Operating System CE0418

Unit-2

Concurrency

Prof. Indr Jeet Rajput

Assistant Professor CE & IT

Concurrency

- Concurrency is the execution of the multiple instruction sequences at the same time.
- It happens in the operating system when there are several process threads running in parallel.
- The running process threads always communicate with each other through shared memory or message passing.
- Concurrency results in sharing of resources result in problems like deadlocks and resources starvation.
- It helps in techniques like coordinating execution of processes, memory allocation and execution scheduling for maximizing throughput.

Principles of Concurrency:

- Today's technology, like multi-core processors and parallel processing, allows multiple processes and threads to be executed simultaneously.
- Multiple processes and threads can access the same memory space, the same declared variable in code, or even read or write to the same file.

Concurrency

- The amount of time it takes a process to execute cannot be simply estimated, and you cannot predict which process will complete first, enabling you to build techniques to deal with the problems that concurrency creates.
- Interleaved and overlapping processes are two types of concurrent processes with the same problems. It is impossible to predict the relative speed of execution, and the following factors determine it:
  - The way operating system handles interrupts
  - Other processes' activities
  - The operating system's scheduling policies

Problems in Concurrency: There are various problems in concurrency. Some of them are as follows:

1. Locating the programming errors:

- It's difficult to spot a programming error because reports are usually repeatable due to the varying states of shared components each time the code is executed.

2. Sharing Global Resources

- Sharing global resources is difficult. If two processes utilize a global variable and both alter the variable's value, the order in which the many changes are executed is critical.

Concurrency

3. Locking the channel

- It could be inefficient for the OS to lock the resource and prevent other processes from using it.

4. Optimal Allocation of Resources

- It is challenging for the OS to handle resource allocation properly.

Issues of Concurrency

Various issues of concurrency are as follows:

1. Non-atomic:

- Operations that are non-atomic but interruptible by several processes may happen issues. A non-atomic operation depends on other processes, and an atomic operation runs independently of other processes.

2. Deadlock:

- In concurrent computing, it occurs when one group member waits for another member, including itself, to send a message and release a lock.
- Software and hardware locks are commonly used to arbitrate shared resources and implement process synchronization in parallel computing, distributed systems, and multiprocessing.

3. Blocking:

- A blocked process is waiting for some event, like the availability of a resource or completing an I/O operation.
- Processes may block waiting for resources, and a process may be blocked for a long time waiting for terminal input.
- If the process is needed to update some data periodically, it will be very undesirable.

Concurrency

4. Race Conditions:

- A race problem occurs when the output of a software application is determined by the timing or sequencing of other uncontrollable events.
- Race situations can also happen in multithreaded software, runs in a distributed environment, or is interdependent on shared resources.

5. Starvation:

- A problem in concurrent computing is where a process is continuously denied the resources it needs to complete its work.
- It could be caused by errors in scheduling or mutual exclusion algorithm, but resource leaks may also cause it.
- Concurrent system design frequently requires developing dependable strategies for coordinating their execution, data interchange, memory allocation, and execution schedule to decrease response time and maximize throughput.

Concurrency

Advantages:

1. Better Performance:

- It improves the operating system's performance.
- When one application only utilizes the processor, and another only uses the disk drive, the time it takes to perform both apps simultaneously is less than the time it takes to run them sequentially.

2. Better Resource Utilization:

- It enables resources that are not being used by one application to be used by another.

3. Running Multiple Applications:

- It enables you to execute multiple applications simultaneously.

Disadvantages:

- It is necessary to protect multiple applications from each other.
- It is necessary to use extra techniques to coordinate several applications.
- Additional performance overheads and complexities in OS are needed for switching between applications..

Race Condition

- A race condition is a situation that may occur inside a critical section.
- This happens when the result of multiple thread execution in critical section differs according to the order in which the threads execute.

- Race conditions in critical sections can be avoided if the critical section is treated as an atomic instruction.
- Also, proper thread synchronization using locks or atomic variables can prevent race conditions.

Critical Section

- The critical section in a code segment where the shared variables can be accessed.
- 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.
- The critical section is given as follows:

```
do{
                    Entry Section
                    Critical Section
                    Exit Section
                    Remainder Section
          } while (TRUE);
```

- In the above Instruction, the entry sections 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 critical section is free.

Critical Section

- The critical section problem needs a solution to synchronise 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.

Progresss:

- 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 Waitings:

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

Peterson's Solution

- Peterson's solution is a software-based solution to the critical-section problem.
- Peterson's solution is restricted to two processes that alternate execution between their critical sections and remainder sections.
- The processes are numbered **P0** and **P1** .
- For convenience, when presenting **Pi** , we use **Pj** to denote the other process; that is, j equals 1 − i.
- Peterson's solution requires the two processes to share two data items:

```
 int turn;
boolean flag[2];
```

- The variable **turn** indicates whose turn it is to enter its critical section.
- That is, **if turn == i**, then process **Pi** is allowed to execute in its critical section.
- The **flag** array is used to indicate if a process is ready to enter its critical section.
- For example, if **flag[i] is true**, this value indicates that Pi is ready to enter its critical section.

