B.C.A. (Sem – IV)

B.C.A. - 404

Operating System

Purushottam Singh

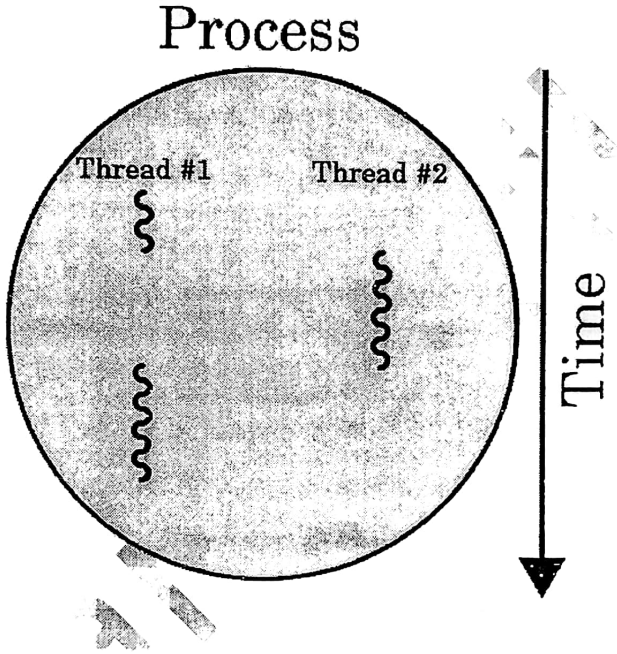
Unit- 3

Unit-3

Thread: -

Introduction of thread: -

- A thread is a single sequence stream within in a process.
- Because threads have some of the properties of processes, they are sometimes called lightweight processes.
- In a process, threads allow multiple executions of streams.
- The CPU switches rapidly back and forth among the threads giving illusion that the threads are running in parallel.
- Like a traditional process i.e., process with one thread, a thread can be in any of several states (Running, Blocked, Ready and Terminated).
- An operating system that has thread facility, the basic unit of CPU utilization is a thread.
- A thread has or consists of a program counter (PC), a register set, and a stack space.
- Threads are not independent of one other like processes as a result threads shares with other threads their code section, data section, OS resources also known as task, such as open files and signals.
- It's important to note that a thread can do anything a process can do.



Process and thread: -

- As we mentioned earlier that in many respect threads operate in the same way as that of processes. Some of the similarities and differences are mentioned as under.
- **Similarities:**
  1. Like processes threads share CPU and only one thread active (running) at a time.
  2. Like processes, threads within processes, threads within a process execute sequentially.
  3. Like processes, thread can create children thread.
  4. And like process, if one thread is blocked, another thread can run.
- **Differences**
  1. Unlike processes, threads are not independent of one another.
  2. Unlike processes, all threads can access every address in the task.
  3. Unlike processes, threads are design to assist one other.
  4. Note that processes might or might not assist one another because processes may originate from different users.

