UNIT-III

Synchronization:

synchronization refers to one of two distinct but related concepts: synchronization of processes, and synchronization of data. *Process synchronization* refers to the idea that multiple processes are to join up or handshake at a certain point, in order to reach an agreement or commit to a certain sequence of action. *Data synchronization* refers to the idea of keeping multiple copies of a dataset in coherence with one another, or to maintain data integrity. Process synchronization primitives are commonly used to implement data synchronization.

Synchronization is a technique which is used to coordinate the process that use shared Data. There are two types of Processes in an Operating Systems.

1. **IndependentProcess** – The process that does not affect or is affected by the other process while its execution then the process is called Independent Process. Example The process that does not share any shared variable, database, files, etc.
2. **CooperatingProcess** – The process that affect or is affected by the other process while execution, is called a Cooperating Process. Example The process that share file, variable, database, etc are the Cooperating Process.

he need for synchronization does not arise merely in multi-processor systems but for any kind of concurrent processes; even in single processor systems. Mentioned below are some of the main needs for synchronization:

two or more works at a same time

Forks and Joins: When a job arrives at a fork point, it is split into N sub-jobs which are then serviced by n tasks. After being serviced, each sub-job waits until all other sub-jobs are done processing. Then, they are joined again and leave the system. Thus, parallel programming requires synchronization as all the parallel processes wait for several other processes to occur.

Producer-Consumer: In a producer-consumer relationship, the consumer process is dependent on the producer process till the necessary data has been produced.

Exclusive use resources: When multiple processes are dependent on a resource and they need to access it at the same time, the operating system needs to ensure that only one processor accesses it at a given point in time. This reduces concurrency.

The Critical Section Problem:

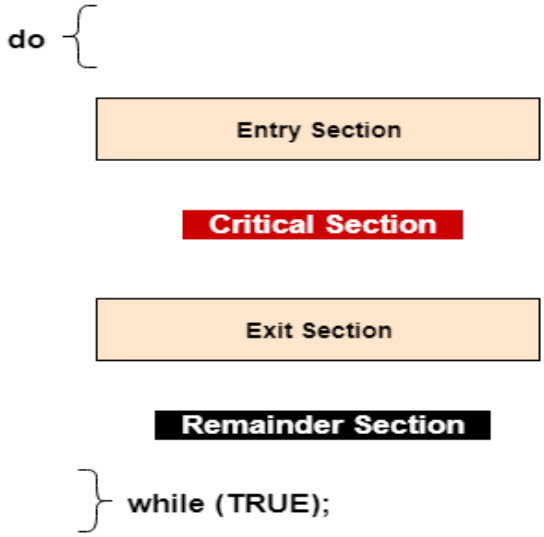
Critical Section is the part of a program which tries to access shared resources. That resource may be any resource in a computer like a memory location, Data structure, CPU or any IO device.

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.

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.

A diagram that demonstrates the critical section is as follows −



In the above diagram, the entry section handles the entry into the critical section. It acquires the resources needed for execution by the process. The exit section handles the exit from the critical section. It releases the resources and also informs the other processes that the critical section is free.

Solution to the Critical Section Problem

The critical section problem needs a solution to synchronize the different processes. The solution to the critical section problem must satisfy the following conditions −

- **Mutual Exclusion**

  Mutual exclusion implies that only one process can be inside the critical section at any time. If any other processes require the critical section, they must wait until it is free.

- **Progress**

  Progress means that if a process is not using the critical section, then it should not stop any other process from accessing it. In other words, any process can enter a critical section if it is free.

- **Bounded Waiting**

  Bounded waiting means that each process must have a limited waiting time. Itt should not wait endlessly to access the critical section.