Peterson’s Solution

- With an explanation of these data structures complete, we are now ready to describe the algorithm shown Below.
- To enter the critical section, process **Pi** first sets **flag[i]** to be true and then sets **tur**n to the value **j**, thereby asserting that if the other process wishes to enter the critical section, it can do so.
- If both processes try to enter at the same time, turn will be set to both i and j at roughly the same time.
- Only one of these assignments will last; the other will occur but will be overwritten immediately.
- The eventual value of turn determines which of the two processes is allowed to enter its critical section first.

```
do {      flag[i] = true;
          turn = j;
          while (flag[j] && turn == j);
                    critical section
        flag[i] = false;
                    remainder section
} while (true);
```

Peterson’s Solution

- We now prove that this solution is correct. We need to show that:

1. Mutual exclusion is preserved.
2. The progress requirement is satisfied.
3. The bounded-waiting requirement is met.

- **To prove property 1,** we note that each Pi enters its critical section only if either flag[j] == false or turn == i.
- Also note that, if both processes can be executing in their critical sections at the same time, then flag[0] ==flag[1] == true.
- These two observations imply that P0 and P1 could not have successfully executed their while statements at about the same time, since the value of turn can be either 0 or 1 but cannot be both.
- Hence, one of the processes —say, P j —must have successfully executed the while statement, whereas Pi had to execute at least one additional statement (“turn == j”).
- However, at that time, flag[j] == true and turn == j, and this condition will persist as long as Pj is in its critical section; as a result, mutual exclusion is preserved

Peterson's Solution

- To prove properties 2 and 3, we note that a process **Pi** can be prevented from entering the critical section only if it is stuck in the while loop with the condition **flag[j] == true** and **turn == j**; this loop is the only one possible.
- If **Pj** is not ready to enter the critical section, then **flag[j] == false**, and **Pi** can enter its critical section.
- If **Pj** has set **flag[j]** to true and is also executing in its while statement, then either **turn == i** or **turn == j**.
- If **turn == i**, then **Pi** will enter the critical section.
- If **turn == j**, then **Pj** will enter the critical section.
- However, once **Pj** exits its critical section, it will reset flag[j] to false, allowing **Pi** to enter its critical section.
- If **Pj** resets **flag[j]** to true, it must also set turn to **i**.
- Thus, since **Pi** does not change the value of the variable turn while executing the while statement, **Pi** will enter the critical section (progress) after at most one entry by **Pj** (bounded waiting).

Mutex Locks

- Operating-systems designers build software tools to solve the critical-section problem.
- The simplest of these tools is the mutex lock. (In fact, the term mutex is short for mutual exclusion.)
- We use the mutex lock to protect critical regions and thus prevent race conditions.
- That is, a process must acquire the lock before entering a critical section; it releases the lock when it exits the critical section.
- The **acquire()** function acquires the lock, and the **release()** function releases the lock.
- A mutex lock has a boolean variable available whose value indicates if the lock is available or not.
- If the lock is available, a call to **acquire()** succeeds, and the lock is then considered unavailable.
- A process that attempts to acquire an unavailable lock is blocked until the lock is released.

Mutex Locks

- **The definition of acquire() is as follows:**

```
acquire() {
        while (!available); /* busy wait */
       available = false;;
      }
do {
      acquire lock
      critical section
      release lock
     remainder section
  } while (true);
```

The definition of release() is as follows:

```
release() {
        available = true;
      }
```

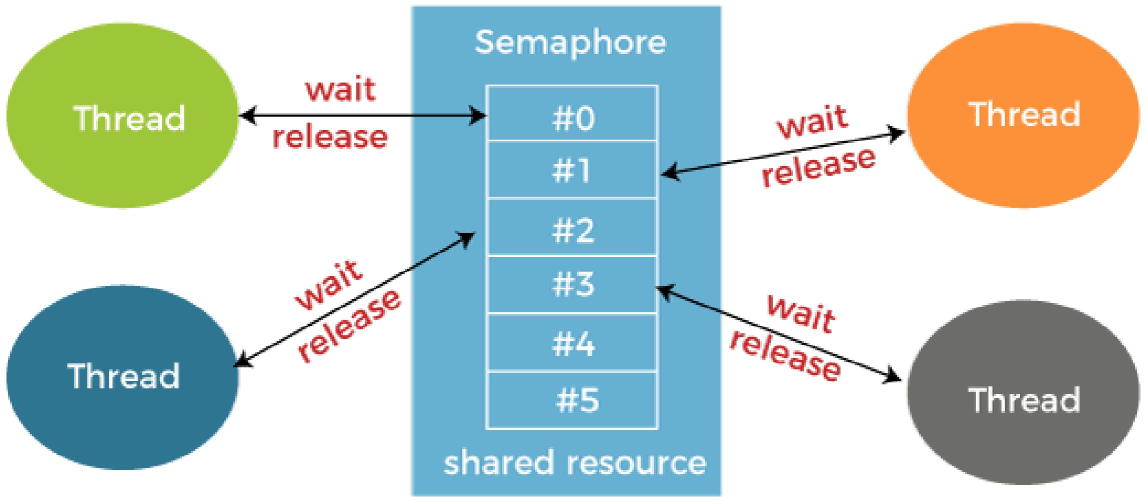
Mutex Locks

Advantages of Mutex:

