UNIT-IV

CONCURRENCY

Process Synchronization, The Critical- Section Problem, Synchronization Hardware, Semaphores, Classic Problems of Synchronization, Monitors, Synchronization Examples.

PROCESS SYNCHRONIZATION:

A co-operation process is one that can affect or be affected by other processes executing in the system. Co-operating process may either directly share a logical address space or be allotted to the shared data only through files. This concurrent access is known as Process synchronization.

Critical Section Problem:

Consider a system consisting of n processes (P0, P1, ………Pn -1) each process has a segment of code which is known as critical section in which the process may be changing common variable, updating a table, writing a file and so on. The important feature of the system is that when the process is executing in its critical section no other process is to be allowed to execute in its critical section. The execution of critical sections by the processes is a mutually exclusive. The critical section problem is to design a protocol that the process can use to cooperate each process must request permission to enter its critical section. The section of code implementing this request is the entry section. The critical section is followed on exit section. The remaining code is the remainder section.

Example:

```
While (1)
{
        Entry Section;
        Critical Section;
        Exit Section;
        Remainder Section;
}
```

A solution to the critical section problem must satisfy the following three conditions.

1. **Mutual Exclusion:** If process Pi is executing in its critical section then no any other process can be executing in their critical section.

2. **Progress:** If no process is executing in its critical section and some process wish to enter their critical sections then only those process that are not executing in their remainder section can enter its critical section next.

3. **Bounded waiting:** There exists a bound on the number of times that other processes are allowed to enter their critical sections after a process has made a request.

Semaphores:

For the solution to the critical section problem one synchronization tool is used which is known as semaphores. A semaphore S is an integer variable which is accessed through two standard operations such as wait and signal. These operations were originally termed P (for wait means to test) and V (for signal means to increment). The classical definition of wait is

```
Wait (S)
{
    While (S <=   0)
        {
            Test;
        }
    S--;
}
```

The classical definition of the signal is

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

In case of wait the test condition is executed with interruption and the decrement is executed without interruption.

Binary Semaphore:

A binary semaphore is a semaphore with an integer value which can range between 0 and 1.

Let S‘ be a counting semaphore. To implement the binary semaphore we need following the structure of data.

Binary Semaphores S1, S2;

int C;

Initially S1 = 1, S2 = 0 and the value of C is set to the initial value of the counting semaphore S.

Then the wait operation of the binary semaphore can be implemented as follows.

```
    Wait (S1)
    C--;
    if (C < 0)
    {
        Signal (S1);
        Wait (S2);
  }
  Signal (S1);
```

The signal operation of the binary semaphore can be implemented as follows:

```
Wait (S1);
        C++;
        if (C <=0)
                Signal (S2);
        Else
                Signal (S1);
```

Classical Problem on Synchronization:

There are various types of problem which are proposed for synchronization scheme such as

Bounded Buffer Problem:

This problem was commonly used to illustrate the power of synchronization primitives. In this scheme we assumed that the pool consists of _N‘ buffer and each capable of holding one item. The _mutex‘ semaphore provides mutual exclusion for access to the buffer pool and is initialized to the value one. The empty and full semaphores count the number of empty and full buffer respectively. The semaphore empty is initialized to _N‘ and the semaphore full is initialized to zero. This problem is known as procedure and consumer problem. The code of the producer is producing full buffer and the code of consumer is producing empty buffer.

THE STRUCTURE OF PRODUCER PROCESS IS AS FOLLOWS:

```
do
{
        produce an item in nextp
        . . . . . . . . . . . .
        Wait (empty);
        Wait (mutex);
        . . . . . . . . . . .
        add nextp to buffer
        . . . . . . . . . . . .
        Signal (mutex);
        Signal (full);
} While (1);
```

THE STRUCTURE OF CONSUMER PROCESS IS AS FOLLOWS:

```
do
{
        Wait (full);
        Wait (mutex);
        . . . . . . . . . . .
        Remove an item from buffer to nextc
        . . . . . . . . . . .
        Signal (mutex);
        Signal (empty);
        . . . . . . . . . . . .
        Consume the item in nextc;
        . . . . . . . .. . . .. .
} While (1);
```

Reader Writer Problem:

In this type of problem there are two types of process are used such as Reader process and Writer process. The reader process is responsible for only reading and the writer process is responsible for writing. This is an important problem of synchronization which has several variations like

o The simplest one is referred as first reader writer problem which requires that no reader will be kept waiting unless a writer has obtained permission to use the shared object. In other words no reader should wait for other reader to finish because a writer is waiting.

o The second reader writer problem requires that once a writer is ready then the writer performs its write operation as soon as possible.

