UNIT-III

Synchronization: Background, The critical section problem. Semaphores: Usage, Implementation, Deadlocks and Starvation, Classic problems of synchronization. Deadlocks: System Model, Deadlock Characterization, Deadlock Prevention.

Synchronization Background:

Introduction

When two or more process cooperates with each other, their order of execution must be preserved otherwise there can be conflicts in their execution and inappropriate outputs can be produced.

A cooperative process is the one which can affect the execution of other process or can be affected by the execution of other process. Such processes need to be synchronized so that their order of execution can be guaranteed.

The procedure involved in preserving the appropriate order of execution of cooperative processes is known as Process Synchronization. There are various synchronization mechanisms that are used to synchronize the processes.

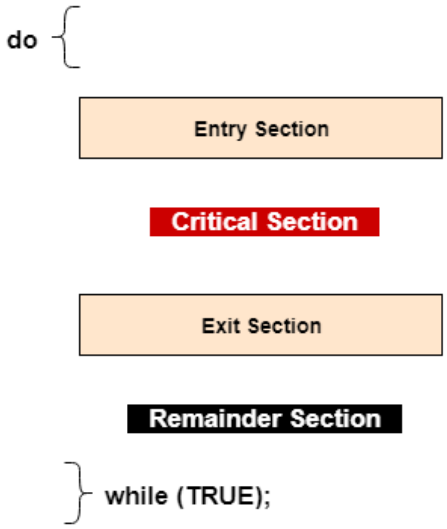
Race Condition

A Race Condition typically occurs when two or more threads try to read, write and possibly make the decisions based on the memory that they are accessing concurrently.

The 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.

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 :

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.

Classic Problems of Synchronization:

These problems are used for testing nearly every newly proposed synchronization scheme. The following problems of synchronization are considered as classical problems:

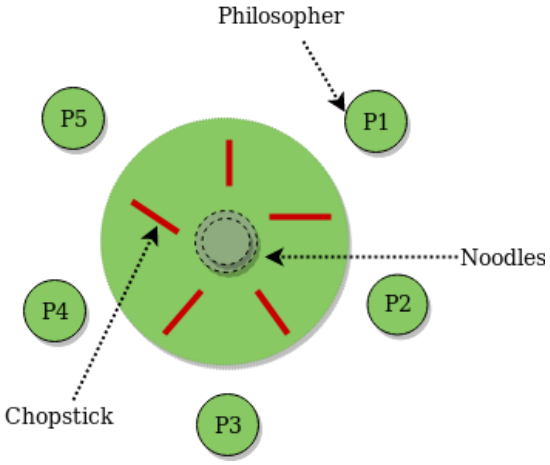
1. Bounded-buffer (or Producer-Consumer) Problem,
2. Dining-Philosophers Problem,
3. Readers and Writers Problem,
4. Sleeping Barber Problem
These are summarized, for detailed explanation, you can view the linked articles for each.

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:

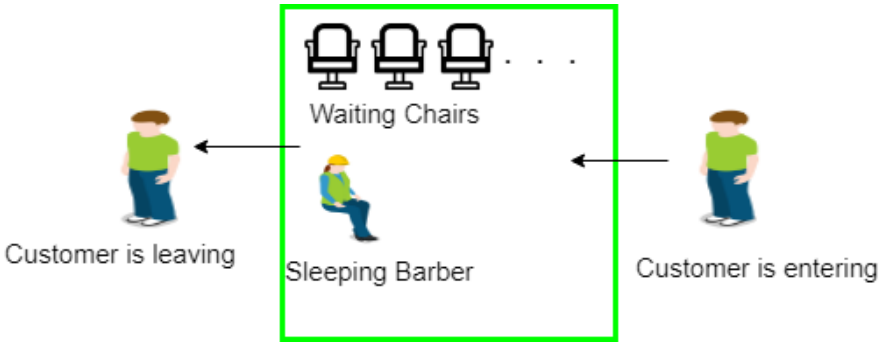
Suppose that a database is to be shared among several concurrent processes. Some of these processes may want only to read the database, whereas others may want to update (that is, to read and write) the database. We distinguish between these two types of processes by referring to the former as readers 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.

Sleeping Barber Problem:

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.



DEADLOCK :

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.

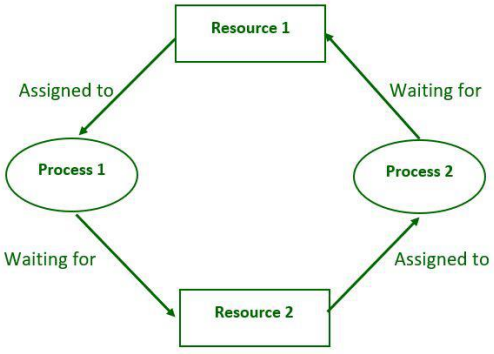


Figure: Deadlock in Operating system

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.

