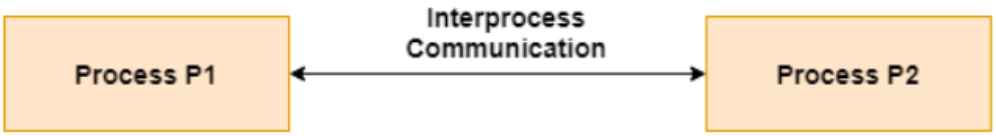
Unit No. 4

Interprocess communication & Synchronisation

Interprocess communication is the mechanism provided by the operating system that allows processes to communicate with each other. This communication could involve a process letting another process know that some event has occurred or the transferring of data from one process to another.

A diagram that illustrates interprocess communication is as follows –



Why we need interprocess communication?

There are numerous reasons to use inter-process communication for sharing the data. Here are some of the most important reasons that are given below:

- It helps to speedup modularity
- Computational
- Privilege separation
- Convenience
- Helps operating system to communicate with each other and synchronize their actions as well.

Thread is an execution unit that consists of its own program counter, a stack, and a set of registers where the program counter mainly keeps track of which instruction to execute next, a set of registers mainly hold its current working variables, and a stack mainly contains the history of execution

- Threads are also known as Lightweight processes

Synchronization in Interprocess Communication

Synchronization is a necessary part of interprocess communication. It is either provided by the interprocess control mechanism or handled by the communicating processes. Some of the methods to provide synchronization are as follows −

- **Semaphore**
  A semaphore is a variable that controls the access to a common resource by multiple processes. The two types of semaphores are binary semaphores and counting semaphores.
- **Mutual Exclusion**
  Mutual exclusion requires that only one process thread can enter the critical section at a time. This is useful for synchronization and also prevents race conditions.
- **Barrier**
  A barrier does not allow individual processes to proceed until all the processes reach it. Many parallel languages and collective routines impose barriers.
- **Spinlock**
  This is a type of lock. The processes trying to acquire this lock wait in a loop while checking if the lock is available or not. This is known as busy waiting because the process is not doing any useful operation even though it is active.

Busy waiting, also known as spinning, or busy looping is a process synchronization technique **in which a process/task waits and constantly checks for a condition to be satisfied before proceeding with its execution**. In busy waiting, a process executes instructions that test for the entry condition to be true, such as the availability of a lock or resource in the computer system.

For resource availability, consider a scenario where a process needs a resource for a specific program. However, the resource is currently in use and unavailable at the moment, therefore the process has to wait for resource availability before it can continue. This is what is known as busy waiting

There are two general approaches to waiting in operating systems: **firstly, a process/task can continuously check for the condition to be satisfied while consuming the processor – busy waiting.**

Secondly, a process can wait without consuming the processor. In such a case, the process/task is alerted or awakened when the condition is satisfied. The latter is known as sleeping, blocked waiting, or sleep waiting.

What is Semaphore?

Semaphore is simply a variable that is non-negative and shared between threads. A semaphore is a signaling mechanism, and a thread that is waiting on a semaphore can be signaled by another thread. It uses two atomic operations, 1) Wait, and 2) Signal for the process synchronization.

A semaphore either allows or disallows access to the resource, which depends on how it is set up.

Characteristic of Semaphore

Here, are characteristic of a semaphore:

- It is a mechanism that can be used to provide synchronization of tasks.
- It is a low-level synchronization mechanism.
- Semaphore will always hold a non-negative integer value.
- Semaphore can be implemented using test operations and interrupts, which should be executed using file descriptors.