Semaphores:

Usage&Implementation:

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

The definitions of wait and signal are as follows −

- **Wait**

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

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

  S--;
}
```

- **Signal**

  The signal operation increments the value of its argument S.

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

Types of Semaphores

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

- **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

Some of the advantages of semaphores are as follows −

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

Disadvantages of Semaphores

Some of the disadvantages of semaphores are as follows −

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

Deadlocks and starvation:

Starvation and Deadlock are situations that occur when the processes that require a resource are delayed for a long time. However they are quite different concepts.

Details about starvation and deadlock are given as follows –

Starvation

Starvation occurs if a process is indefinitely postponed. This may happen if the process requires a resource for execution that it is never alloted or if the process is never provided the processor for some reason.

Some of the common causes of starvation are as follows −

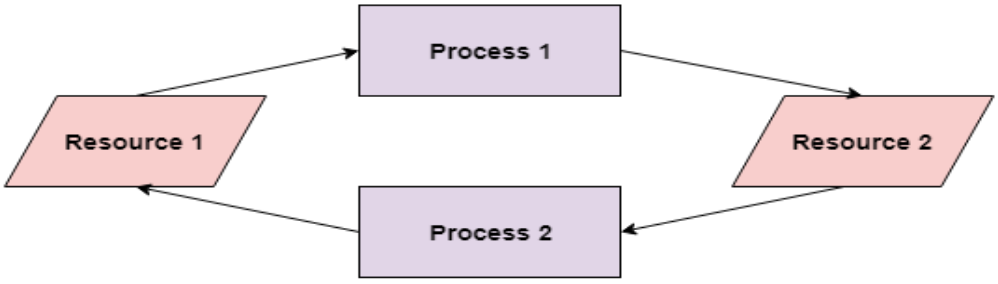
- If a process is never provided the resources it requires for execution because of faulty resource allocation decisions, then starvation can occur.
- A lower priority process may wait forever if higher priority processes constantly monopolize the processor.
- Starvation may occur if there are not enough resources to provide to every process as required.
- If random selection of processes is used then a process may wait for a long time because of non-selection.

Some solutions that can be implemented in a system to handle starvation are as follows −

- An independent manager can be used for allocation of resources. This resource manager distributes resources fairly and tries to avoid starvation.
- Random selection of processes for resource allocation or processor allocation should be avoided as they encourage starvation.
- The priority scheme of resource allocation should include concepts such as aging, where the priority of a process is increased the longer it waits. This avoids starvation.

Deadlock

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



Deadlock in Operating System

In the above diagram, the process 1 has resource 1 and needs to acquire resource 2. Similarly process 2 has resource 2 and needs to acquire resource 1. Process 1 and process 2 are in deadlock as each of them needs the other's resource to complete their execution but neither of them is willing to relinquish their resources.

A deadlock will only occur if the four Coffman conditions hold true. These conditions are not necessarily mutually exclusive. They are given as follows −