THE STRUCTURE OF A READER PROCESS IS AS FOLLOWS:

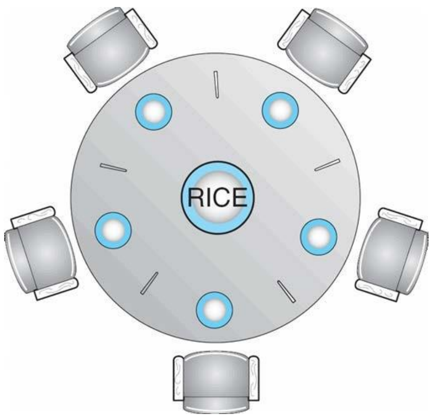
```
Wait (mutex);
Read count++;
if (read count == 1)
        Wait (wrt);
Signal (mutex);
        . . . . . . . . . . .
Reading is performed
        . . . . . . . . . . .
Wait (mutex);
Read count --;
if (read count == 0)
        Signal (wrt);
Signal (mutex);
```

THE STRUCTURE OF THE WRITER PROCESS IS AS FOLLOWS:

```
Wait (wrt);
        Writing is performed;
Signal (wrt);
```

Dining Philosopher Problem:

Consider 5 philosophers to spend their lives in thinking & eating. A philosopher shares common circular table surrounded by 5 chairs each occupies by one philosopher. In the center of the table there is a bowl of rice and the table is laid with 6 chopsticks as shown in below figure. When a philosopher thinks she does not interact with her colleagues. From time to time a philosopher gets hungry and tries to pickup two chopsticks that are closest to her. A philosopher may pickup one chopstick or two chopsticks at a time but she cannot pickup a chopstick that is already in hand of the neighbor. When a hungry philosopher has both her chopsticks at the same time, she eats without releasing her chopsticks. When she finished eating, she puts down both of her chopsticks and starts thinking again. This problem is considered as classic synchronization problem. According to this problem each chopstick is represented by a semaphore. A philosopher grabs the chopsticks by executing the wait operation on that semaphore. She releases the chopsticks by executing the signal operation on the appropriate semaphore.



THE STRUCTURE OF DINING PHILOSOPHER IS AS FOLLOWS:

```
do
{
            Wait ( chopstick [i]);
            Wait (chopstick [(i+1)%5]);
             . . . . . . . . . . . . .
            Eat
             . . . . . . . . . . . . .
            Signal (chopstick [i]);
            Signal (chopstick [(i+1)%5]);
            . . . . . . . . . . . . .
             Think
            . . . . . . . . . . . . .
} While (1);
```

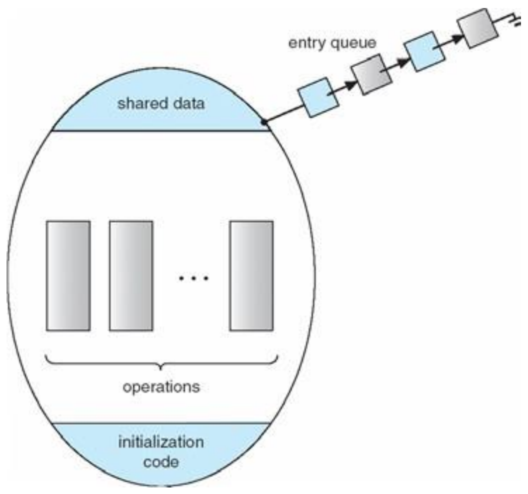
Monitors:

- ❖ Monitors are based on abstract data types.
- ❖ A monitor is a programming language construct that provides equivalent functionality to that of semaphores but is easier to control.
- ❖ A high-level abstraction that provides a convenient and effective mechanism for process synchronization Only one process may be active within the monitor at a time.
- ❖ A monitor consists of procedures, the shared object and administrative data.