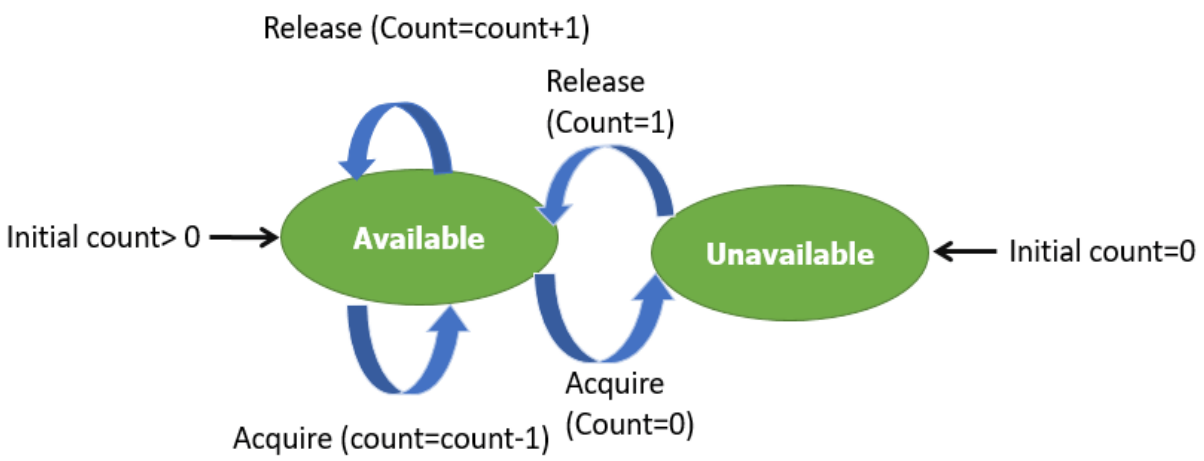
Types of Semaphores

The two common kinds of semaphores are

- Counting semaphores
- Binary semaphores.

Counting Semaphores

This type of Semaphore uses a count that helps task to be acquired or released numerous times. If the initial count = 0, the counting semaphore should be created in the unavailable state.

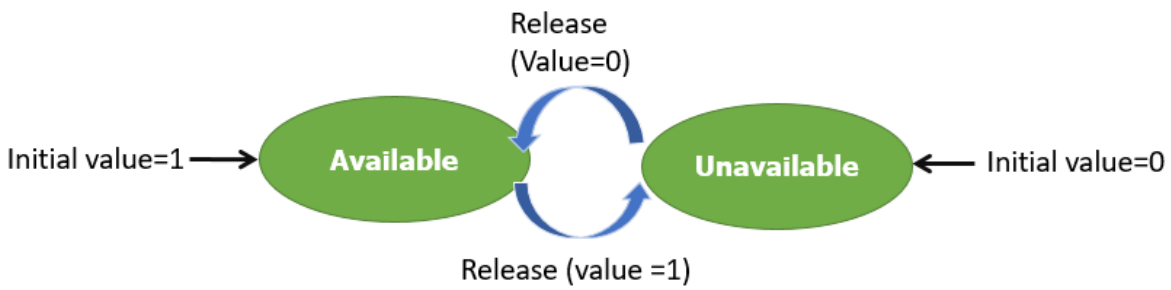


Counting Semaphores

However, If the count is > 0, the semaphore is created in the available state, and the number of tokens it has equals to its count.

Binary Semaphores

The binary semaphores are quite similar to counting semaphores, but their value is restricted to 0 and 1. In this type of semaphore, the wait operation works only if semaphore = 1, and the signal operation succeeds when semaphore= 0. It is easy to implement than counting semaphores.



Implementation of Semaphore

Any processes that want to access the shared resources have to first execute the **entry section** where the **wait()** operation decrements the value of the semaphore. If the value of the semaphore is 0 that means all the resources are in use and now onwards the processes trying to execute the entry section would be blocked.

Entry Section Code

```
Wait(Semaphore S)
{
S.vlaue = S.value -1;
If (S.value≤0)
{
block the process Sleep();
}
else
return;
}
```

Resources

Exit Section code

```
Signal(Semaphore S)
{
S.value= S.value + 1;
If(S.value≤0)
{
Unblock the process wake
up();
}
}
```

Implementation of Semaphore

Now when a process wants to release the resource then it has to execute the **exit code** before releasing the resource. Executing the exit code the process increments the value of semaphore with the **signal()** operation, unblocks a blocked process and thereby release the resource.

The unblocked process can now try to execute the entry section code to access the resource. There is always a waiting queue or blocked queue linked with each semaphore. The processes are unblocked or removed

