

UNIT-3

SYLLABUS:

Process Synchronization: The Critical-Section Problem, Peterson's Solution, Synchronization Hardware, Semaphores, Classic Problems of Synchronization, Monitors.

Deadlocks: System Model, Deadlock Characterization, Deadlock Prevention, Deadlock Avoidance, Deadlock Detection, Recovery from Deadlock.

Process Synchronization:

Process Synchronization:

- There are several situations, where different processes need to interact with each other to achieve a common goal.
- The interaction can be done by sharing data or by sending messages.
- When the data is shared by several independent processes, then there is a chance of data becoming data inconsistent.
- Concurrent access to shared data may result in data inconsistency.
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes.

Example:

Let us consider two processes P1 and P2, both share a variable named 'counter'.

If both P1 and P2 execute simultaneously, with P1 incrementing the counter while P2 is decrementing it. Then, at the end the result of this variable will be such that, it is not expected by either P1 or P2 because it became inconsistent. This is often called as race condition.

Hence a synchronization mechanism is needed to avoid inconsistency among several processes.

Basically, process synchronization is implemented by making a process to wait for another process performs an appropriate action on shared data. It is also called as signaling where one process waits for notification of an event that all occur in another process.

Race Condition: A race condition refers to the situation that results when many processes or threads reads and writes data items in a way that the final result generated is in accordance with the order of instructions execution in multiple processes.

- When several processes access and manipulate the same data concurrently and the outcome of the execution depends on the particular order in which the access takes place is called **race condition**.
- To guard against the race condition, we need to ensure that only one process at a time can be manipulating the variable counter. Hence processes must be synchronized.

Example:

Let P1 and P2 be two processes that shares a global variable 'x'. During execution, if at some point process P1 updates some value of 'x' to 5 and at some other point updates it to 10. Thus, a race among two processes starts for changing the value of 'x' and a process that performs an update operation last determines the final value of 'x'.

Consider another situation in which two processes P3 and P4 shares two global variables 'y' and 'z' whose initial values are 5 and 10 respectively. During some point in execution,

if process P3 executes an assignment statement $y = y + z$ and

if process P4 executes an assignment statement $z = y + z$

The final values of y and z depends on the order in which the two assignment statements are executed if process P3 is executed prior to P4, then the final values of y and z are 15 and 25 respectively, on the other hand if execution of P4 precedes P3, then the result values of y and z are 20 and 15 respectively.

Process Competition for Resources: Conflict arises among the various concurrently executing processes when they are competing for the same resource.

Consider a situation in which two or more processes want to access a resource. Each of these concurrently executing processes is unaware of the presence of other processes and the execution of one process does not cause any effect on the execution of the other process. Hence, the state of the resources used, remains unaffected.

For instance, consider that information is not exchanged between these processes and the execution of one process causes a significant effect on the behavior of the other competing processes, i.e., if two processes want to access a single resource then the OS grants the resource access to only one process and let the other process to wait, as a result of which the blocked process may never get the resource and terminates in an inappropriate manner.

Three problems dominate in case of competing process.

- 1) The need for mutual exclusion.
- 2) The occurrence of deadlock.
- 3) The problems of starvation.

1) The need for Mutual Exclusion: Consider a situation in which two or more processes need access to a single non-sharable resource (Example printer). During the execution process, each process sends the commands to I/O devices or sends and receives data or receives status information etc. Such an I/O device is said to be critical resource and a portion of a program that uses is called a **critical section**. An important point to be considered is that there is only one program is permitted to enter into the critical section at any time.

2) The occurrence of deadlocks: The major cause for the occurrence of a deadlock is the imposition of the mutual exclusion.

Consider an example, granting of two resources R1 and R2 to two processes P1 and P2. Further suppose that each of these processes want to access both the resources in order to execute some function. A situation may occur in which an operating system assigns the resource R1 to process P2 and resource R2 to process P1. Hence, each process is waiting for one of the resources and will not release the acquired resource till it gets the other resource that is a process needs both these resources in order to proceed. This leads to deadlock.