Characteristics of Monitors are as follows:

1. Only one process can be active within the monitor at a time.
2. The local data variables are accessible only by the monitors procedures and not by any external procedure.
3. A process enters the monitors by invoking one of its procedures.

- ❖ Monitor provides high level of synchronization. The synchronization of process is accomplished via two special operations namely, wait and signal, which are executed within the monitors procedures.
- ❖ Monitors are a high level data abstraction tool combining three features:
  1. Shared Data
  2. Operation on Data
  3. Synchronization, Scheduling
- ❖ A monitor is characterized by a set of programmer defined operators. Monitors were derived to simplify the complexity of synchronization problems. Every synchronization problem that can be solved with monitors can be also be solved with semaphores and vice versa.
- ❖ Monitor is an abstract data type for which only one process may be executing a procedure at any given time.
- ❖ Process desiring to enter the monitor when it is already in use must wait. This waiting is automatically managed by the monitor.
- ❖ **Schematic view of a Monitor**



- A monitor is a software module consisting of one or more procedures, an initialization sequence and local data.

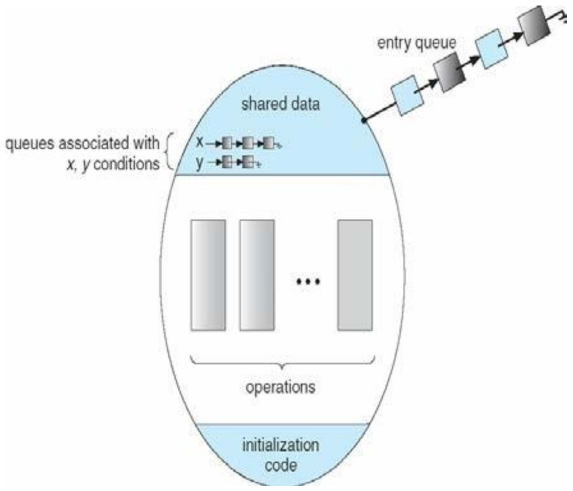
Syntax of the Monitor:

```
monitor monitor-name
{
// shared variable declarations
procedure P1 (…)
{
            ….
            ….
}
…
procedure Pn (…)
{
            ……
            …. ..
}
            { …
                  Initialization code ( ….)
            }
…
}
```

- A monitor supports synchronization by the use of condition variables that are contained within the monitor and accessible only within the monitor. Two **Condition Variables are:**

1. x.wait () – a process that invokes the operation is suspended.
2. x.signal () – resumes one of processes (if any) that invoked x.wait ()

Monitor with Condition Variables



DEADLOCKS

When processes request a resource and if the resources are not available at that time the process enters into waiting state. Waiting process may not change its state because the resources they are requested are held by other process. This situation is called deadlock. The situation where the process waiting for the resource i.e., not available is calleddeadlock.

System Model

A system may consist of finite number of resources and is distributed among number of processes. There resources are partitioned into several instances each with identical instances,

A process must request a resource before using it and it must release the resource after using it. It can request any number of resources to carry out a designated task. The amount of resource requested may not exceed the total number of resources available.

A process may utilize the resources in only the following sequences:

1. Request: If the request is not granted immediately then the requesting process must wait it can acquire the resources.
2. Use:- The process can operate on the resource.
3. Release:- The process releases the resource after using it.

Deadlock may involve different types of resources.

For eg:- Consider a system with one printer and one tape drive. Ifa process Pi currently holds a printer and a process Pj holds the tape drive. If process Pi request a tape drive and process Pj request a printer then a deadlock occurs. Multithread programs are good candidates for deadlock because they compete for shared resources.

Deadlock Characterization

Necessary Conditions:

A deadlock situation can occur if the following 4 conditions occur simultaneously in a system

1. Mutual Exclusion:

Only one process must hold the resource at a time. If any other process requests for the resource, the requesting process must be delayed until the resource has been released,

2. Hold and Wait:

A process must be holding at least one resource and waiting to acquire additional resources that are currently being held by the other process.

3. No Preemption:

Resources can’t be preempted i.e., only the process holding the resources must release it after the process has completed its task.