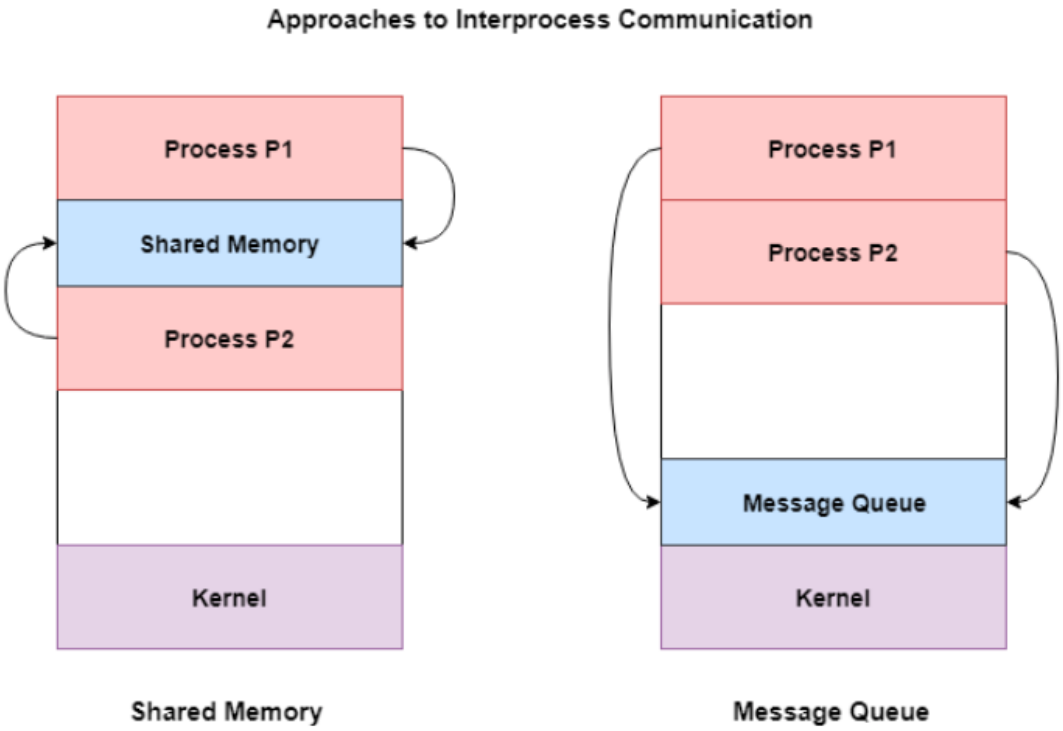
from the waiting queue in the **First-in-First-out** manner. That means the processes that have been waiting for a long is released first.

Approaches to Interprocess Communication

The different approaches to implement interprocess communication are given as follows −

- **Pipe**
  A pipe is a data channel that is unidirectional. Two pipes can be used to create a two-way data channel between two processes. This uses standard input and output methods. Pipes are used in all POSIX systems as well as Windows operating systems.
- **Socket**
  The socket is the endpoint for sending or receiving data in a network. This is true for data sent between processes on the same computer or data sent between different computers on the same network. Most of the operating systems use sockets for interprocess communication.
- **File**
  A file is a data record that may be stored on a disk or acquired on demand by a file server. Multiple processes can access a file as required. All operating systems use files for data storage.
- **Signal**
  Signals are useful in interprocess communication in a limited way. They are system messages that are sent from one process to another. Normally, signals are not used to transfer data but are used for remote commands between processes.
- **Shared Memory**
  Shared memory is the memory that can be simultaneously accessed by multiple processes. This is done so that the processes can communicate with each other. All POSIX systems, as well as Windows operating systems use shared memory.
- **Message Queue**
  Multiple processes can read and write data to the message queue without being connected to each other. Messages are stored in the queue until their recipient retrieves them. Message queues are quite useful for interprocess communication and are used by most operating systems.

A diagram that demonstrates message queue and shared memory methods of interprocess communication is as follows −