3) The problem of Starvation: Let P1, P2 and P3 be the three processes, each of which requires a periodical access to resource R. If access to resource 'R' is granted to the process P1, then the other two processes P2 and P3 are delayed as they are waiting for the resource 'R'. Now let the access is granted to P3 and if P1 again needs 'R' prior to the completion of its critical section. If the OS permits P1 to use 'R' after P3 has completed its execution, then these ultimate access permissions provided to P1 and P3 causes P2 to be blocked.

This competition among the processes can be controlled by involving an OS which is responsible for allocating the resources to all processes in the system.

The Critical-Section Problem:

The resource that cannot be shared between two or more processes at the same time is called as a **critical resource**.

There may be a situation where more than one process requires to access the critical resource. Then, during the execution of these processes they can send data to the critical resource, receive data from the critical resource or they can just get the information about the status of the critical resource by sending related commands to it. An example of a source by sending related commands to it. An

example of a critical or a non-sharable resource is “printer”. A critical resource can be accessed only from the critical section of a program.

Critical section

A critical section is a segment of code present in a process in which the process may be modifying or accessing common variables or shared data items. The most important thing that a system should control is that, when one process is executing its critical section, it should not allow other processes to execute in its critical sections.

Before executing critical section the process should get permission to enter its critical section from the system. This is called an entry section. After that process executes its critical section and comes out of it this is called exit section. Then, it executes the remaining code called remainder section.

Starvation

Two or more processes are said to be in starvation, if they are waiting perpetually for a resource which is occupied by another process. The process that has occupied the resource may or may not be present in the list of processes that are starved.

Let P1, P2 and P3 be the three processes, each of which requires a periodical access to resource R. If access to resource ‘R’ is granted to the process P1, then the other two processes P2 and P3 are delayed as they are waiting for the resource ‘R’. Now, let the access be granted to P3 and if P1 again needs ‘R’ prior to the completion of its critical section. If the OS permits P1 to use ‘R’ after P3 has completed its execution, then these alternate access permissions provided to P1 and P3 causes P2 to be blocked.

Here, we need to illustrate where starvation is possible or not in algorithms like FCFS, SPN, SRT and priority. Consider FCFS (First Come First Served) Algorithm, in this starvation is not possible. The reason is the CPU picks the process according to arrival of its burst time and runs the process till its completion.

Consider SPN (Shortest Processing Next) Algorithm, in this starvation is possible with the process that has long burst time. The reason is the CPU picks the process that has shortest next burst time. Here, we can overcome starvation problem by using primitive SPN algorithm, which prompts the currently running process.

Next SRT (shortest remaining time) algorithm, in this starvation is possible with the processes that have shortest remaining time. The reason is CPU picks the process that has shortest remaining time. Here, we can overcome the problem of starvation by giving chance to processes that are waiting for a long period of time. Finally, consider priority algorithm, in this starvation is possible with low priority processes. The reason is, CPU picks the process with highest priority.

We can overcome starvation problem by a technique called aging. This technique increases the priority of the processes that are waiting for a long period of time.

Requirements for Mutual Exclusion?

Mutual Exclusion must meet the following requirements.

- 1) Mutual exclusion must be enforced only one process at a time is allowed into its critical section, among all processes that have critical sections for the same resource or the shared object.
- 2) A process that halts in its non-critical section must do so without interfering with other processes.
- 3) It must not be possible for a process requiring access to a critical section to be delayed indefinitely, no deadlock or starvation.
- 4) When no process is in a critical section, any process that requests entry to its critical section must be permitted to enter without delay.
- 5) No assumptions are made about relative process speeds or number of processors.
- 6) A process remains inside its critical section for a finite time only.

Critical section problem:

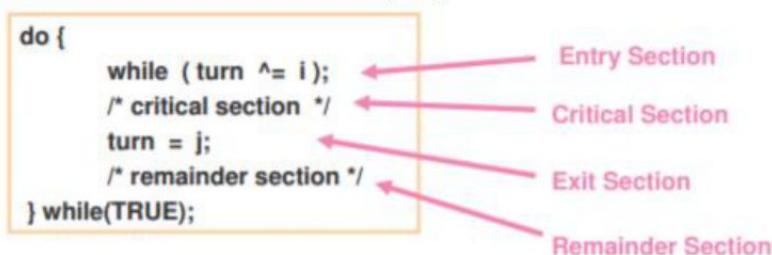
- Each process has a segment of code called a **critical section**. Critical section is used to avoid race conditions on data items.
- In critical section, the process may be changing common variables, updating a table, writing a file and so on.
- At any moment at **most one process can execute** in critical section.
- A **critical section** is a piece of code that accesses a shared resource (data structure or a device) that must not be concurrently accessed by more than one thread of execution.
- The critical-section problem is to design a protocol that the processes can use to cooperate. A Critical Section Environment contains:
 1. **Entry Section:** Code requesting entry into the critical section.
 2. **Critical Section:** Code in which only one process can execute at any one time.
 3. **Exit Section:** The end of the critical section, releasing or allowing others in.
 4. **Remainder Section:** Rest of the code after the critical section.

```

do {
    entry section
    critical section
    exit section
    remainder section
} while (TRUE);