4. Circular Wait:

A set {PO.P1........Pn} of waiting process must exist such that PO is waiting for a resource i.e., held by P1, Pl is waiting for a resource ie., held by P2. Pn-l is waiting for resource held by process Pn and Pn is waiting for the resource i.e., held by P1.

All the four conditions must hold for a deadlock to occur.

Resource Allocation Graph:

- Deadlocks are described by using a directed graph called system resource allocation graph. The graph consists of set of vertices (v) and set of edges (e).
- The set of vertices (v) can be described into two different types of nodes

  P={P1,P2,…………. Pn} ie, set consisting of all active processes and

  R={R1,R2...................Rn} i.e., set consisting of all resource types in the system.
- A directed edge from process Pi to resource type Rj denoted by Pi →Ri indicates that Pi requested an instance of resource Rj and is waiting. This edge is called Request edge.
- A directed edge Ri → Pj signifies that resource Rj is held by process Pi. This is called Assignment edge.
- If the graph contain no cycle, then no process in the system is deadlock. If the graph contains a cycle then a deadlock may exist.
- If each resource type has exactly one instance than a cycle implies that a deadlock has occurred. If each resource has several instances then a cycle do not necessarily implies that a deadlock has occurred.

Methods for Handling Deadlocks:

There are three ways to deal with deadlock problem

- **We can use a protocol to prevent deadlocks ensuring that the system will never enter into the deadlock state.**
- **We allow a system to enter into deadlock state, detect it and recover from it.**
- **We ignore the problem and pretend that the deadlock never occur in the system. This is used by most OS including UNIX.**
- To ensure that the deadlock never occur the system can use either deadlock avoidance or a deadlock prevention.
- Deadlock prevention is a set of method for ensuring that at least one of the necessary conditions does not occur.
- Deadlock avoidance requires the OS is given advance information about which resource a process will request and use during its lifetime.
- If a system does not use either deadlock avoidance or deadlock prevention then a deadlock situation may occur. During this it can provide an algorithm that examines the state of the system to determine whether a deadlock has occurred and algorithm to recover from deadlock.
- Undetected deadlock will result in deterioration of the system performance.

Deadlock Prevention:

- For a deadlock to occur each of the four necessary conditions must hold. If at least one of the three conditions does not hold then we can prevent occurrence of deadlock.

1. **Mutual Exclusion:** This holds for non-sharable resources

Eg:- A printer can be used by only one process at a time.

Mutual exclusion is not possible in sharable resources and thus they cannot be involved in deadlock. Read-only files are good examples for sharable resources. A process never waits for accessing a sharable resource. So we cannot prevent deadlock by denying the mutual exclusion condition in non-sharable resources.

2. **Hold and Wait:** This condition can be eliminated by forcing a process to release all its resources held by it when it request a resource i.e., not available

   * One protocol can be used is that each process is allocated with all of its resources before its start execution.

   Eg:- consider a process that copies the data from a tape drive to the disk, sorts the file Eg:- consider a process that copies the data from a tape drive to the disk, sorts the file and then prints the results to a printer. If all the resources are allocated at the beginning then the tape drive, disk files and printer are assigned to the process. The main problem with this is it leads to low resource utilization because it requires printer at the last and is allocated with it from the beginning so that no other process can use it.

   * Another protocol that can be used is to allow a process to request a resource when the process has none. ie., the process is allocated with tape drive and disk file. It performs the required operation and releases both. Then the process once again request for disk file and the printer and the problem and with this is starvation is possible.

3. **No Preemption:**

   To ensure that this condition never occurs the resources must be preempted. The following protocol can be used.

   * If a process is holding some resource and request another resource that cannot be immediately allocated to it, then all the resources currently held by the requesting process are preempted and added to the list of resources for which other processes may be waiting. The process will be restarted only when it regains the old resources and the new resources that it is requesting.

   * When a process request resources, we check whether they are available or not. If they are available we allocate them else we check that whether they are allocated to some other waiting process. If so we preempt the resources from the waiting process and allocate them to the requesting process. The requesting process must wait.