- Mutex is just simple locks obtained before entering its critical section and then releasing it.
- Since only one thread is in its critical section at any given time, there are no race conditions, and data always remain consistent.
- Disadvantages of Mutex
- If a thread obtains a lock and goes to sleep or is preempted, then the other thread may not move forward. This may lead to starvation.
- It can't be locked or unlocked from a different context than the one that acquired it.
- Only one thread should be allowed in the critical section at a time.
- The normal implementation may lead to a busy waiting state, which wastes CPU time.

What is Semaphore

- Semaphore is simply a variable that is non-negative and shared between threads.
- A semaphore is a signaling mechanism, and another thread can signal a thread that is waiting on a semaphore.



What is Semaphore

A semaphore uses two atomic operations:

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<=0);
  S--;
}
```

2. Signal for the process synchronization: The signal operation increments the value of its argument S.

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

What is Semaphore

Types of Semaphore:

1. Counting Semaphore: The semaphore S value is initialized to the number of resources present in the system.

- Whenever a process wants to access the resource, it performs the wait() operation on the semaphore and decrements the semaphore value by one.
- When it releases the resource, it performs the signal() operation on the semaphore and increments the semaphore value by one.
- When the semaphore count goes to 0, it means the processes occupy all resources.
- A process needs to use a resource when the semaphore count is 0.
- It executes the wait() operation and gets blocked until the semaphore value becomes greater than 0.

2. Binary semaphore: The value of a semaphore ranges between 0and 1.

- It is similar to mutex lock, but mutex is a locking mechanism, whereas the semaphore is a signaling mechanism.
- In binary semaphore, if a process wants to access the resource, it performs the wait() operation on the semaphore and decrements the value of the semaphore from 1 to 0

What is Semaphore

- When it releases the resource, it performs a signal() operation on the semaphore and increments its value to 1.
- Suppose the value of the semaphore is 0 and a process wants to access the resource.
- In that case, it performs wait() operation and block itself till the current process utilizing the resources releases the resource.

Advantages of Semaphore:

- It allows more than one thread to access the critical section.
- Semaphores are machine-independent.
- Semaphores are implemented in the machine-independent code of the microkernel.
- They do not allow multiple processes to enter the critical section.
- As there is busy and waiting in semaphore, there is never wastage of process time and resources.

What is Semaphore

Disadvantage of Semaphores:

- One of the biggest limitations of a semaphore is priority inversion.
- The operating system has to keep track of all calls to wait and signal semaphore.
- Their use is never enforced, but it is by convention only.
- The Wait and Signal operations require to be executed in the correct order to avoid deadlocks in semaphore.
- Semaphore programming is a complex method, so there are chances of not achieving mutual exclusion.

Monitors

- It is a synchronization technique that enables threads to mutual exclusion and the wait() for a given condition to become true.
- It is an abstract data type.
- It has a shared variable and a collection of procedures executing on the shared variable.
- A process may not directly access the shared data variables, and procedures are required to allow several processes to access the shared data variables simultaneously.
- At any particular time, only one process may be active in a monitor.
- Other processes that require access to the shared variables must queue and are only granted access after the previous process releases the shared variables.

Monitors

The syntax of the monitor may be used as:

```
monitor monitorName
    {   data variables;
            Procedure P1(....)  {      }
            Procedure P2(....)  {      }