```

General Structure of a Typical Process



A solution to the critical-section problem must satisfy the following requirements:

- 1) **Mutual Exclusion:** If process P_i is executing in its critical section, then no other processes can be executing in their critical sections.

- 2) **Progress:** If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then only those processes that are not executing in their remainder section can participate. In the decision on which will enter its critical next. This selection of the processes that will enter the critical section next cannot be postponed indefinitely.
- 3) **Bounded Waiting:** When a process requests access to a critical section, a decision that grants its access may not be delayed indefinitely. A process may not be denied access because of starvation or deadlock. A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted. (i.e. All requesters must eventually be let into the critical section).

There are two general ways for handling critical sections in the operating systems. They are:

- 1) **Preemptive Kernel:** It allows the Kernel mode process to be preempted (i.e., interrupted) during execution.
- 2) **Non-preemptive Kernel:** It does not allow a Kernel mode process to be preempted during execution, the process will execute until it exits Kernel mode or voluntarily leaves control of the CPU. This approach is helpful in avoiding race conditions.

Peterson's Solution:

- A classic software based solution to the critical section problem is known as Peterson's solution.
- It provides a good algorithmic description of solving the critical section problem and illustrates some of the complexities involved in designing software that addresses the requirements of mutual exclusion, progress and bounded waiting requirements.
- Peterson's solution is restricted to two processes that alternate execution between their critical sections and remainder sections.

Peterson's solution requires two data items to be shared between the two processes:

The variable turn indicates whose turn it is to enter its critical section.

The flag array is used to indicate if a process is ready to enter its critical section, flag[i] = true indicates that process P_i is ready.

```
do {
    flag[i] = TRUE;
    turn = j;
    while (flag[j] && turn == j);

    critical section

    flag[i] = FALSE;

    remainder section

} while (TRUE);
```

The structure of process P_i in Peterson's solution.

The algorithm does satisfy the three essential criteria to solve the critical section problem. For two processes P₀ and P₁:

- 1) **Mutual exclusion:** P₀ and P₁ can never be in the critical section at the same time: If P₀ is in its critical section, then **flag [0] is true** and either **flag [1] is false** (meaning P₁ has left its critical section) or turn is 0 (meaning P₁ is just now trying to enter the critical section, but graciously waiting). In both cases, P₁ cannot be in critical section when P₀ is in critical section.
- 2) **Progress:** Each process can only be blocked at the while if the other process wants to use the critical section (flag[j] = true), AND it is the other process's turn to use the critical section (turn == j). If both of those conditions are true, then the other process (j) will be allowed to enter the critical section, and upon exiting the critical section, will set flag[j] to false, releasing process i. The shared variable turn assures that only one process at a time can be blocked, and the flag variable allows one process to release the other when exiting their critical section.
- 3) **Bounded waiting:** As each process enters their entry section, they set the turn variable to be the other processes turn. Since no process ever sets it back to their own turn, this ensures that each process will have to let the other process go first at most one time before it becomes their turn again.

Synchronization Hardware:

- One simple solution to the critical section problem is to simply prevent a process from being interrupted while in their critical section, which is the approach taken by non-preemptive kernels.
- Unfortunately this does not work well in multiprocessor environments, due to the difficulties in disabling and the re-enabling interrupts on all processors.
- There is also a question as to how this approach affects timing if the clock interrupt is disabled.
- Another approach is for hardware to provide certain *atomic* operations.
- These operations are guaranteed to operate as a single instruction, without interruption.
- One such operation is the "Test and Set", which simultaneously sets a boolean lock variable and returns its previous value as shown below:

```
boolean TestAndSet(boolean *target) {
    boolean rv = *target;
    *target = TRUE;
    return rv;
}
```

The definition of the TestAndSet() instruction.