4. **Circular Wait:**

   The fourth and the final condition for deadlock is the circular wait condition. One way to ensure that this condition never, is to impose ordering on all resource types and each process requests resource in an increasing order.

   Let R={R1.R2..........Rn} be the set of resource types. We assign each resource type with a unique integer value. This will allows us to compare two resources and determine whether one precedes the other in ordering.

Eg:-we can define a one to one function

F:R →N as follows :- F(disk drive)=5

F(printer)=12

F(tape drive)=1

Deadlock can be prevented by using the following protocol:-

* Each process can request the resource in increasing order. A process can request any number of instances of resource type say Ri and it can request instances of resource type Rj only F(Rj) > F(Ri).

* Alternatively when a process requests an instance of resource type Rij, it has released any resource Ri such that F(Ri) >= F(Rj).

If these two protocols are used then the circular wait can’t hold.

Deadlock Avoidance:

- ❖ Deadlock prevention algorithm may lead to low device utilization and reduces system throughput.
- ❖ Avoiding deadlocks requires additional information about how resources are to be requested. With the knowledge of the complete sequences of requests and releases we can decide for each requests whether or not the process should wait.
- ❖ For each requests it requires to check the resources currently available. resources that are currently allocated to each processes future requests and release of each process to decide whether the current requests can be satisfied or must wait to avoid future possible deadlock.
- ❖ A deadlock avoidance algorithm dynamically examines the resources allocation state to ensure that a circular wait condition never exists. The resource allocation state is defined by the number of available and allocated resources and the maximum demand of each process.

Safe State:

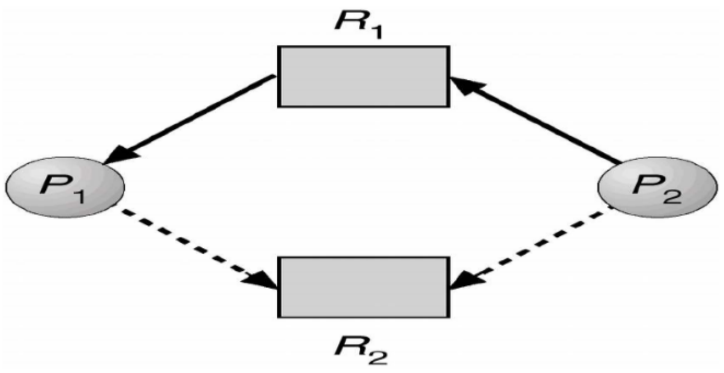
- ❖ A state is a safe state in which there exists at least one order in which all the process will run completely without resulting in a deadlock.
- ❖ A system is in safe state if there exists a safe sequence.
- ❖ A sequence of processes <P1,P2 ...Pn> is a safe sequence for the current allocation state if for each Pi the resources that Pi can request can be satisfied by the currently available resources.
- ❖ If the resources that Pi requests are not currently available then Pi can obtain all of its needed resource to complete its designated task.
- ❖ A safe state is not a deadlock state.
- ❖ Whenever a process request a resource ie., currently available, the system must decide whether resources can be allocated immediately or whether the process must wait. The request is granted only if the allocation leaves the system in safe state.
- ❖ In this, if a process requests a resource ie., currently available it must still have to wait.Thus resource utilization may be lower than it would be without a deadlock avoidance algorithm.

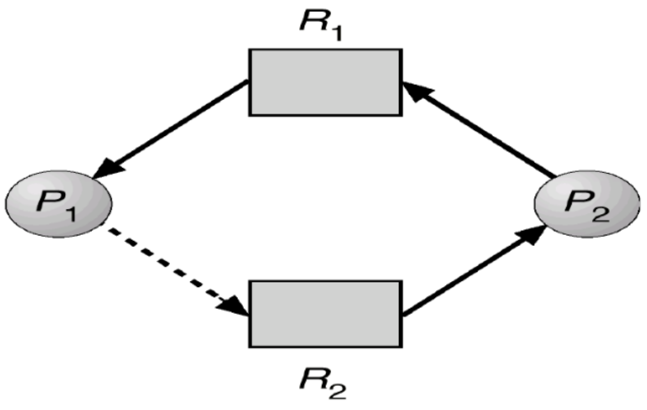
Resource Allocation Graph Algorithm:

- ❖ This algorithm is used only if we have one instance of a resource type.
- ❖ In addition to therequest edge and the assignment edge a new edge called claim edge is used.