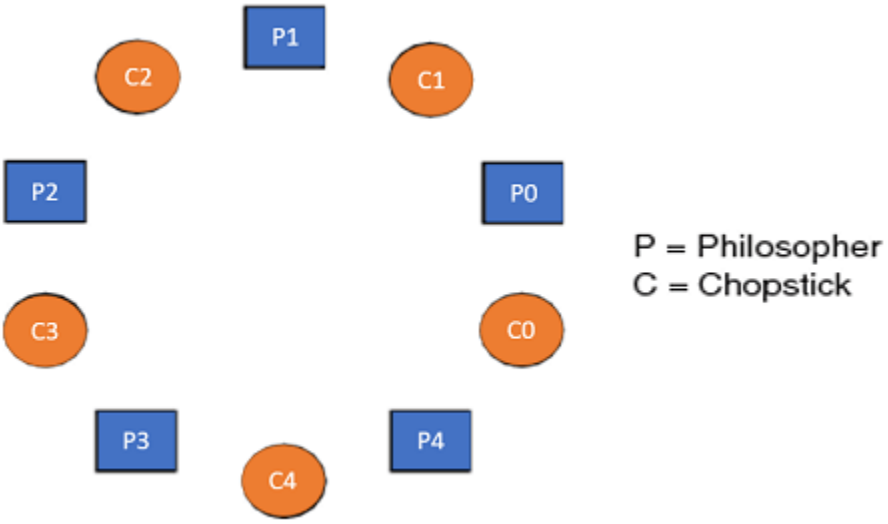
THE DINING PHILOSOPHERS PROBLEM

The dining philosopher's problem is the classical problem of synchronization which says that Five philosophers are sitting around a circular table and their job is to think and eat alternatively. A bowl of noodles is placed at the center of the table along with five chopsticks for each of the philosophers. To eat a philosopher needs both their right and a left chopstick. A philosopher can only eat if both immediate left and right chopsticks of the philosopher is available. In case if both immediate left and right chopsticks of the philosopher are not available then the philosopher puts down their (either left or right) chopstick and starts thinking again.

The dining philosopher demonstrates a large class of concurrency control problems hence it's a classic synchronization problem.

Five Philosophers sitting around the table

Dining Philosophers Problem- Let's understand the Dining Philosophers Problem with the below code, we have used fig 1 as a reference to make you understand the problem exactly. The five Philosophers are represented as P0, P1, P2, P3, and P4 and five chopsticks by C0, C1, C2, C3, and C4.




```
Void Philosopher
 {
 while(1)
  {
   take_chopstick[i];
   take_chopstick[ (i+1) % 5] ;
   . .
```

```
    . EATING THE NOODLE
    .
    put_chopstick[i] );
    put_chopstick[ (i+1) % 5] ;
    .
    . THINKING
  }
}
```

Let's discuss the above code:

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.

The solution of the Dining Philosophers Problem

We use a semaphore to represent a chopstick and this truly acts as a solution of the Dining Philosophers Problem. Wait and Signal operations will be used for the solution of the Dining Philosophers Problem, for picking a chopstick wait operation can be executed while for releasing a chopstick signal semaphore can be executed.

Semaphore: A semaphore is an integer variable in S, that apart from initialization is accessed by only two standard atomic operations - wait and signal, whose definitions are as follows:

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

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

From the above definitions of wait, it is clear that if the value of S <= 0 then it will enter into an infinite loop(because of the semicolon; after while loop). Whereas the job of the signal is to increment the value of S.

The structure of the chopstick is an array of a semaphore which is represented as shown below -

```
semaphore C[5];
```

Initially, each element of the semaphore C0, C1, C2, C3, and C4 are initialized to 1 as the chopsticks are on the table and not picked up by any of the philosophers.

Let's modify the above code of the Dining Philosopher Problem by using semaphore operations wait and signal, the desired code looks like

```
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
  }
 }
```

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)

The drawback of the above solution of the dining philosopher problem

From the above solution of the dining philosopher problem, we have proved that no two neighbouring philosophers can eat at the same point in time. The drawback of the above solution is that this solution can lead to a deadlock condition. This situation happens if all the philosophers pick their left chopstick at the same time, which leads to the condition of deadlock and none of the philosophers can eat.

To avoid deadlock, some of the solutions are as follows -

- Maximum number of philosophers on the table should not be more than four, in this case, chopstick C4 will be available for philosopher P3, so P3 will start eating and after the finish of his eating procedure, he will put down his both the chopstick C3 and C4, i.e. semaphore C3 and C4 will now be incremented to 1. Now philosopher P2 which was holding chopstick C2 will also have chopstick C3 available, hence similarly, he will put down his chopstick after eating and enable other philosophers to eat.
- A philosopher at an even position should pick the right chopstick and then the left chopstick while a philosopher at an odd position should pick the left chopstick and then the right chopstick.
- Only in case if both the chopsticks (left and right) are available at the same time, only then a philosopher should be allowed to pick their chopsticks
- All the four starting philosophers (P0, P1, P2, and P3) should pick the left chopstick and then the right chopstick, whereas the last philosopher P4 should pick the right chopstick and then the left chopstick. This will force P4