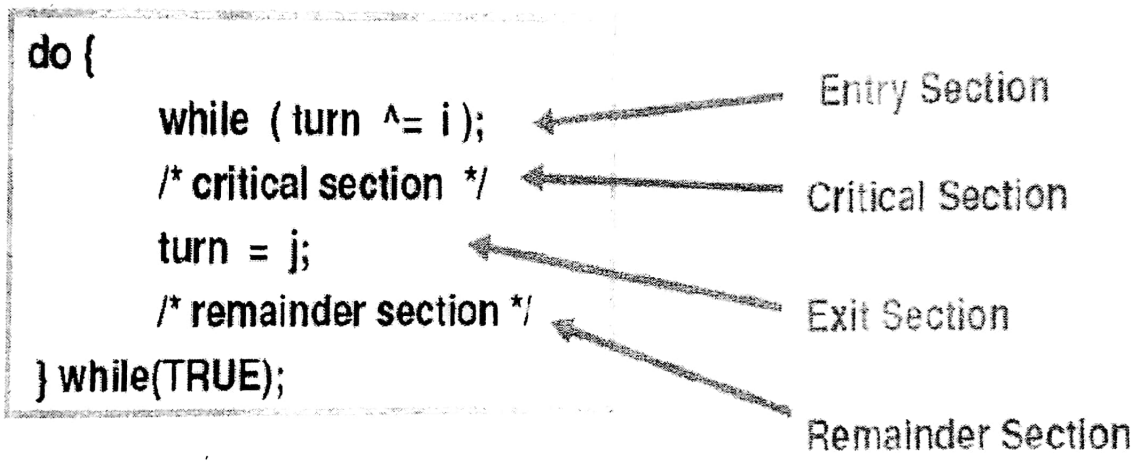
Process / Thread synchronization: -

- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes.
- Process synchronization or coordination seeks to make sure that concurrent processes or threads don't interfere with each other when accessing or changing shared resources.
- Some terms to help frame the problem better:
  1. Segments of code that touch shared resources are called critical sections — thus, no two processes should be in their critical sections at the same time.
  2. The critical-section problem is the problem of designing a protocol for ensuring that cooperating processes' critical sections don't interleave.

Critical section: -

- A critical section is piece of code that accesses a shared resource (data structure or device) that must not be accessed at same time by more then one thread of execution.
- The goal is to provide a mechanism by which only one instance of a critical section is executing for a particular shared resource.
- Unfortunately, it is often very difficult to detect critical section problems.
- A critical section environment contains following sections.
  1. **Entry Section:** An entry section code requesting to entry into the critical section.
  2. **Critical Section:** Critical section code in which only one process can execute at any one time.
  3. **Exit Section:** An exit section is the end of the critical section releasing or allowing others to enter in critical section.
  4. **Remainder Section:** The remainder section is the rest part of the code after the critical section.

Critical Sections



- Solution of critical section problem it must enforce all following three rules.
  1. **Mutual Exclusion (MutEx):** If process P_i is executing in its critical section, then no other process can be executing in their critical sections.
  2. **Progress:** If no process is executing in its critical section and there is some other processes wish to enter their critical section, and then the selection of the processes that will enter critical section next cannot be postponed indefinitely.
  3. **Bounded Waiting:** A bound must exist on the number of times that other processes are allowed to enter their critical section after a process has made a request to enter its critical section.

Semaphores: -

- A semaphore (S) is an integer variable that is manipulated automatically by using two operations known as **P (for wait)** and **V (for signal)**.
- The classical definition for **wait** is as under.

```
wait(S)
{
        while(S<=0)
        ;       // No – Operation
        S-- ;
}
```

(Note: - Block until semaphore (S) > 0, ten subtract 1 from semaphore (S) and proceed.)

- The classical definition for **signal** is as under.

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

(Note: - Add 1 to semaphore (S))

- There are two types of semaphores. [1] Binary semaphore [2] Counting semaphore.
- **Binary semaphore:**
  1. The binary semaphore is also known as Mutual Exclusion (mutex) semaphore.
  2. Binary semaphore gives the guarantees that mutually exclusive access to resource.
  3. Binary semaphore allows only one thread to enter into critical section at a time.
  4. The value of binary semaphore is between 0 and 1.

- **Counting semaphore:**
  1. The counting semaphore represents a resource with many instances.
  2. The counting semaphore is allows thread to enter as long more instance are available.
  3. The counting semaphore is initializes to N where N=number of units which are available.