- **Mutual Exclusion**

  Mutual exclusion implies there should be a resource that can only be held by one process at a time. This means that the resources should be non-sharable.

- **Hold and Wait**

  A process can hold multiple resources and still request more resources from other processes which are holding them.

- **No preemption**

  A resource cannot be preempted from a process by force. A process can only release a resource voluntarily.

- **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.

Classic problems of synchronization:

we will see number of classical problems of synchronization as examples of a large class of concurrency-control problems. In our solutions to the problems, we use semaphores for synchronization, since that is the traditional way to present such solutions. However, actual implementations of these solutions could use mutex locks in place of binary semaphores.
These problems are used for testing nearly every newly proposed synchronization scheme. The following problems of synchronization are considered as classical problems:

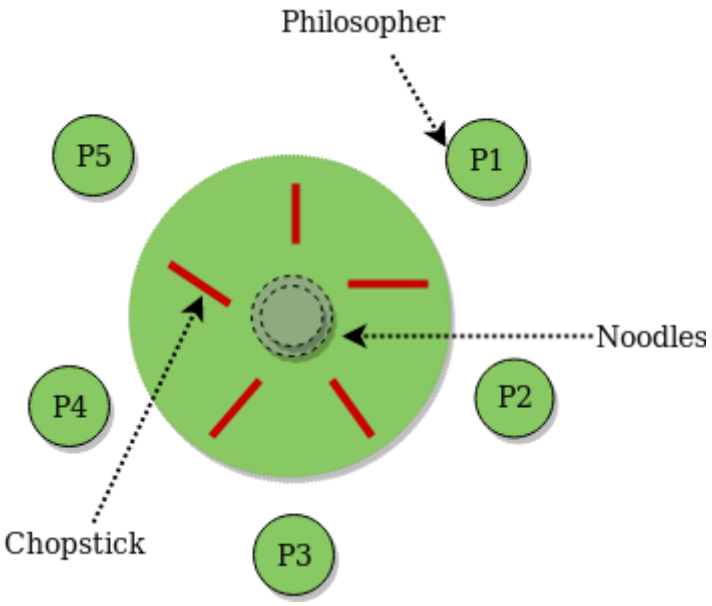
1. Bounded-buffer (or Producer-Consumer) Problem,
2. Dining-Philosophers Problem,
3. Readers and Writers Problem,
4. Sleeping Barber Problem
These are summarized, for detailed explanation, you can view the linked articles for each.

- **Bounded-buffer (or Producer-Consumer) Problem:**

  Bounded Buffer problem is also called producer consumer problem. This problem is generalized in terms of the Producer-Consumer problem. Solution to this problem is, creating two counting semaphores "full" and "empty" to keep track of the current number of full and empty buffers respectively. Producers produce a product and consumers consume the product, but both use of one of the containers each time.

- **Dining-Philosophers Problem:**

  The Dining Philosopher Problem states that K philosophers 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 pickup the two chopsticks adjacent to him. One chopstick may be picked up by any one of its adjacent followers but not both. This problem involves the allocation of limited resources to a group of processes in a deadlock-free and starvation-free manner.

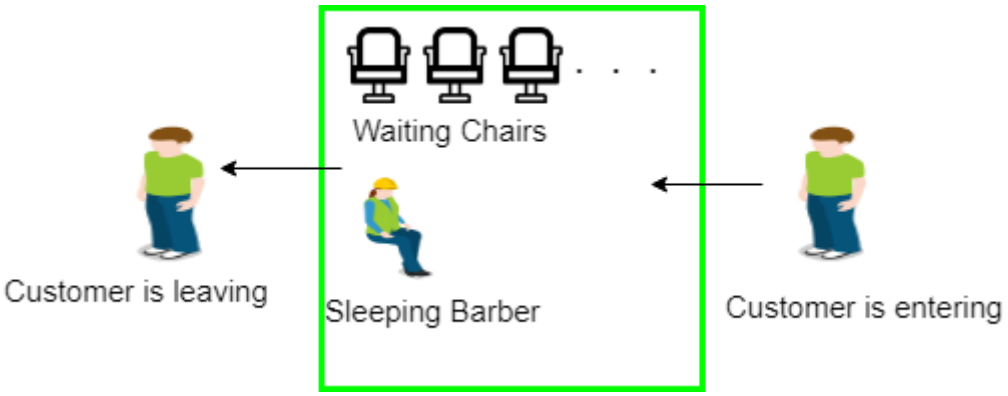


- **Readers and Writers Problem:**

  Suppose that a database is to be shared among several concurrent processes. Some of these processes may want only to read the database, whereas others may want to update (that is, to read and write) the database. We distinguish between these two types of processes by referring to the former as readers and to the latter as writers. Precisely in OS we call this situation as the readers-writers problem. Problem parameters:

  - 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.

- **SleepingBarberProblem:**