If the machine supports the TestAndSet instruction, then we can implement mutual exclusion by declaring a Boolean variable lock, initialized to false. The structure of process P_i is shown:

```
do {
    while (TestAndSetLock(&lock))
        ; // do nothing
    // critical section
    lock = FALSE;
    // remainder section
} while (TRUE);
```

Mutual-exclusion implementation with TestAndSet()

The **Swap** instruction, defined as shown below operates on the contents of two words; like the TestAndSet instruction, it is executed atomically.

```
void Swap(boolean *a, boolean *b) {
    boolean temp = *a;
    *a = *b;
    *b = temp;
}
```

The definition of the Swap() instruction.

If the machine supports the Swap instruction, then mutual exclusion can be provided as follows.

A global Boolean variable lock is declared and is initialized to false.

In addition, each process also has a local Boolean variable key.

The structure of process P₁ is shown below:

```
do {
    key = TRUE;
    while (key == TRUE)
        Swap(&lock, &key);

    // critical section

    lock = FALSE;

    // remainder section
}while (TRUE);
```

Mutual-exclusion implementation with the Swap() instruction.

But these algorithms do not satisfy the bounded – waiting requirement. The below algorithm satisfies all the critical section problems. Common data structures used in this algorithm are:

```
Boolean waiting[n];
Boolean lock;
```

Both these data structures are initialized to false.

For proving that the mutual exclusion requirement is met, we must make sure that process P_i can enter its critical section only if either waiting[i] == false or key == false.

The value of key can become false only if the TestAndSet() is executed.

```
do {
    waiting[i] = TRUE;
    key = TRUE;
    while (waiting[i] && key)
        key = TestAndSet(&lock);
    waiting[i] = FALSE;

    // critical section

    j = (i + 1) % n;
    while ((j != i) && !waiting[j])
        j = (j + 1) % n;

    if (j == i)
        lock = FALSE;
    else
        waiting[j] = FALSE;

    // remainder section
}while (TRUE);
```

Bounded-waiting mutual exclusion with TestAndSet()

Semaphores:

- The various hardware based solutions can be difficult for application programmers to implement.
- **Semaphores** are most often used to synchronize operations (to avoid race conditions) when multiple processes access a common, non-shareable resource.
- *Semaphores* are integer variables for which only two (atomic) operations are defined, the **wait (P)** and **signal** operations, whose definitions in pseudocode are shown in the following figure.

Wait:

```
wait(S) {
    while S <= 0
        ; // no-op
    S--;
}
```

Signal:

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

P(S) or S.wait(): decrement or block if already 0

V(S) or S.signal(): increment and wake up process if any

- To indicate a process has gained access to the resource, the process decrements the semaphore.
- Modifications to the integer value of the semaphore in the wait and signal operations must be executed indivisibly.
- When one process modifies the semaphore value, no other process can simultaneously modify that same semaphore value.
- Access to the semaphore is provided by a series of semaphore system calls.
- Semaphores can be used to deal with the *n-processes* critical section problem, where the *n-processes* share a semaphore **mutex** (mutual exclusion) initialized to 1. Each process P_i is organized as shown:


```

do {
    waiting(mutex);
    // critical section
    signal(mutex);
    // remainder section
}while (TRUE);

```

Mutual-exclusion implementation with semaphores.

Implementation

- The big problem with semaphores described above is the busy loop in the wait call (**busy waiting**), which consumes CPU cycles without doing any useful work.
- This type of lock is known as a **spinlock**, because the lock just sits there and spins while it waits.
- While this is generally a bad thing, it does have the advantage of not invoking context switches, and so it is sometimes used in multi-processing systems when the wait time is expected to be short - One thread spins on one processor while another completes their critical section on another processor.
- An alternative approach is to block a process when it is forced to wait for an available semaphore, and swap it out of the CPU.
- In this implementation each semaphore needs to maintain a list of processes that are blocked waiting for it, so that one of the processes can be woken up and swapped back in when the semaphore becomes available. (Whether it gets swapped back into the CPU immediately or whether it needs to hang out in the ready queue for a while is a scheduling problem.).
- The new definition of a semaphore and the corresponding wait and signal operations are shown as follows:

Semaphore Structure:

```

typedef struct {
    int value;
    struct process *list;
} semaphore;

```

Wait Operation:

```

wait(semaphore *S) {
    S->value--;
    if (S->value < 0) {
        add this process to S->list;
        block();
    }
}

```

Signal Operation:

```

signal(semaphore *S) {
    S->value++;
    if (S->value <= 0) {
        remove a process P from S->list;
        wakeup(P);
    }
}

```

- OS's distinguish between **counting and binary semaphores**.
- The value of a **counting semaphore** can range over an unrestricted domain.
- The value of a **binary semaphore** can range only between 0 and 1.
- A binary semaphore must be initialized with 1 or 0, and the completion of P and V operations must alternate.
- If the semaphore is initialized with 1, then the first completed operation must be P.
- If the semaphore is initialized with 0, then the first completed operation must be V.
- Both P and V operations can be blocked, if they are attempted in a consecutive manner.
- **Binary semaphores** are known as **mutex locks** as they are locks that provide mutual exclusion. Binary semaphores are used to deal with the critical section problem for multiple processes.
- **Counting semaphores** can be used to control access to a given resource consisting of a finite number of instances. Counting semaphores maintain a count of the number of times a resource is given.
- The semaphore is initialized to the number of resources available.
- Each process that wishes to use a resource performs a wait() operation on the semaphore.
- When a process releases a resource, it performs a signal() operation.

Deadlocks and Starvation:

The waiting queue may result in a situation where two or more processes are waiting indefinitely that can be caused only by one of the waiting processes. When such a state is reached:

<p>P_0</p> <pre> wait(S); wait(Q); . . . signal(S); signal(Q); </pre>	<p>P_1</p> <pre> wait(Q); wait(S); . . . signal(Q); signal(S); </pre>
--	--

<p>P_0</p> <pre> wait(S); wait(Q); . . . signal(S); signal(Q); </pre>	<p>P_1</p> <pre> wait(Q); wait(S); . . . signal(Q); signal(S); </pre>
--	--

Another problem related to deadlocks is indefinite blocking or starvation, a situation in which processes wait indefinitely within the semaphore. Indefinite blocking may occur if we add and remove processes from the list associated with a semaphore in LIFO order

Drawbacks of Semaphore:

- 1) They are essentially shared global variables.
- 2) Access to semaphores can come from anywhere in a program.
- 3) There is no control or guarantee of proper usage.
- 4) They serve two purposes, mutual exclusion and scheduling constraints.

Classic Problems of Synchronization:

The Bounded Buffer problem (or) Producer-Consumer Problem:

Here the pool consists of n buffers, each capable of holding one item. The mutex semaphore provides mutual exclusion for accesses to the buffer pool and is initialized to the value 1. The empty and full semaphores count the number of empty and full buffers. The

semaphore empty is initialized to the value n, the semaphore full is initialized to value 0. The code below can be interpreted as the producer producing full buffers for the consumer or as the consumer producing empty buffers for the producer.

<pre>do { . . . // produce an item in nextp . . . wait(empty); wait(mutex); . . . // add nextp to buffer . . . signal(mutex); signal(full); }while (TRUE);</pre> <p>The structure of the producer process.</p>	<pre>do { wait(full); wait(mutex); . . . // remove an item from buffer to nextc . . . signal(mutex); signal(empty); . . . // consume the item in nextc . . . }while (TRUE);</pre> <p>The structure of the consumer process.</p>
--	---

The Readers-Writers Problem:

In the readers-writers problem there are some processes (termed readers), who only read the shared data, and never change it, and there are other processes (termed writers), who may change the data in addition to or instead of reading it. There is no limit to how many readers can access the data simultaneously, but when a writer accesses the data, it needs exclusive access. This synchronization problem is referred to as the **readers-writers problem**. There are several variations to the readers-writers problem, most centered around relative priorities of readers versus writers.