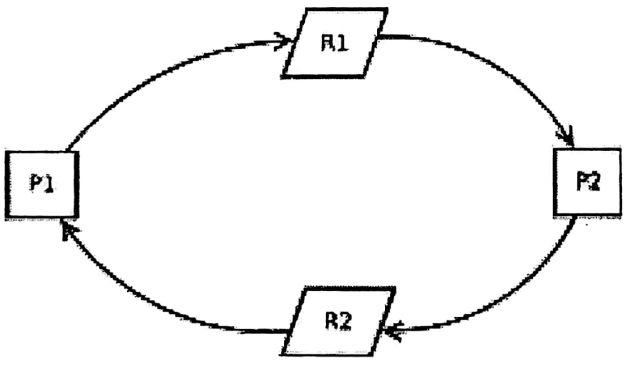
- **Usage:**
  1. We can use the semaphore to deal with critical section problem.
  2. We can also use the semaphore to solve various synchronization problems.
  3. We can use the semaphore to implementation of mutual exclusion.

- **Implementation:**
  1. Each semaphore has an associated queue of thread when **P** (wait) is called by a thread
     - If semaphore was available (>0), decrement semaphore and let thread continue.
     - If semaphore was unavailable (<=0), place thread on associated queue and run some other thread.
  2. When **V** (signal) is called by thread
     - If threads are waiting on the associated queue, unblock one.
     - If no threads are waiting on the associated queue then increment semaphore.

Deadlock: -

Definition of Deadlock and Deadlock Situations: -

- A process is requesting for resources. If the resources are not available at that time, the process entered in wait state. It may happen that waiting processes will never change his waiting state, because that requested resources are held by another waiting processes.
- In other words: "Deadlock - Occurs when resources needed by one process are held by some other waiting process."
- Under normal mode of operation, a process may use a resource in following sequence.
  1. **Request:** If request cannot be granted immediately, then the requesting process must wait until it can acquire the resource.
  2. **Use:** The process can operate on the resource.
  3. **Release:** The process releases that resource.



- In this figure process P1 holding resource R2 and requested for resource R1.
- The process P2 holding the resource R1 which is requested by P1 and P2 requested for resource R2 which is allocated to P1. This situation is called deadlock.