For eg:- A claim edge Pi→Rj indicates that process Pi may request Rj in future. Theclaim edge is represented by a dotted line

- When a process Pi requests the resource Rj, the claim edge is converted to arequest edge.
- When resource Rj is released by process Pi, the assignment edge Rj → Pi isreplaced by the claim edge Pi →Rj.
- When a process Pi requests resource Rj the request is granted only if converting the request edge Pi>Rj to as assignment edge Rj>Pi do not result in a cycle. Cycle detection algorithm is used to detect the cycle. If there are no cycles then the allocation of the resource to process leave the system in safe state.





Bankers Algorithm

This algorithm is applicable to the system with multiple instances of each resource types, but this is less efficient then the resource allocation graph algorithm.

When a new process enters the system it must declare the maximum number of resources that it may need. This number may not exceed the total number of resources in the system. The system must determine that whether the allocation of the resources will leave the system in a safe state or not. If it is so resources are allocated else it should wait until the process release enough resources.

Several data structures are used to implement the banker's algorithm. Let 'n' be the number of processes in the system and 'm' be the number of resources types. We need the following data structures:

Available: A vector of length m indicates the number of available resources. If Available[i]=k, then k instances of resource type Rj is available.

Max: Ann*m matrix defines the maximum demand of each process if Max[i,j]=k, then Pi may request at most k instances of resource type Rj.

Allocation: An n*m matrix defines the number of resources of each type currently allocated to each process. If Allocation[i,j]=k. then Pi is currently k instances of resource type Rj

Need: An n*m matrix indicates the remaining resources need of each process. If Need [i,j]=k. then Pi may need k more instances of resource type Rj to compute its task.

So Need[i,j]=Max[i,j]-Allocation[i]

Safety Algorithm

This algorithm is used to find out whether or not a system is in safe state or not.

```
Step 1. Let work and finish be two vectors of length M and N respectively.
            Initialize work = available and
            Finish[i] = false for i=1.2.3........0
Step 2. Find i such that both
            Finish[i] = false
            Need i <= work
            If no such i exist then go to step 4
Step 3.     Work = work + Allocation
            Finish[i]=true
            Go to step 2
Step 4. If finish[i]=true for all i, then the system is in safe state.
```

Resource Request Algorithm:

Let Request(i) be the request vector of process Pi. If Request_i [j] = k. then process Pi wants K instances of the resource type Rj. When a request for resources is made by process Pi the following actions are taken.

1. If Request_i <= Need_i go to step 2 otherwise raise an error condition since the process has exceeded its maximum claim.
2. If Request_i <= Available go to step 3 otherwise Pi must wait. Since the resources are not available.
3. If the system want to allocate the requested resources to process Pi then modify the state as follows

    $\text{Available} = \text{Available} — \text{Request}_i$

    $\text{Allocation}_i= \text{Allocation}_i+ \text{Request}_i$

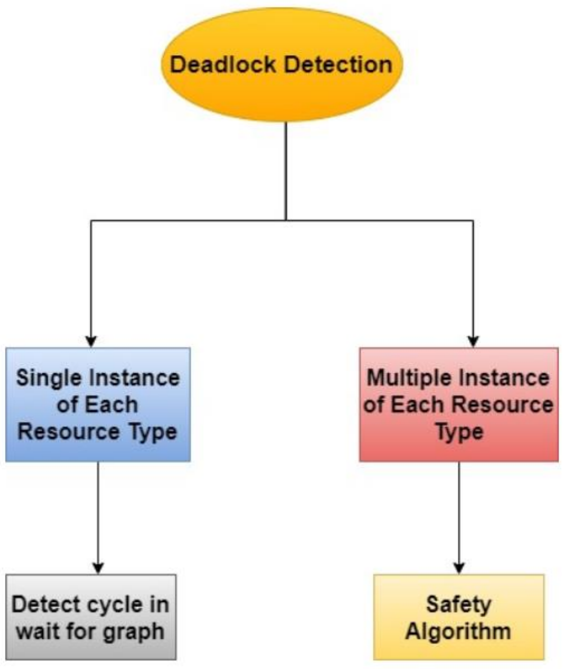
    $\text{Need}_i= \text{Need}_i— \text{Request}_i$

If the resulting resource allocation state is safe, the transaction is complete and Pi is allocated its resources. If the new state is unsafe then Pi must wait for Request(i) and old resource allocation state is restored.