           Procedure Pn(....)   {        }

        Initialization Code(....)   {   }
   }
```

- Monitor in an operating system is simply a class containing variable_declarations, condition_variables, various procedures (functions), and an initializing_code block that is used for process synchronization.

Monitors

Characteristics of Monitors in OS:

- We can only run one program at a time inside the monitor.
- Monitors in an operating system are defined as a group of methods and fields that are combined with a special type of package in the os.
- A program cannot access the monitor's internal variable if it is running outside the monitor. Although, a program can call the monitor's functions.
- Monitors were created to make synchronization problems less complicated.
- Monitors provide a high level of synchronization between processes.

Components of Monitor in an operating system:

1. **Initialization:** The code for initialization is included in the package, and we just need it once when creating the monitors.
2. **Private Data:** It is a feature of the monitor in an operating system to make the data private.
   - It holds all of the monitor's secret data, which includes private functions that may only be utilized within the monitor.
   - As a result, private fields and functions are not visible outside of the monitor.

Monitors

3. Monitor Procedure: Procedures or functions that can be invoked from outside of the monitor are known as monitor procedures.

4. Monitor Entry Queue: Another important component of the monitor is the Monitor Entry Queue. It contains all of the threads, which are commonly referred to as procedures only.

Condition Variables: There are two sorts of operations we can perform on the monitor's condition variables:

1. **Wait**
2. **Signal**

- Consider a condition variable (y) is declared in the monitor:

y.wait(): The activity/process that applies the wait operation on a condition variable will be suspended, and the suspended process is located in the condition variable's block queue.

y.signal(): If an activity/process applies the signal action on the condition variable, then one of the blocked activity/processes in the monitor is given a chance to execute.

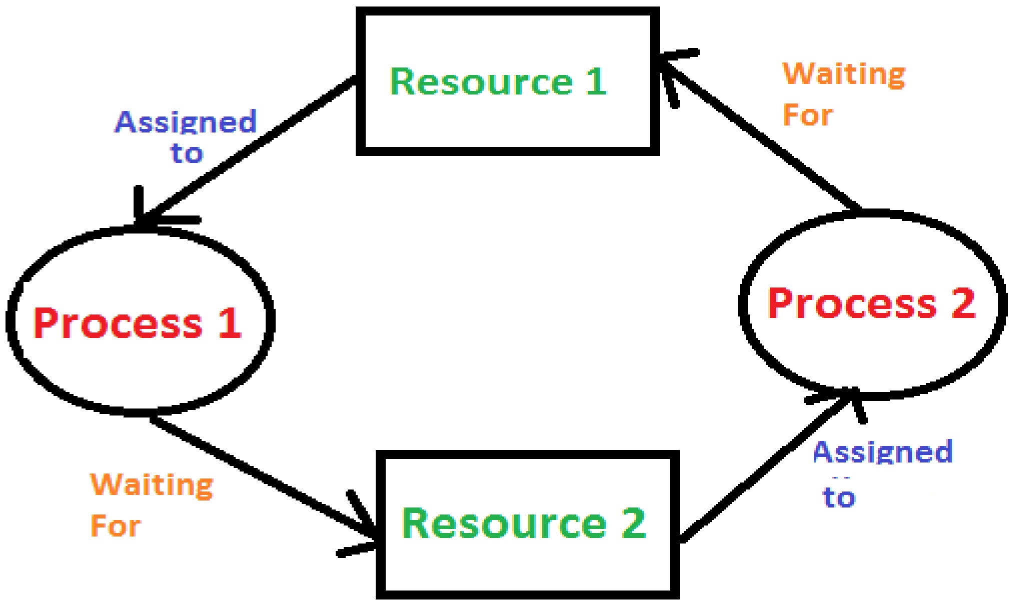
Monitors

Advantages:

- Mutual exclusion is automatic in monitors.
- Monitors are less difficult to implement than semaphores.
- Monitors may overcome the timing errors that occur when semaphores are used.
- Monitors are a collection of procedures and condition variables that are combined in a special type of module.

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.
- For example, in the below diagram, Process 1 is holding Resource 1 and waiting for resource 2 which is acquired by process 2, and process 2 is waiting for resource 1.

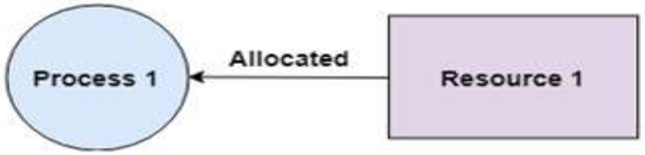


Deadlock

- A deadlock occurs if the four Coffman conditions hold true

Mutual Exclusion :

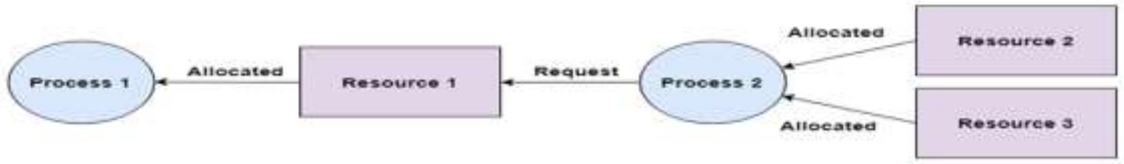
- There should be a resource that can only be held by one process at a time.
- A resource can only be shared in mutually exclusive manner. It implies, if two process cannot use the same resource at the same time.
- In the diagram below, there is a single instance of Resource 1 and it is held by Process 1 only.



Hold and Wait:

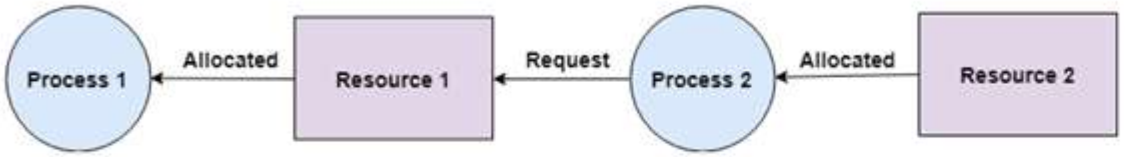
- A process can hold multiple resources and still request more resources from other processes which are holding them.

Deadlock

- 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.
- 

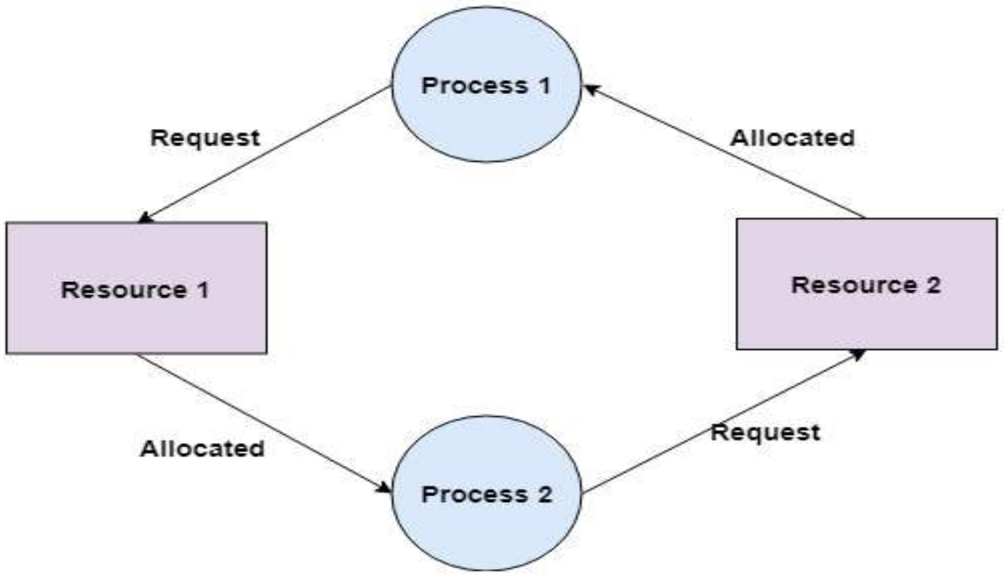