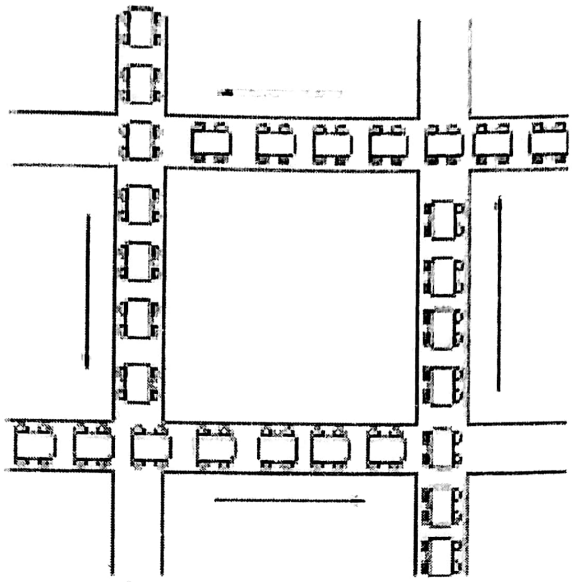


Fig: - Deadlock Situation

Deadlock: -

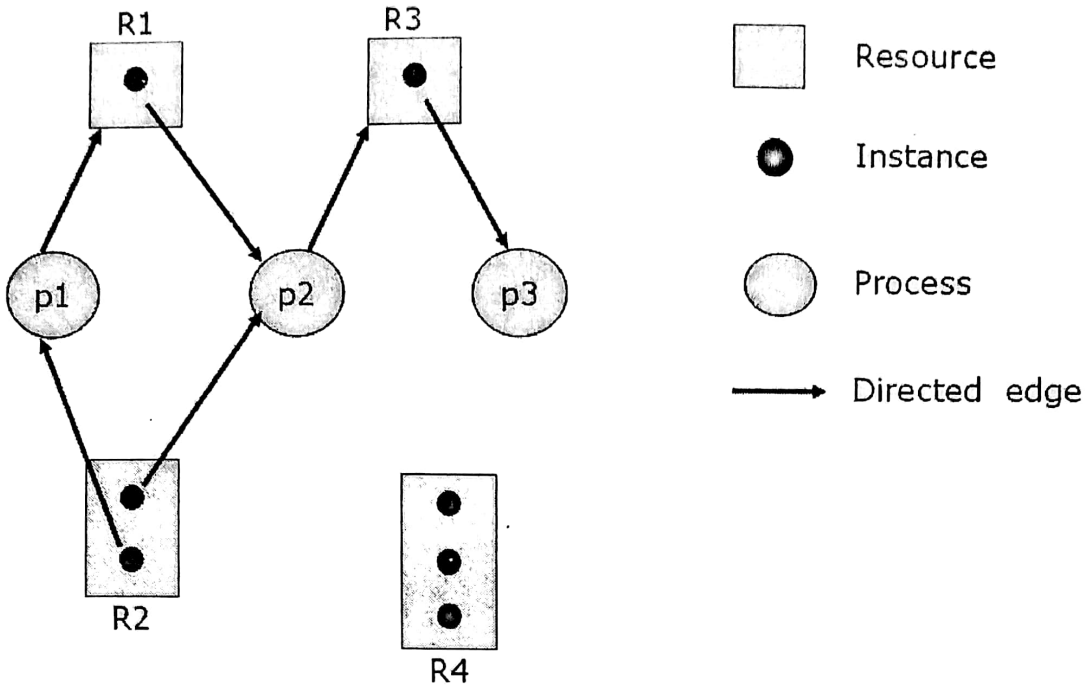
Cause for deadlock (Necessary condition for deadlock): -

- A deadlock can arise in following four situations.
  1. **Mutual exclusion:** A resource that cannot be used by more than one process at a time. If another process is requesting for that resource, the requesting process must be delayed until the resource has been released.
  2. **Hold and wait:** A process that is holding at least one resource and also waiting for another resource which used by another process.
  3. **No preemption:** Resource cannot be preempted; a resource can be released after that process has completed its task.
  4. **Circular wait:** Given a set of processes {P1, P2, P3 ... Pn} P1 has a resource needed by P2, P2 has a resource needed by P3, ..., Pn has a resource needed by P1.
- We emphasize that all four conditions must hold for a deadlock to occur.

Resource allocation graph: -

- Deadlock can be described more simply in term of graphs is known as *system resource-allocation graph*.
- This graph is consists of a set of vertices V and a set of edges E.
- The set of vertices V is partitioned into two different Commands of nodes.
- First is P = {P1, P2, ..., Pn}.
- This set of vertices is consisting of all active processes in system.
- And second is R = {R1, R2, ..., Rm}, this set is consisting of all resources Commands in the system.
- A directed edge from process Pi to resource Command Rj is denoted by Pi → Rj.
- Meaning of Pi → Rj is that Pi process is requested for resource Command Rj and is currently waiting for that resource.
- A directed edge from resource Command Rj to process Pi is denoted by Rj → Pi.
- Meaning of Rj → Pi is that resource Command Rj has been allocated to process Pi.
- A directed edge Pi → Rj is called request edge and directed edge Rj → Pi is called an assignment edge.
- Consider the following situation.
- The set P,R and E:
  - P = {P1, P2, P3}
  - R = {R1, R2, R3, R4}
  - E = {P1 → R1, P2 → R3, R1 → P2, R2 → P2, R2 → P1, R3 → P3}

- Resource instance:
  - One instance of resource Command R1
  - Two instance of resource Command R2
  - One instance of resource Command R3
  - Three instance of resource Command R4



Resource allocation graph

- **Process states:**
  - Process p1 is holding an instance of resource Command R2, and is waiting for an instance of resource Command R1.
  - Process p2 is holding an instance of R1 and R2, and is waiting for an instance of resource Command R3.
  - Process p3 is holding an instance of R3.
- In this example there is sufficient condition for deadlock.