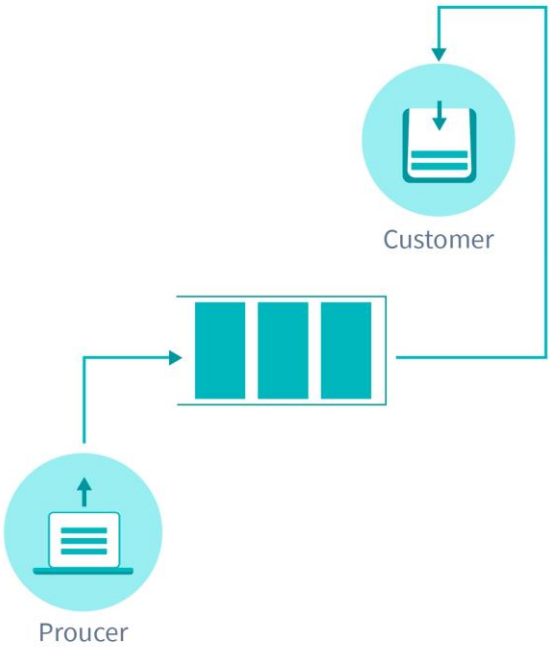
to hold his right chopstick first since the right chopstick of P4 is C0, which is already held by philosopher P0 and its value is set to 0, i.e C0 is already 0, because of which P4 will get trapped into an infinite loop and chopstick C4 remains vacant. Hence philosopher P3 has both left C3 and right C4 chopstick available, therefore it will start eating and will put down its both chopsticks once finishes and let others eat which removes the problem of deadlock.

The design of the problem was to illustrate the challenges of avoiding deadlock, a deadlock state of a system is a state in which no progress of system is possible. Consider a proposal where each philosopher is instructed to behave as follows:

- The philosopher is instructed to think till the left fork is available, when it is available, hold it.
- The philosopher is instructed to think till the right fork is available, when it is available, hold it.
- The philosopher is instructed to eat when both forks are available.
- then, put the right fork down first
- then, put the left fork down next
- repeat from the beginning.

Producer Consumer Problem

- In operating System **Producer is a process which is able to produce data/item**.
- Consumer is a **Process that is able to consume the data/item produced by the Producer**.
- Both Producer and Consumer share a common memory buffer. This buffer is a space of a certain size in the memory of the system which is used for storage. The producer produces the data into the buffer and the consumer consumes the data from the buffer.



So, what are the Producer-Consumer Problems?

1. Producer Process should not produce any data when the shared buffer is full.
2. Consumer Process should not consume any data when the shared buffer is empty.
3. The access to the shared buffer should be mutually exclusive i.e at a time only one process should be able to access the shared buffer and make changes to it.

For consistent data synchronization between Producer and Consumer, the above problem should be resolved.

Solution For Producer Consumer Problem

To solve the Producer-Consumer problem three **semaphores variable** are used :

Semaphores are variables used to indicate the number of resources available in the system at a particular time. semaphore variables are used to achieve `Process Synchronization.

Full

The full variable is used to track the space filled in the buffer by the Producer process. It is initialized to 0 initially as initially no space is filled by the Producer process.

Empty

The Empty variable is used to track the empty space in the buffer. The Empty variable is initially initialized to the **BUFFER-SIZE** as initially, the whole buffer is empty.

Mutex

Mutex is used to achieve mutual exclusion. mutex ensures that at any particular time only the producer or the consumer is accessing the buffer.

Mutex - mutex is a binary semaphore variable that has a value of 0 or 1.

We will use the Signal() and wait() operation in the above-mentioned semaphores to arrive at a solution to the Producer-Consumer problem.

Signal() - The signal function increases the semaphore value by 1. **Wait() -** The wait operation decreases the semaphore value by 1.