  Barber shop with one barber, one barber chair and N chairs to wait in. When no customers the barber goes to sleep in barber chair and must be woken when a customer comes in. When barber is cutting hair new customers take empty seats to wait, or leave if no vacancy.



Deadlocks

System model:

Overview :

A deadlock occurs when a set of processes is stalled because each process is holding a resource and waiting for another process to acquire another resource. In the diagram below, for example, Process 1 is holding Resource 1 while Process 2 acquires Resource 2, and Process 2 is waiting for Resource 1.

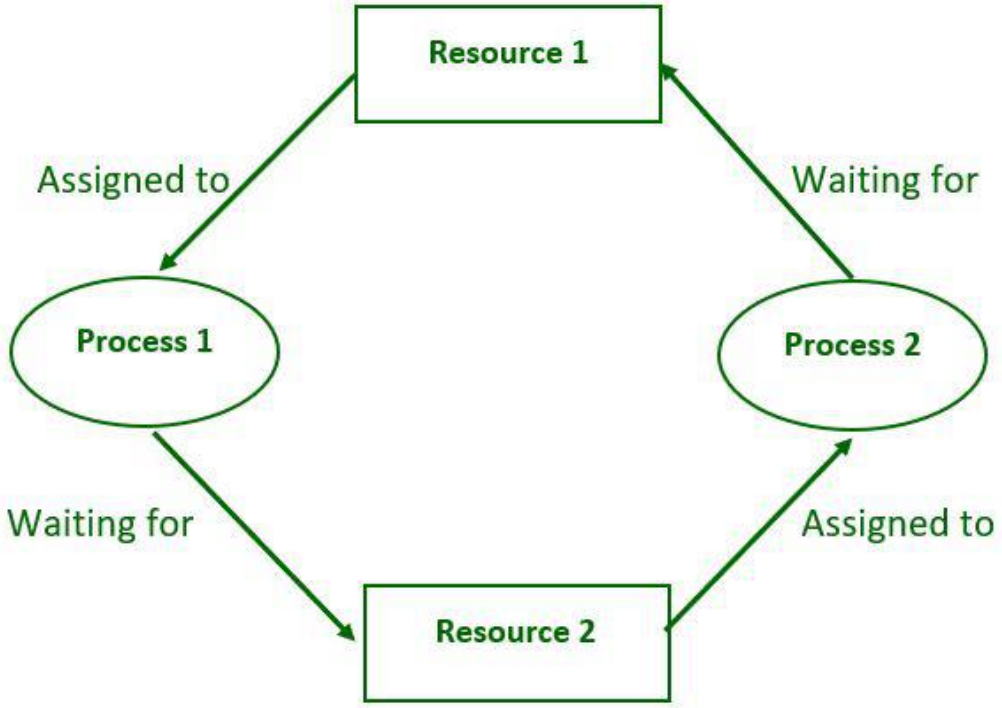


Figure: Deadlock in Operating system

System Model :

- For the purposes of deadlock discussion, a system can be modeled as a collection of limited resources that can be divided into different categories and allocated to a variety of processes, each with different requirements.
- Memory, printers, CPUs, open files, tape drives, CD-ROMs, and other resources are examples of resource categories.
- By definition, all resources within a category are equivalent, and any of the resources within that category can equally satisfy a request from that category. If this is not the case (i.e. if there is some difference between the resources within a category), then that category must be subdivided further. For example, the term

"printers" may need to be subdivided into "laser printers" and "color inkjet printers."

- Some categories may only have one resource.
- The kernel keeps track of which resources are free and which are allocated, to which process they are allocated, and a queue of processes waiting for this resource to become available for all kernel-managed resources. Mutexes or wait() and signal() calls can be used to control application-managed resources (i.e. binary or counting semaphores.)
- When every process in a set is waiting for a resource that is currently assigned to another process in the set, the set is said to be deadlocked.

Operations :

In normal operation, a process must request a resource before using it and release it when finished, as shown below.

1. **Request** –
   If the request cannot be granted immediately, the process must wait until the resource(s) required to become available. The system, for example, uses the functions open(), malloc(), new(), and request ().
2. **Use** –
   The process makes use of the resource, such as printing to a printer or reading from a file.
3. **Release** –
   The process relinquishes the resource, allowing it to be used by other processes.

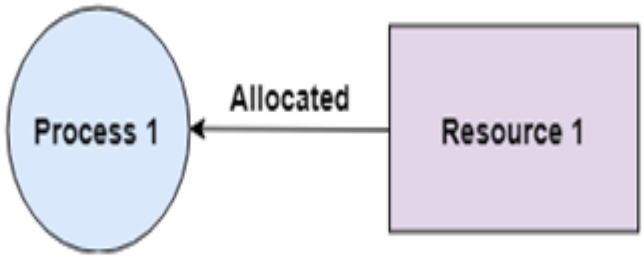
Deadlock characterization:

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.

A deadlock occurs if the four Coffman conditions hold true. But these conditions are not mutually exclusive. They are given as follows –

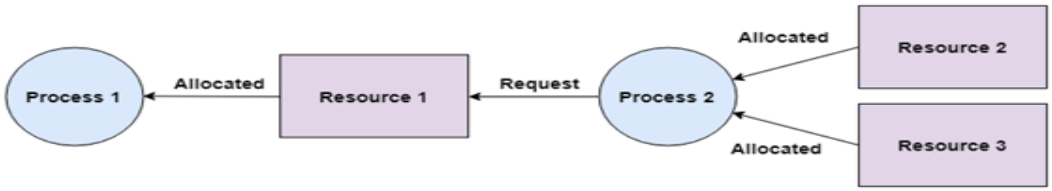
Mutual Exclusion

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.



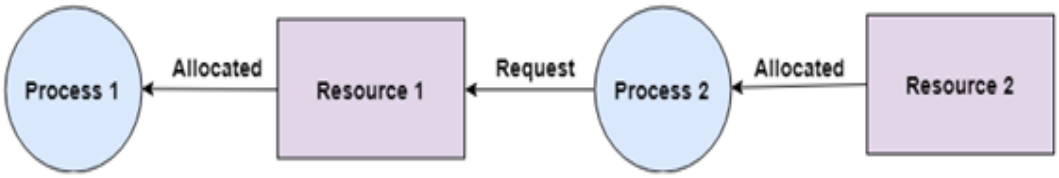