Method for handling deadlock: -

Deadlock Prevention: -

- **Mutual exclusion: -**
  1. The mutual-exclusion condition must hold for non sharable resources.
  2. For example – a printer can not be simultaneously shared by several processes.
  3. Sharable resources do not require mutually exclusive access and thus cannot be involved in a deadlock.
  4. Read-only files are a good example of a sharable resource. If several resource processes attempt to open a read-only files at the same time, they can be granted simultaneous access to the files. A process never needs to wait for a sharable resource.
- **Hold and wait: -**
  1. To ensure that the hold and wait condition never occurs in the system, we must guarantee that, whenever a process requests a resource, it does not hold any other resource.
  2. One rule is that one process requires the resource then allocate that resource to the process before it begins its execution.
  3. An alternative rules is that a process may request some resources and use them and before it can request any other resources, it must release all the resources that it is currently used.
- **No preemption: -**
  1. The third necessary condition is that there is no preemption of resource that have already allocated to the process. To ensure that this condition does not hold, we can use following rules.
  2. If a process is holding some resources and request for another resource that cannot be immediately allocated to the process (The process must wait), then all resources currently held by that process that are preempted.

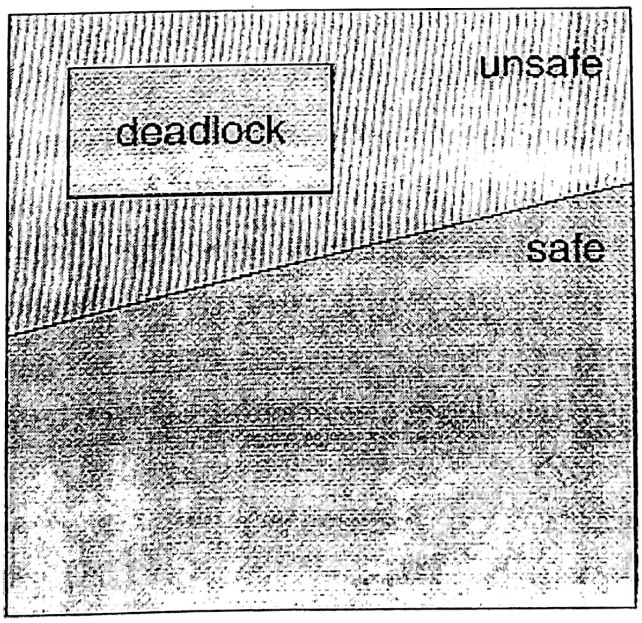
3. If a process requests some resources, we first check whether they are available. If they are available, we can allocate them to the process.
4. But if they are not available, we check whether they are allocated to some other process that is waiting for some additional resources. If so, we preempt the desired resources from the waiting process and allocate them to the requesting process. If the resources are not either available or held by a waiting process, the requesting process must wait.

- **Circular wait: -**
  1. Circular wait prevention consists of allowing processes to wait for resources, but ensure that the waiting can't be circular.
  2. One approach might be to assign number to each resource and force processes to request resources in order of increasing number.
  3. That is to say that if a process holds some resources and the highest number of these resources is m, and then this process cannot request any resource with number that is smaller than m.
  4. This forces resource allocation to follow a particular and non-circular ordering, so circular wait cannot occur.
  5. Another approach is to allow holding only one resource per process; if a process requests another resource, it must first free the one it's currently holding (or hold-and-wait).

Deadlock avoidance: -

- This technique is use to avoid deadlock.
- Deadlock avoidance is arranging that how the resources are granted at run time.
- Simplest and most useful model requires that each process declare the maximum number of resources of each type that it may need.
- The deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that there can never be a circular-wait condition.
- Resource-allocation state is defined by the number of available and allocated resources, and the maximum demands of the processes.
- The most common deadlock avoidance algorithm is **Dijkstra's Banker's Algorithm**.

