exclusion, hold and wait, no preemption and circular wait) does not hold true. However most prevention algorithms have poor resource utilization, and hence result in reduced throughputs.

- Mutual Exclusion -- No Shared Variable
- No Preemption -- Preemption
- Hold and wait -- Only 1 Resource
- Circular Wait

- 1. Mouse  P1
- 2. Printer 
- 3. Keyboard 

- At least one condition have to be false

Deadlock Detection

- If deadlock prevention and avoidance are not done properly, as deadlock may occur and only things left to do is to detect the recover from the deadlock.
- If all resource types has only single instance, then we can use a graph called wait-for-graph, which is a variant of resource allocation graph.
- The wait-for-graph is not much useful if there are multiple instances for a resource, as a cycle may not imply a deadlock. In such a case, we can use an algorithm similar to Banker's algorithm to detect deadlock.

Deadlock Recovery

- Once a deadlock is detected, you will have to break the deadlock. It can be done through different ways, including, aborting one or more processes to break the circular wait condition causing the deadlock and preempting resources from one or more processes which are deadlocked.

Deadlock Avoidance

- Avoidance is kind of futuristic in nature. By using strategy of “Avoidance”, we have to make an assumption.
- We need to ensure that all information about resources which process will need are known to us prior to execution of the process.
- We use Banker’s algorithm (Which is in-turn a gift from Dijkstra) in order to avoid deadlock.