Necessary Conditions:

There are four conditions that must be met in order to achieve deadlock as follows.

1. **Mutual Exclusion** –

   At least one resource must be kept in a non-shareable state; if another process requests it, it must wait for it to be released.

2. **Hold and Wait** –
   A process must hold at least one resource while also waiting for at least one resource that another process is currently holding.

3. **No preemption** –
   Once a process holds a resource (i.e. after its request is granted), that resource cannot be taken away from that process until the process voluntarily releases it.

4. **Circular Wait** –
   There must be a set of processes P0, P1, P2,…, PN such that every P[I] is waiting for P[(I + 1) percent (N + 1)]. (It is important to note that this condition implies the hold-and-wait condition, but dealing with the four conditions is easier if they are considered separately).

Methods for Handling Deadlocks:

In general, there are three approaches to dealing with deadlocks as follows.

1. Preventing or avoiding deadlock by Avoid allowing the system to become stuck in a loop.
2. Detection and recovery of deadlocks, When deadlocks are detected, abort the process or preempt some resources.
3. Ignore the problem entirely.
4. To avoid deadlocks, the system requires more information about all processes. The system, in particular, must understand what resources a process will or may request in the future. (Depending on the algorithm, this can range from a simple worst-case maximum to a complete resource request and release plan for each process.)
5. Deadlock detection is relatively simple, but deadlock recovery necessitates either aborting processes or preempting resources, neither of which is an appealing option.
6. If deadlocks are not avoided or detected, the system will gradually slow down as more processes become stuck waiting for resources that the deadlock has blocked and other waiting processes. Unfortunately, when the computing requirements of a real-time process are high, this slowdown can be confused with a general system slowdown.

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.

Deadlock Avoidance:

- The general idea behind deadlock avoidance is to avoid deadlocks by avoiding at least one of the aforementioned conditions.
- This necessitates more information about each process AND results in low device utilization. (This is a conservative approach.)
- The scheduler only needs to know the maximum number of each resource that a process could potentially use in some algorithms. In more complex algorithms, the scheduler can also use the schedule to determine which resources are required and in what order.
- When a scheduler determines that starting a process or granting resource requests will result in future deadlocks, the process is simply not started or the request is denied.
- The number of available and allocated resources, as well as the maximum requirements of all processes in the system, define a resource allocation state.

Deadlock Detection:

- If deadlocks cannot be avoided, another approach is to detect them and recover in some way.
- Aside from the performance hit of constantly checking for deadlocks, a policy/algorithm for recovering from deadlocks must be in place, and when processes must be aborted or have their resources preempted, there is the possibility of lost work.

Recovery From Deadlock:

There are three basic approaches to getting out of a bind:

1. Inform the system operator and give him/her permission to intervene manually.
2. Stop one or more of the processes involved in the deadlock.
3. Prevent the use of resources.

Approach of Recovery from Deadlock:

Here, we will discuss the approach of Recovery from Deadlock as follows.
Approach-1:

Process Termination:

There are two basic approaches for recovering resources allocated to terminated processes as follows.

1. Stop all processes that are involved in the deadlock. This does break the deadlock, but at the expense of terminating more processes than are absolutely necessary.
2. Processes should be terminated one at a time until the deadlock is broken. This method is more conservative, but it necessitates performing deadlock detection after each step.

In the latter case, many factors can influence which processes are terminated next as follows.

1. Priorities in the process
2. How long has the process been running and how close it is to completion.
3. How many and what kind of resources does the process have? (Are they simple to anticipate and restore?)
4. How many more resources are required for the process to be completed?
5. How many processes will have to be killed?
6. Whether the process is batch or interactive.

Approach-2:

Resource Preemption:

When allocating resources to break the deadlock, three critical issues must be addressed:

1. **Selecting a victim –**

   Many of the decision criteria outlined above apply to determine which resources to preempt from which processes.
2. **Rollback –**

   A preempted process should ideally be rolled back to a safe state before the point at which that resource was originally assigned to the process. Unfortunately, determining such a safe state can be difficult or impossible, so the only safe rollback is to start from the beginning. (In other words, halt and restart the process.)
3. **Starvation –**

   How do you ensure that a process does not go hungry because its resources are constantly being preempted? One option is to use a priority system and raise the priority of a process whenever its resources are preempted. It should eventually gain a high enough priority that it will no longer be preempted.