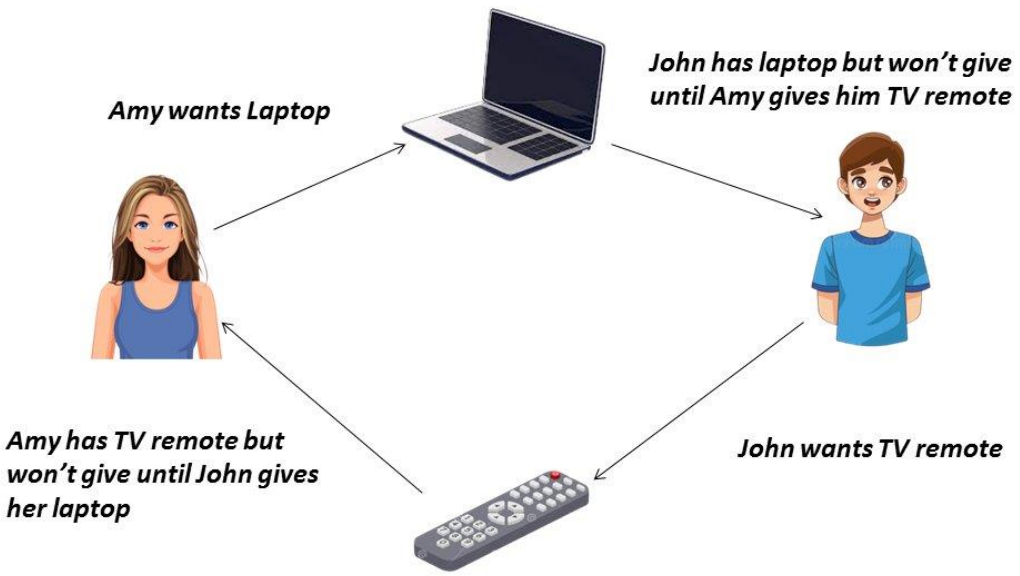
- Producer Process produces data item and consumer process consumes data item.
- Both producer and consumer processes share a common memory buffer.
- Producer should not produce any item if the buffer is full.
- Consumer should not consume any item if the buffer is empty.
- Not more than one process should access the buffer at a time i.e mutual exclusion should be there.
- Full, Empty and mutex semaphore help to solve Producer-consumer problem.
- Full semaphore checks for the number of filled space in the buffer by the producer process
- Empty semaphore checks for the number of empty spaces in the buffer.
- mutex checks for the mutual exclusion.

Deadlock in Operating System-

Deadlock is the state when two or more processes fight for same resource

What is Deadlock?

Deadlock is when two or more processes wait for each other to release a resource. This creates a standstill, and the system becomes unresponsive until one of the processes is killed.



Necessary Conditions for Deadlock

These four conditions must be met for a deadlock to happen in an operating system.

1. Mutual Exclusion

In this, two or more processes must compete for the same resources. There must be some resources that can only be used one process at a time. This means the resource is non-sharable. This could be a physical resource like a printer or an abstract concept like a lock on a shared **data structure**.

2. Hold and Wait

Hold and wait is when a process is holding a resource and waiting to acquire another resource that it needs but cannot proceed because another process is keeping the first resource. Each of these processes must have a hold on at least one of the resources it's requesting. If one process doesn't have a hold on any of the resources, it can't wait and will give up immediately.

3. No Preemption

Preemption means temporarily interrupting a task or process to execute another task or process. Preemption can occur due to an external event or internally within the system. If we take away the resource from the process that is causing deadlock, we can avoid deadlock. But is it a good approach? The answer is NO because that will lead to an inconsistent state. For example, if we take away memory from any

process(whose data was in the process of getting stored) and assign it to some other process. Then will lead to an inconsistent state.

4. Circular Wait

The circular wait is when two processes wait for each other to release a resource they are holding, creating a deadlock. There must be a cycle in the graph below. As you can see, process 1 is holding on to a resource R1 that process 2 in the cycle is waiting for. This is an example of a circular wait. To better understand let's understand with another example. For example, Process A might be holding on to Resource X while waiting for Resource Y, while Process B is holding on to Resource Y while waiting for Resource Z, and so on around the cycle.

Difference between Deadlock and Starvation

Deadlock	Starvation
In this, two or more processes are each waiting for the other to release a resource, and neither process is able to continue.	In this, a process is unable to obtain the resources it needs to continue running.
Different processes are unable to proceed because they are each waiting for the other to do something.	A process is unable to proceed due to the unavailability of that resource.
Deadlock is also called Circular wait.	Starvation is also called lived lock.
Avoiding the necessary conditions for deadlock can be prevented.	Starvation can be easily prevented by Aging.