Hold and Wait

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.



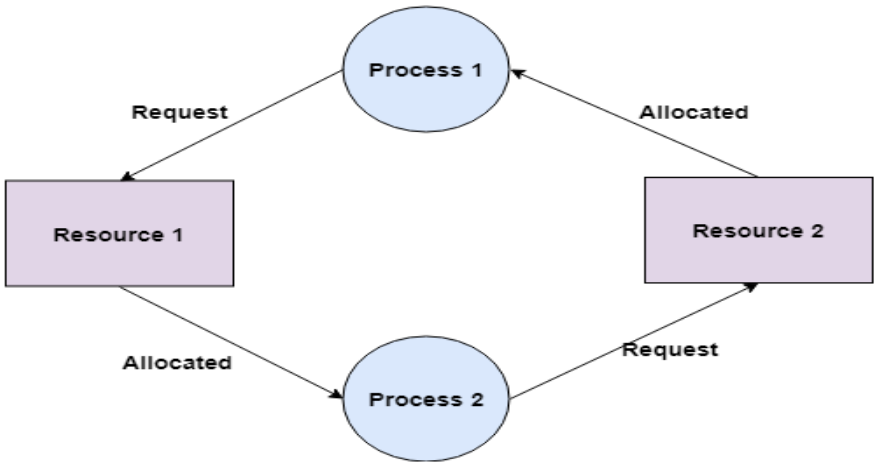
No Preemption

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.



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 Resource2 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.



Deadlock Prevention :

Deadlocks can be avoided by avoiding at least one of the four necessary conditions: as follows.

Condition-1 :

Mutual Exclusion :

- Read-only files, for example, do not cause deadlocks.
- Unfortunately, some resources, such as printers and tape drives, require a single process to have exclusive access to them.

Condition-2 :

Hold and Wait :

To avoid this condition, processes must be prevented from holding one or more resources while also waiting for one or more others. There are a few possibilities here:

- Make it a requirement that all processes request all resources at the same time. This can be a waste of system resources if a process requires one resource early in its execution but does not require another until much later.
- Processes that hold resources must release them prior to requesting new ones, and then re-acquire the released resources alongside the new ones in a single new request. This can be a problem if a process uses a resource to partially complete an operation and then fails to re-allocate it after it is released.
- If a process necessitates the use of one or more popular resources, either of the methods described above can result in starvation.

Condition-3 :

No Preemption :

When possible, preemption of process resource allocations can help to avoid deadlocks.

- One approach is that if a process is forced to wait when requesting a new resource, all other resources previously held by this process are implicitly released

(preempted), forcing this process to re-acquire the old resources alongside the new resources in a single request, as discussed previously.

- Another approach is that when a resource is requested, and it is not available, the system looks to see what other processes are currently using those resources and are themselves blocked while waiting for another resource. If such a process is discovered, some of their resources may be preempted and added to the list of resources that the process is looking for.
- Either of these approaches may be appropriate for resources whose states can be easily saved and restored, such as registers and memory, but they are generally inapplicable to other devices, such as printers and tape drives.

Condition-4 :

Circular Wait :

- To avoid circular waits, number all resources and insist that processes request resources is strictly increasing (or decreasing) order.
- To put it another way, before requesting resource Rj, a process must first release all Ri such that I >= j.
- The relative ordering of the various resources is a significant challenge in this scheme.