Deadlock Detection:

If a system does not employ either deadlock prevention or a deadlock avoidance algorithm then a deadlock situation may occur. In this environment the system may provide:

- ❖ An algorithm that examines the state of the system to determine whether a deadlock has occurred.
- ❖ An algorithm to recover from the deadlock.

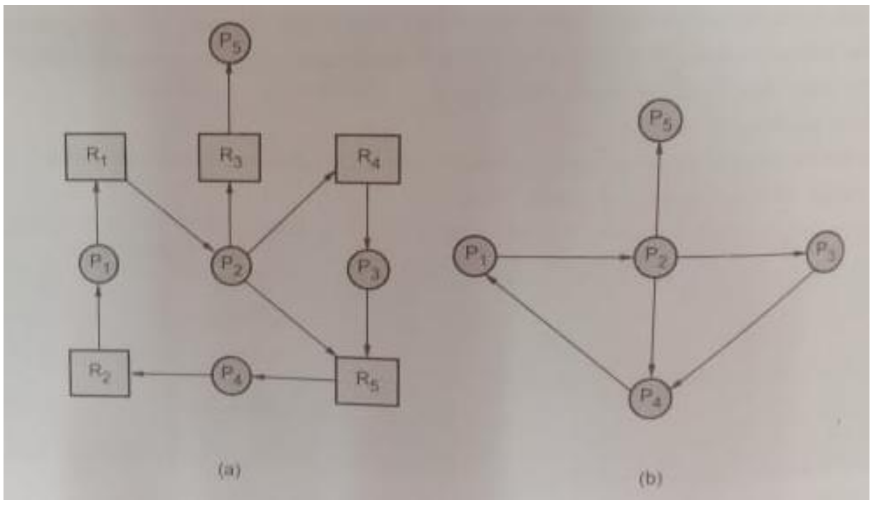

Deadlock Detection
Single Instance of Each Resource Type
Multiple Instance of Each Resource Type
Detect cycle in wait for graph
Safety Algorithm

Single Instances of each Resource Type:

If all the resources have only a single instance, then a deadlock-detection algorithm can be defined that mainly uses the variant of the resource-allocation graph and is known as a wait-for graph. This wait-for graph is obtained from the resource-allocation graph by removing its resource nodes and collapsing its appropriate edges.

- ❖ An edge from Pi to Pj in wait for graph implies that Pi is waiting for Pj to release a resource that Pi needs.
- ❖ An edge from Pi to Pj exists in wait for graph if and only if the corresponding resource allocation graph contains the edges Pi → Rq and Rq → Pj

Deadlock exists with in the system if and only if there is a cycle. In order to detect the deadlock, the system needs to maintain the wait-for graph and periodically system invokes an algorithm that searches for the cycle in the **wait-for** graph.



Resource Allocation Graph with corresponding Wait for Graph

Multiple Instances of Each Resource Type

- The above scheme that is a wait-for graph is not applicable to the resource-allocation system having multiple instances of each resource type. Now we will move towards a deadlock detection algorithm that is is applicable for such systems.
- This algorithm mainly uses several time-varying data structures that are similar to those used in Banker's Algorithm and these are as follows:

1. Available: It is an **array** of length m. It represents the number of available resources of each type.
2. Allocation: It is an n x m matrix which represents the number of resources of each type currently allocated to each process.
3. Request: It is an **n*m** matrix that is used to indicate the request of each process; if Request[i][j] equals to k, then process Pi is requesting k more instances of resource type Ri.

- Allocation and Request are taken as vectors and referred to as Allocation and Request. The Given detection algorithm is simply used to investigate every possible allocation sequence for the processes that remain to be completed.