No Preemption:

- 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.



Deadlock

- 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.



Methods for handling Deadlock

Prevention:

- The idea is to not let the system into a deadlock state.
- This system will make sure that above mentioned four conditions will not arise.
- These techniques are very costly so we use this in cases where our priority is making a system deadlock-free.
- One can zoom into each category individually, Prevention is done by negating one of above mentioned necessary conditions for deadlock.
- Prevention can be done in four different ways:

1. Eliminate mutual exclusion
2. Solve hold and Wait
3. Allow preemption
4. Circular wait Solution

Methods for handling Deadlock

Deadlock Avoidance:

- When a process requests a resource, the deadlock avoidance algorithm examines the resource-allocation state.
- If allocating that resource sends the system into an unsafe state, the request is not granted.
- Therefore, it requires additional information such as how many resources of each type is required by a process.
- If the system enters into an unsafe state, it has to take a step back to avoid deadlock.

Deadlock Detection and Recovery:

- We let the system fall into a deadlock and if it happens, we detect it using a detection algorithm and try to recover.
- Some ways of recovery are as follows:
- Aborting all the deadlocked processes.
- Abort one process at a time until the system recovers from the deadlock.
- Resource Preemption: Resources are taken one by one from a process and assigned to higher priority processes until the deadlock is resolved.

Methods for handling Deadlock

- Deadlock Ignorance
- If a deadlock is very rare, then let it happen and reboot the system.
- This is the approach that both Windows and UNIX take.
- This approach is best suitable for a single end user system where User uses the system only for browsing and all other normal stuff.

Handling Deadlock using Prevention

- **Let's see how we can prevent each of the conditions:**

Mutual Exclusion:

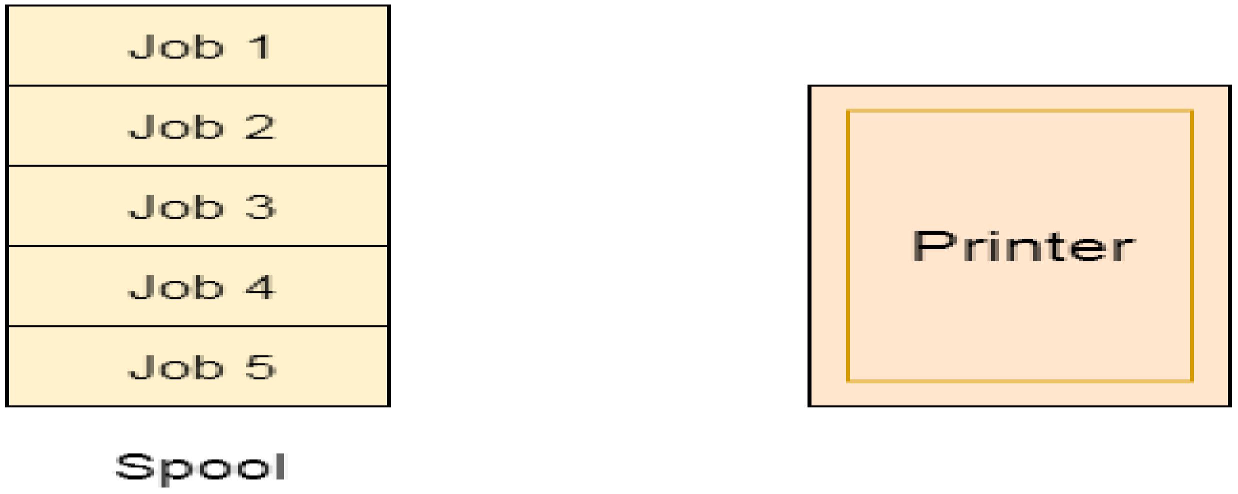
- Mutual section from the resource point of view is the fact that a resource can never be used by more than one process simultaneously which is fair enough but that is the main reason behind the deadlock.
- If a resource could have been used by more than one process at the same time then the process would have never been waiting for any resource.
- However, if we can be able to violate resources behaving in the mutually exclusive manner then the deadlock can be prevented.

Spooling

- For a device like printer, spooling can work.
- There is a memory associated with the printer which stores jobs from each of the process into it.
- Later, Printer collects all the jobs and print each one of them according to FCFS.
- By using this mechanism, the process doesn't have to wait for the printer and it can continue whatever it was doing.
- Later, it collects the output when it is produced.

Handling Deadlock using Prevention

- Although, Spooling can be an effective approach to violate mutual exclusion but it suffers from two kinds of problems.
  I. This cannot be applied to every resource.
  II. After some point of time, there may arise a race condition between the processes to get space in that spool.
-



Handling Deadlock using Prevention

Hold and Wait:

- Hold and wait condition lies when a process holds a resource and waiting for some other resource to complete its task.
- Deadlock occurs because there can be more than one process which are holding one resource and waiting for other in the cyclic order.