- The *first* readers-writers problem gives priority to readers. In this problem, if a reader wants access to the data, and there is not already a writer accessing it, then access is granted to the reader. A solution to this problem can lead to starvation of the writers, as there could always be more readers coming along to access the data.
- The *second* readers-writers problem gives priority to the writers. In this problem, when a writer wants access to the data it jumps to the head of the queue - All waiting readers are blocked, and the writer gets access to the data as soon as it becomes available. In this solution the readers may be starved by a steady stream of writers.

The following code is an example of the first readers-writers problem, and involves an important counter and two binary semaphores:

<pre>do { wait(wrt); . . . // writing is performed . . . signal(wrt); }while (TRUE);</pre> <p>The structure of a writer process.</p>	<pre>do { wait(mutex); readcount++; if (readcount == 1) wait(wrt); signal(mutex); . . . // reading is performed . . . wait(mutex); readcount--; if (readcount == 0) signal(wrt); signal(mutex); }while (TRUE);</pre> <p>The structure of a reader process.</p>
--	--

- *readcount* is used by the reader processes, to count the number of readers currently accessing the data.
- *mutex* is a semaphore used only by the readers for controlled access to readcount.
- *wrt* is a semaphore used to block and release the writers. The first reader to access the data will set this lock and the last reader to exit will release it; The remaining readers do not touch wrt.
- Note that the first reader to come along will block on wrt if there is currently a writer accessing the data, and that all following readers will only block on mutex for their turn to increment readcount

Some hardware implementations provide specific reader-writer locks, which are accessed using an argument specifying whether access is requested for reading or writing. The use of reader-writer locks is beneficial for situation in which: (1) processes can be easily identified as either readers or writers, and (2) there are significantly more readers than writers, making the additional overhead of the reader-writer lock pay off in terms of increased concurrency of the readers.

The Dining-Philosophers Problem:

The dining philosopher's problem is a classic synchronization problem involving the allocation of limited resources among a group of processes in a deadlock-free and starvation-free manner: Consider five philosophers sitting around a table, in which there are five chopsticks evenly distributed and an endless bowl of rice in the center, as shown in the diagram below. (There is exactly one chopstick between each pair of dining philosophers.)

- These philosophers spend their lives alternating between two activities: eating and thinking.
- When it is time for a philosopher to eat, it must first acquire two chopsticks - one from their left and one from their right.
- When a philosopher thinks, it puts down both chopsticks in their original locations.



The situation of the dining philosophers

One simple solution is to represent each chop stick with a semaphore. A philosopher tries to grab a chop stick by executing a wait () operation on that semaphore; she releases her chop sticks by executing the signal () operation on the appropriate semaphores. Thus, the shared data are *Semaphore chopstick* [5]; where all elements of chopstick are initialized to 1. This solution is rejected as it could create a dead lock. Suppose that all five philosophers become hungry simultaneously and each grabs her left chop stick. All the elements of chop stick will now be equal to 0. When each philosopher tries to grab her right chopstick, she will be delayed forever.


```

do {
    wait(chopstick[i]);
    wait(chopstick[(i+1) % 5]);
    // eat
    // ...
    signal(chopstick[i]);
    signal(chopstick[(i+1) % 5]);
    // think
    // ...
} while (TRUE);

```

The structure of philosopher i .

Some potential solutions to the problem include:

- o Only allow four philosophers to dine at the same time. (Limited simultaneous processes.)
- o Allow philosophers to pick up chopsticks only when both are available, in a critical section. (All or nothing allocation of critical resources.)
- o Use an asymmetric solution, in which odd philosophers pick up their left chopstick first and even philosophers pick up their right chopstick first.

A deadlock-free solution to the dining philosophers problem does not necessarily guarantee a starvation-free one.

Deadlocks:

System Model:

- Finite number of resources is available in the system. These resources are distributed among a number of competing processes.
- Two general categories of resources can be distinguished.
 - i. Reusable Resources
 - ii. Consumable Resource
- i. **Reusable Resource:** A reusable resource is one that can be safely used by only one process at a time and is not depleted by that use. Processes obtained resource units that they later release for reuse by other processes.
Example: processors, I/O channels, I/O devices, primary and secondary memory, files, database, semaphores etc.
- ii. **Consumable Resource:** A consumable resource is one that can be created and destroyed. There is no limit on the number of variable resources of a particular type.
Example: interrupts, signals messages and information in I/O buffers.
- A process requests resources before using it, and it must release the resource after using it.
- The number of resources requested may not exceed the total number of resources available in the system.
- If the system have 4 printers, then the request for printer is equal to or less than 4.
- A process may utilize the resource if only the following sequence:
 - 1) **Request:** If the request is not guaranteed immediately, then the requesting process may wait until it acquire the resources.
 - 2) **Use:** The process can operate the resource.
 - 3) **Release:** The process can release the resources.
- Process 1 is holding resource 1 and requesting resource 2; Process 2 is holding resource 2 and requesting resource 3; Process 3 is holding resource 3 and holding resource 1.
- None of the process can proceed because all are waiting for a resource held by another blocked process.
- Unless one of the process detects the situation and is able to withdraw the request for a resource and release the one resource allocated to it, none of the processes will be ever be able to run.
- The below diagram shows the deadlock with three processes. Pictorially process is represented by circle and resource by square.



Three deadlocked processes

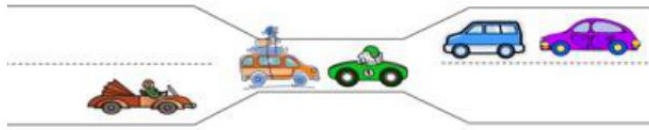
- Request is shown by dotted arrow from process to resource. Holding resource by process is shown by arrow.
- Deadlock is global condition rather than local one. Deadlock condition must be handled by operating system.

Deadlock Characterization:

Necessary Conditions for deadlock:

A deadlock is a condition in a system where a process cannot proceed because it needs to obtain a resource held by another process but it itself is holding a resource that the other process needs. More formally, four conditions have to be met for a deadlock to occur in a system:

- 1) **Mutual exclusion:** A resource can be held by at most one process.
- 2) **Hold and wait:** Processes that already hold resources can wait for another resource.
- 3) **Non-preemption:** A resource, once granted, cannot be taken away.
- 4) **Circular wait:** Two or more processes are waiting for resources held by one of the other processes.



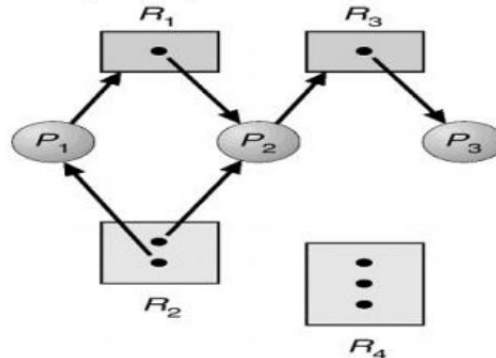
Resource-Allocation Graph:

Deadlocks can be described in terms of a directed graph called a **system resource allocation graph**. This graph consists of a set of vertices V and set of edges E . The set of vertices V is partitioned into two different types of nodes:

- P –the set consisting of all the active processes in the system and
- R –the set consisting of all resource types in the system.

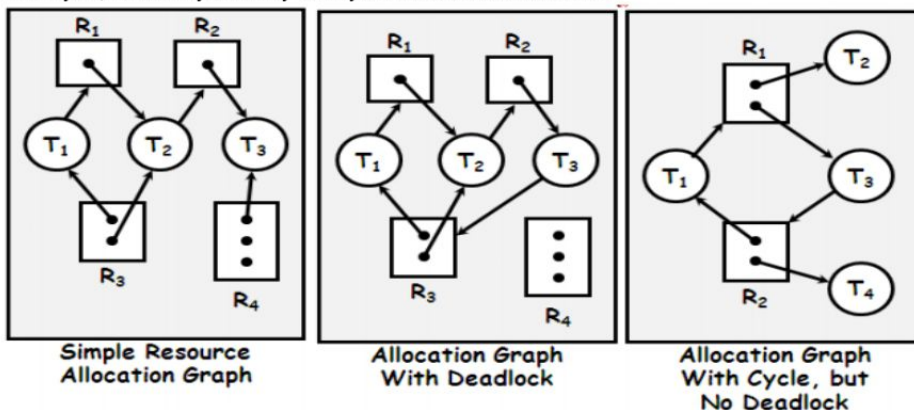
A directed edge from resource type R_j to process P_i is denoted by $R_j \rightarrow P_i$; it signifies that an instance of resource type R_j has been allocated to process P_i .