- **Safe State**
  1. A state is safe if the system can allocate all resources requested by all processes (up to their stated maximums) without entering a deadlock state.
  2. More formally, a state is safe if there exists a safe sequence of processes { P0, P1, P2, ..., PN } such that all of the resource requests for Pi can be granted using the resources currently allocated to Pi and all processes Pj where j < i. (I.e. if all the processes prior to Pi finish and free up their resources, then Pi will be able to finish also, using the resources that they have freed up.)
  3. If a safe sequence does not exist, then the system is in an unsafe state, which MAY lead to deadlock. (All safe states are deadlock free, but not all unsafe states lead to deadlocks.)



- **Banker's Algorithm**: -

1. The algorithm requires some information like how many resources is needed by processes.
2. Then the Operating System (OS) behaves like a banker.
3. It only allocates the resources to process if it has enough available resources.
4. The Banker's Algorithms can be mentioned as under.

1. Let Work and Finish be vectors of length m and n, respectively. Initialize:
   Work = Available
   Finish [i] = false for i - 1,3, ..., n.

2. Find and i such that both:
   (a) Finish [i] = false
   (b) Needi ≤ Work
   If no such i exists,
   go to step 4.

3. Work = Work + Allocationi
   Finish[i] = true
   go to step 2.

4. If Finish [i] == true for all i, then the systen is in a safe state.

5. We characterise this as a safe state or unsafe state from following example.
6. **Safe state** - Consider that OS has 10 available resources for processes.

Process	Allocated	Maximum
P0	1	6
P1	1	5
P2	2	4
P3	4	7

Available = 2

7. **Unsafe state** – Consider that OS has 10 available resources for processes.

Process	Allocated	Maximum
P0	1	6
P1	2	5
P2	2	4
P3	4	7

Available = 1

8. An unsafe state does not mean that the system is deadlocked, but it shows the possibility that system can entered into deadlock situation.

Deadlock detection: -

- If deadlocks are not avoided, then another approach is to detect when they have occurred and recover from deadlock situation.
- In addition to the performance hit of constantly checking for deadlocks, a policy / algorithm must be in place for recovering from deadlocks, and there is potential for lost work when processes must be aborted or have their resources preempted.

- **Single instance of each resource: -**

1. If each resource category has a single instance, then we can use a variation of the resource-allocation graph known as a wait-for graph.
2. A wait-for graph can be constructed from a resource-allocation graph by eliminating the resources and collapsing the associated edges, as shown in the figure below.
3. An arc from **Pi** to **Pj** in a wait-for graph indicates that process **Pi** is waiting for a resource that process **Pj** is currently holding.
4. Consider the following two figures.