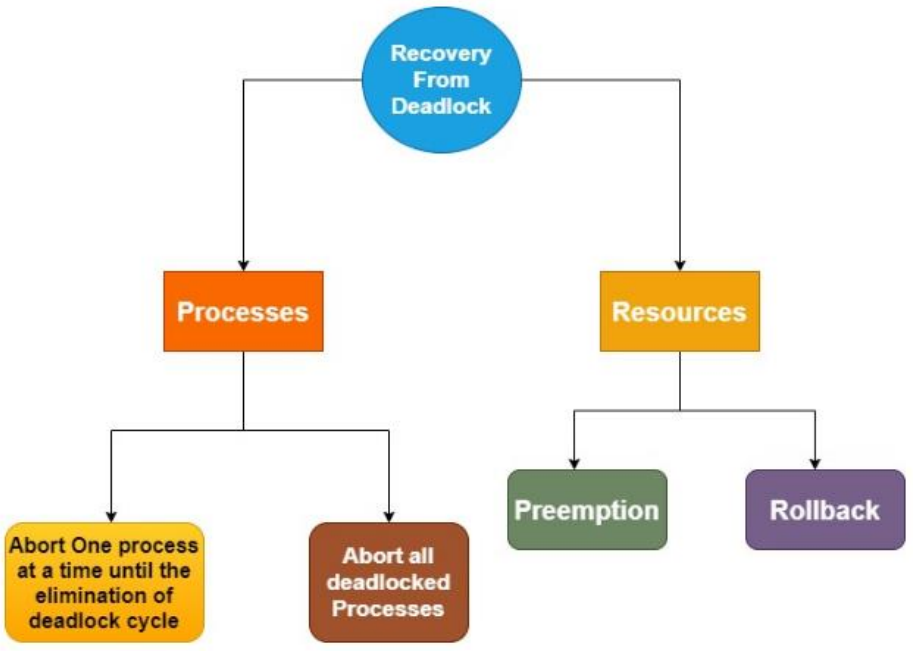
Algorithm:

```
1.  Let Work and Finish be vectors of length m and n, respectively. Initialize:
      Work = Available
      Finish [i] = False if allocation not equal to 0
      Finish [i] = True if allocation = 0
2.  Find an i such that
      Finish [i] = False
      Request <= Work
3.  Work = Work + Allocation
      Finish[i] = true, goto step 2
4.  If Finish[i] == false for some i, 0<=i<n, then it means the system is in a deadlocked state
    otherwise safe state.
```

Recovery from Deadlock

When a detection algorithm determines that a deadlock exists then there are several available alternatives. There one possibility and that is to inform the operator about the deadlock and let him deal with this problem manually.

Another possibility is to let the system recover from the deadlock automatically. These are two options that are mainly used to break the deadlock.


Recovery From Deadlock
Processes
Resources
Abort One process at a time until the elimination of deadlock cycle
Abort all deadlocked Processes
Preemption
Rollback

1. Process Termination

In order to eliminate deadlock by aborting the process, we will use one of two methods given below. In both methods, the system reclaims all resources that are allocated to the terminated processes.

- **Aborting all deadlocked Processes** Clearly, this method is helpful in breaking the cycle of deadlock, but this is an expensive approach. This approach is not suggestible but can be used if the problem becomes very serious. If all the processes are killed then there may occur insufficiency in the system and all processes will execute again from starting.
- **Abort one process at a time until the elimination of the deadlock cycle** This method can be used but we have to decide which process to kill and this method incurs considerable overhead. The process that has done the least amount of work is killed by the Operating system firstly.

- Many factors may affect which process is chosen, including:

a. Least amount of processor time consumed so far.

b. Least amount of output produced so far.

c. Lowest Priority

d. Most estimated time remaining.

e. Least total resources allocated so far.

2. **Resource Preemption**

If preemption is required to deal with deadlocks then three issues need to be addressed:

1. **Selecting a Victim:** Which resources and which processes are to be preempted?
2. **Rollback:** Backup of each deadlocked process to some previously defined check point and restart all processes. This requires rollback and restart mechanisms are built in to the system.
3. **Starvation:** How do we ensure that starvation will not occur?

Methods for Handling Deadlock:

1. Deadlock Prevention
2. Deadlock Avoidance
3. Deadlock Detection
4. Deadlock Recovery