- However, we have to find out some mechanism by which a process either doesn't hold any resource or doesn't wait.
- That means, a process must be assigned all the necessary resources before the execution starts.
- A process must not wait for any resource once the execution has been started.
- This can be implemented practically if a process declares all the resources initially.
- However, this sounds very practical but can't be done in the computer system because a process can't determine necessary resources initially.

Handling Deadlock using Prevention

No Preemption:

- Deadlock arises due to the fact that a process can't be stopped once it starts.
- However, if we take the resource away from the process which is causing deadlock then we can prevent deadlock.
- This is not a good approach at all since if we take a resource away which is being used by the process then all the work which it has done till now can become inconsistent.

Circular Wait:

- To violate circular wait, we can assign a priority number to each of the resource.
- A process can't request for a lesser priority resource.
- This ensures that not a single process can request a resource which is being utilized by some other process and no cycle will be formed.

Handling Deadlock using **Avoidance**

- In deadlock avoidance, the request for any resource will be granted if the resulting state of the system doesn't cause deadlock in the system.
- The state of the system will continuously be checked for safe and unsafe states.
- In order to avoid deadlocks, the process must tell OS, the maximum number of resources a process can request to complete its execution.
- The simplest and most useful approach states that the process should declare the maximum number of resources of each type it may ever need.
- The Deadlock avoidance algorithm examines the resource allocations so that there can never be a circular wait condition.

Safe and Unsafe States:

- The resource allocation state of a system can be defined by the instances of available and allocated resources, and the maximum instance of the resources demanded by the processes.

Handling Deadlock using **Avoidance**

- A state of a system recorded at some random time is shown below.
- Resources Assigned

Process	Type 1	Type 2	Type 3	Type 4
A	3	0	2	2
B	0	0	1	1
C	1	1	1	0
D	2	1	4	0

Resources still needed

Process	Type 1	Type 2	Type 3	Type 4
A	1	1	0	0
B	0	1	1	2
C	1	2	1	0
D	2	1	1	2

total instances of each resource in the system (E) = (7 6 8 4)

Instances of resources that have been assigned to processes (P) = (6 2 8 3)

Represents the number of resources that are not in use (A) = (1 4 0 1)

- A state of the system is called **safe** if the system can allocate all the resources requested by all the processes without entering into deadlock.
- If the system cannot fulfill the request of all processes then the state of the system is called **unsafe.**

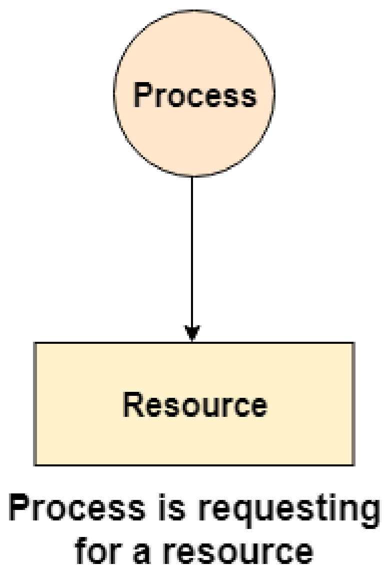
Handling Deadlock using **Avoidance**

Resource Allocation Graph :

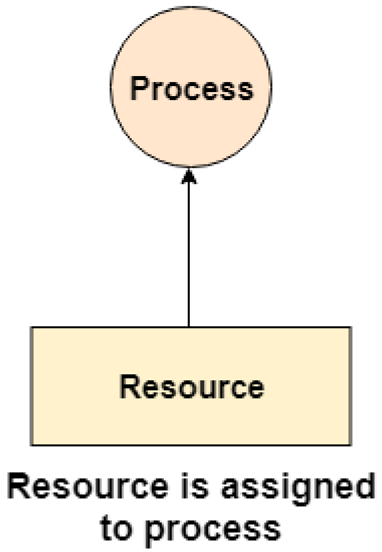
- The resource allocation graph is the pictorial representation of the state of a system.
- As its name suggests, the resource allocation graph is the complete information about all the processes which are holding some resources or waiting for some resources.
- It also contains the information about all the instances of all the resources whether they are available or being used by the processes.