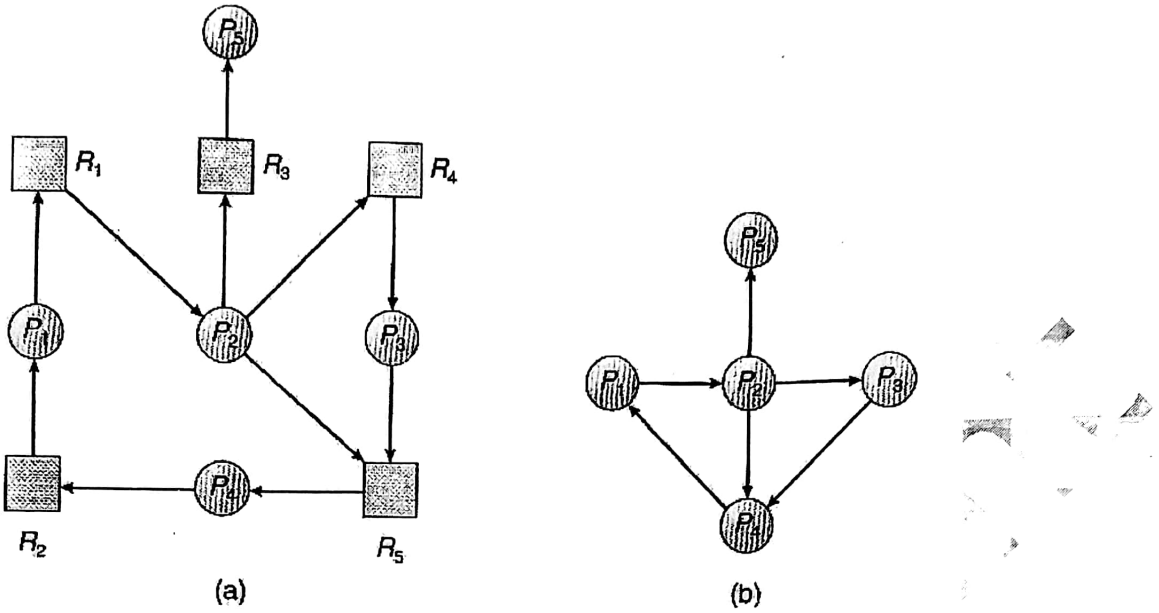


Figure (a) Resource-allocation graph. (b) Corresponding wait-for graph.

5. As before, cycles in the wait-for graph indicate deadlocks.
6. This algorithm must maintain the wait-for graph, and periodically search it for cycies.

- **Several instance of resource type: -**
  1. The wait-for graph scheme is not applicable to a resource allocation system with multiple instance of each resource type.
  2. For several instance of resource type following approach is applicable.
  3. **Available:** A vector of length m indicates the number of available resources of each type.
  4. **Allocation:** An n x m matrix defines the number of resources of each type currently allocated to each process.
  5. **Request:** An **n x m** matrix indicates the current request of each process. If Request **[ij] = k**, then process **Pi** is requesting **k** more instances of resource type **Rj**.
  6. Consider the following algorithm.

1. Let Work and Finish be vectors of length m and n, respectively Initialize:
   (a) Work = Available
   (b) For i = 1,2, ..., n, if Allocation i ≠ 0, then
   Finish[i] = false;
   otherwise Finish[i] = true.

2. Find an index i such that both:
   (a) Finish[i] == false
   (b) Requesti ≤ Work
   If no such i exists,
   go to step 4.

3. Work = Work + Allocationi
   Finish[i] = true
   go to step 2.

4. If Finish[i] == false, for some i, 1 ≤ i ≤ n, then the system is in deadlock state. Moreover, if Finish[i] == false, then Pi is deadlocked.

Deadlock recovery: -

- There are two basic approaches to recovery from deadlock: [1] Terminate one or more processes involved in the deadlock [2] Preempt resources.
- **Process Termination: -**
  1. Terminate all processes involved in the deadlock. This definitely solves the deadlock, but at the expense of terminating more processes than would be absolutely necessary.

2. Terminate processes one by one until the deadlock is broken. This is more conservative, but requires doing deadlock detection after each step.
3. In the other case there are many factors that can go into deciding which processes to terminate next:
    - Process priorities.
    - How long the process has been running, and how close it is to finishing.
    - How many and what type of resources is the process holding. (Are they easy to preempt and restore?)
    - How many more resources does the process need to complete.
    - How many processes will need to be terminated
    - Whether the process is interactive or batch.

- **Resource Preemption: -**
    1. **Selecting a victim** - Deciding which resources to preempt from which processes involves many of the same decision criteria outlined above.
    2. **Rollback** - Ideally one would like to roll back a preempted process to a safe state prior to the point at which that resource was originally allocated to the process. Unfortunately it can be difficult or impossible to determine what such a safe state is, and so the only safe rollback is to roll back all the way back to the beginning. (I.e. abort the process and make it start over.)
    3. **Starvation –** We can use a priority system, and increase the priority of a process every time its resources get preempted. Eventually it should get a high enough priority that it won't get preempted any more.