Methods For Handling Deadlocks

1. Deadlock avoidance

Deadlock avoidance is the process of taking steps to prevent deadlock from occurring. Operating system uses the deadlock Avoidance method to ensure the system is in a safe state(when the system can allocate resources and can avoid being in a deadlock state). We have a Deadlock avoidance algorithm-**Banker's algorithm** for this. When a new process is to be executed, it requires some resources. So banker's algorithm needs to know

- How many resources the process could request
- Which processes hold many resources.
- How many resources the system has.

And accordingly, resources are being assigned if available resources are more than requested to avoid deadlock. Tell the operating system about the maximum number

of resources a process can request to complete its execution. The deadlock avoidance graph(shown in fig-2) assesses the resource-allocation state to check if a circular wait situation is not occurring.

If a deadlock does occur, it can sometimes be resolved by terminating one of the processes involved. However, this can cause data loss or corruption, so it's always preferable to try and prevent the deadlock from happening in the first place.

Bankers algorithm Pseudocode:

1. In ***s***tarting all the processes are to be executed. Define two data structure finish and work:

Finish[n]=False.

Work=Available

Where n is a number of processes to be executed.

2. Find the process for which Finish[i]=False

And ***Need <=Work***(This means a request is valid as the number of requested resources of each resource type is less than the **available resources**, In case no such process is there then go to step

3.***Work=Work+Allocation***

Finish[i]=True

Go to step 2 to find other processes

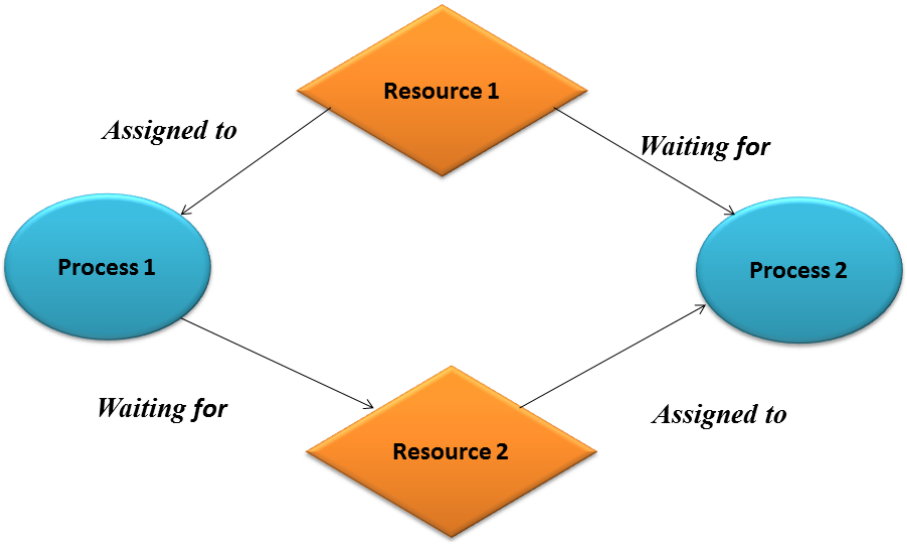
Any process says process 'i' finishes its execution. So that means the resources allocated to it previously, get free. So these resources are added to Work and Finish(i) of the process is set as true.

4. If ***Finish[i]=True*** for n processes then the system is in a safe state(If all the processes are executed in some sequence). Otherwise, it is in an unsafe state

2. Deadlock Detection

Detecting deadlocks is one of the most important steps in preventing them. A deadlock can happen anytime when two or more processes are trying to acquire a resource, and each process is waiting for other processes to release the resource.

The deadlock can be detected in the resource-allocation graph as shown in fig below.



This graph checks if there is a cycle in the **Resource Allocation Graph** and each resource in the cycle provides only one instance, If there is a cycle in this graph then the processes will be in a **deadlock state.**

So always remember detecting deadlocks is one of the most important steps in preventing them.

3. Deadlock Prevention

The best way to prevent deadlocks is by understanding how they form in the first place. Deadlock can be prevented by eliminating the necessary conditions for deadlock(explained above).

Some ways of prevention are as follows

- **Preempting resources:** Take the resources from the process and assign them to other processes.
- **Rollback:** When the process is taken away from the process, roll back and restart it.
- **Aborting:** Aborting the deadlocked processes.
- **Sharable resource:** If the resource is sharable, all processes will get all resources, and a deadlock situation won't come.

Conclusion

A deadlock is a situation that can occur in an operating system when two or more processes, each waiting for an event that only the other can cause, lock up the system so that no process can continue execution.