- In Resource allocation graph, the process is represented by a Circle while the Resource is represented by a rectangle. Let's see the types of vertices and edges in detail.
- Vertices are mainly of two types, Resource and process.
- Each of them will be represented by a different shape.
- Circle represents **process** while **rectangle** represents resource.
- A resource can have more than one instance.
- Each instance will be represented by a **dot** inside the rectangle.

Handling Deadlock using **Avoidance**

Resource Allocation Graph :



Process is requesting for a resource



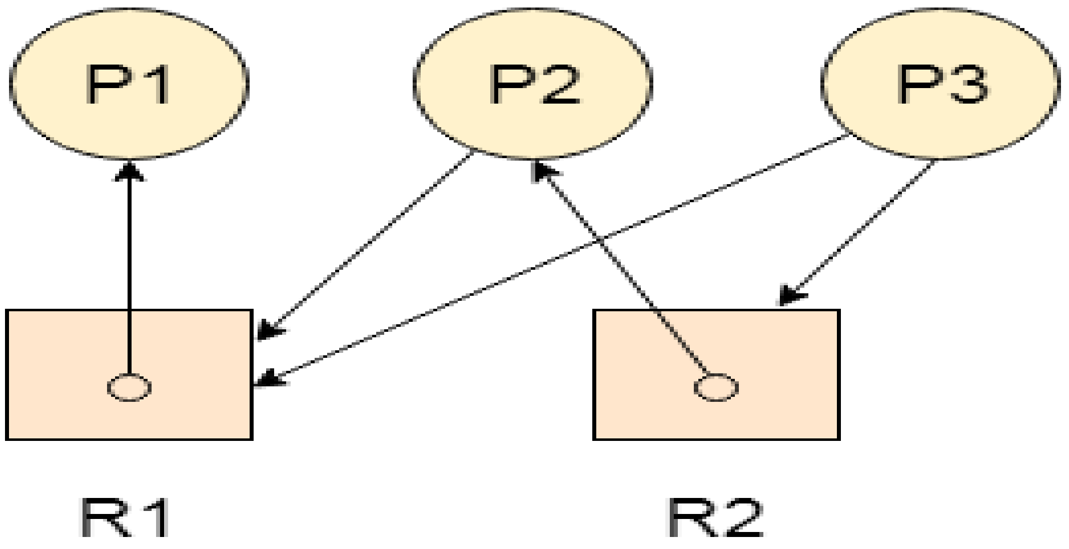
Resource is assigned to process

Handling Deadlock using **Avoidance**

Resource Allocation Graph :

Example**:**

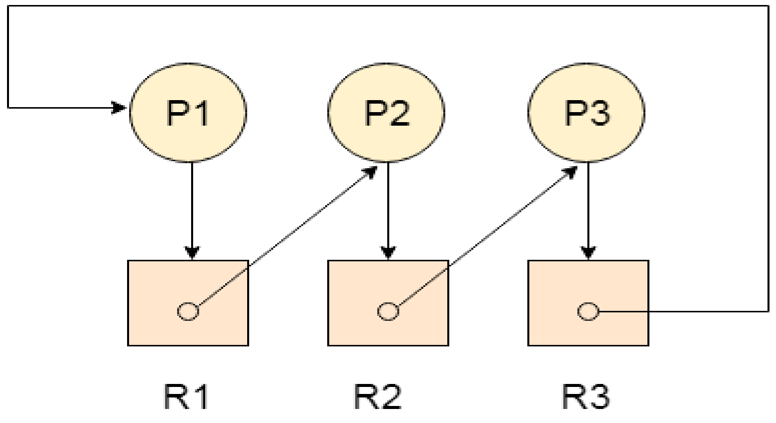
- Let'sconsider 3 processes P1, P2 and P3, and two types of resources R1 and R2. The resources are having 1 instance each.
- According to the graph, R1 is being used by P1, P2 is holding R2 and waiting for R1, P3 is waiting for R1 as well as R2.
- The graph is deadlock free since no cycle is being formed in the graph.



Handling Deadlock using **Avoidance**

Deadlock Detection using RAG :

- If a cycle is being formed in a Resource allocation graph where all the resources have the single instance then the system is deadlocked.
- In Case of Resource allocation graph with multi-instanced resource types, Cycle is a necessary condition of deadlock but not the sufficient condition.
- The following example contains three processes P1, P2, P3 and three resources R1, R2, R3. All the resources are having single instances each.



- If we analyze the graph then we can find out that there is a cycle formed in the graph since the system is satisfying all the four conditions of deadlock.

Handling Deadlock using **Avoidance**

Deadlock Detection using RAG:

Allocation Matrix:

Process	R1	R2	R3
P1	0	0	1
P2	1	0	0
P3	0	1	0

Request Matrix:

Process	R1	R2	R3
P1	1	0	0
P2	0	1	0
P3	0	0	1

Avial = (0,0,0)

- Neither we are having any resource available in the system nor a process going to release.
- Each of the process needs at least single resource to complete therefore they will continuously be holding each one of them.
- We cannot fulfill the demand of at least one process using the available resources therefore the system is deadlocked as determined earlier when we detected a cycle in the graph.

Handling Deadlock using **Avoidance**

Banker's Algorithm in Operating System (OS) :

- It is a banker algorithm used to avoid deadlock and allocate resources safely to each process in the computer system.
- When a new process is created in a computer system, the process must provide all types of information to the operating system like upcoming processes, requests for their resources, counting them, and delays.
- Based on these criteria, the operating system decides which process sequence should be executed or waited so that no deadlock occurs in a system.
- Therefore, it is also known as deadlock avoidance algorithm or deadlock detection in the operating system.
- When working with a banker's algorithm, it requests to know about three things:

1. How much each process can request for each resource in the system. It is denoted by the **[MAX]** request.
2. How much each process is currently holding each resource in a system. It is denoted by the **[ALLOCATED]** resource.
3. It represents the number of each resource currently available in the system. It is denoted by the **[AVAILABLE]** resource.

Handling Deadlock using **Avoidance**

Following are the important data structures terms applied in the banker's algorithm as follows:

- **Suppose n is the number of processes, and m is the number of each type of resource used in a computer system.**
  1. **Available:** It is an array of length 'm' that defines each type of resource available in the system. When Available[j] = K, means that 'K' instances of Resources type R[j] are available in the system.
  2. **Max:** It is a [n x m] matrix that indicates each process P[i] can store the maximum number of resources R[j] (each type) in a system.
  3. **Allocation:** It is a matrix of m x n orders that indicates the type of resources currently allocated to each process in the system. When Allocation [i, j] = K, it means that process P[i] is currently allocated K instances of Resources type R[j] in the system.
  4. **Need:** It is an M x N matrix sequence representing the number of remaining resources for each process. When the Need[i] [j] = k, then process P[i] may require K more instances of resources type Rj to complete the assigned work.
  5. **Nedd[i][j] = Max[i][j] - Allocation[i][j].**
  6. **Finish:** It is the vector of the order m. It includes a Boolean value (true/false) indicating whether the process has been allocated to the requested resources, and all resources have been released after finishing its task.

Handling Deadlock using **Avoidance**

The Banker's Algorithm is the combination of the safety algorithm and the resource request algorithm to control the processes and avoid deadlock in a system:

Safety Algorithm: It is a safety algorithm used to check whether or not a system is in a safe state or follows the safe sequence in a banker's algorithm.

1. There are two vectors Wok and Finish of length m and n in a safety algorithm.

Initialize: Work = Available

Finish[i] = false; for I = 0, 1, 2, 3, 4… n - 1.

2. Check the availability status for each type of resources [i], such as:

Need[i] <= Work

Finish[i] == false

If the i does not exist, go to step 4.

3. Work = Work +Allocation(i) // to get new resource allocation

Finish[i] = true

Go to step 2 to check the status of resource availability for the next process.

4. If Finish[i] == true; it means that the system is safe for all processes.

Handling Deadlock using **Avoidance**

Resource Request Algorithm: A resource request algorithm checks how a system will behave when a process makes each type of resource request in a system as a request matrix.

- Let create a resource request array R[i] for each process P[i]. If the Resource Requesti [j] equal to 'K', which means the process P[i] requires 'k' instances of Resources type R[j] in the system

1. When the number of requested resources of each type is less than the Need resources, go to step 2 and if the condition fails, which means that the process P[i] exceeds its maximum claim for the resource. As the expression suggests:

- If Request(i) <= Need
- Go to step 2;

2. And when the number of requested resources of each type is less than the available resource for each process, go to step (3). As the expression suggests:

- If Request(i) <= Available
- Else Process P[i] must wait for the resource since it is not available for use.

Handling Deadlock using **Avoidance**

3. When the requested resource is allocated to the process by changing state:

Available = Available - Request

Allocation(i) = Allocation(i) + Request (i)

Needi = Needi - Requesti

When the resource allocation state is safe, its resources are allocated to the process P(i). And if the new state is unsafe, the Process P (i) has to wait for each type of Request R(i) and restore the old resource-allocation state.

Question and Answer

1.