- A directed edge $P_i \rightarrow R_j$ is called a **request edge**.
- A directed edge $R_j \rightarrow P_i$ is called an **assignment edge**.



Resource-Allocation Graph

- A process is represented using circle and resource type is represented using a rectangle.
- Since resource type may have more than one instance, each instance is represented using a dot within the rectangle.
- A request edge points to the rectangle whereas an assignment edge must also designate one of the dots in the rectangle.
- When a process requests an instance of resource type, a request edge is inserted in the resource allocation graph.
- When this request can be fulfilled, the request edge is instantaneously transformed to an assignment edge.
- When the process no longer needs access to the resource, it releases the resource; as a result, the assignment edge is deleted.
- If a resource allocation graph does not have a cycle, then the system is not in a deadlocked state.
- If there is a cycle, then the system may or may not be in a deadlocked state.



Methods for handling deadlocks:

Deadlock problem can be dealt with in one of three ways –

- 1) Use a protocol to prevent or avoid deadlocks ensuring that the system will never enter a deadlock state
 - 2) Allow the system to enter a deadlock state, detect it and recover.
 - 3) Ignore the problem altogether and pretend that deadlocks never occur in the system.
- To ensure that deadlocks never occur, the system can use either deadlock prevention or a deadlock avoidance scheme.
 - **Deadlock prevention** provides a set of methods for ensuring that at least one of the necessary conditions (listed under deadlock characterization) cannot hold.
 - These methods prevent deadlocks by constraining how requests for resources can be made.
 - **Deadlock avoidance** requires that the OS be given in advance additional information concerning which resources a process will request and use during its lifetime. With this additional knowledge it can decide for each request whether or not the process should wait.
 - To decide whether the current request can be satisfied or must be delayed, the system must consider the resources currently available, the resources currently allocated to each process and the future requests and releases of each process.
 - **Deadlock detection** is fairly straightforward, but deadlock recovery requires either aborting processes or preempting resources, neither of which is an attractive alternative.
 - If deadlocks are neither prevented nor detected, then when a deadlock occurs the system will gradually slow down, as more and more processes become stuck waiting for resources currently held by the deadlock and by other waiting processes.
 - Unfortunately this slowdown can be indistinguishable from a general system slowdown when a real-time process has heavy computing needs.

Deadlock Prevention:

For a deadlock to occur, each of the four necessary conditions must hold. By ensuring that at least one of these conditions cannot hold, we can prevent the occurrence of a deadlock.

- 1) Mutual Exclusion
- 2) Hold and Wait
- 3) No Preemption
- 4) Circular Wait

1) **Mutual Exclusion:**

- The mutual exclusion condition must hold for non-sharable resources (printer).
- Sharable resources do not require mutually exclusive access and thus cannot be involved in a deadlock (read only file).
- We cannot prevent deadlocks by denying the mutual exclusion condition because some resources are intrinsically non-sharable.

2) **Hold and Wait:**

- 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 resources.
- One protocol that can be used requires each process to request and be allocated all its resources before it begins execution.
- An alternative protocol allows a process to request resources only when it has none.
- A process may request some resources and use them. Before it can request any additional resources, it must release all the resources that it is currently allocated.

Both these protocols have two main disadvantages.

1. First, resource utilization may be low since resources may be allocated but unused for a long period.
2. Second, starvation is possible. A process that needs several popular resources may have to wait indefinitely because at least one of the resources that it needs is always allocated to some other process.

3) **No Preemption:**

- Preemption of process resource allocations can prevent this condition of deadlocks, when it is possible.
- One approach is that if a process is forced to wait when requesting a new resource, then all other resources previously held by this process are implicitly released, (preempted), forcing this process to re-acquire the old resources along with the new resources in a single request, similar to the previous discussion.
- Another approach is that when a resource is requested and not available, then the system looks to see what other processes currently have those resources *and* are themselves blocked waiting for some other resource.
- If such a process is found, then some of their resources may get preempted and added to the list of resources for which the process is waiting.
- Either of these approaches may be applicable for resources whose states are easily saved and restored, such as registers and memory, but are generally not applicable to other devices such as printers and tape drives.

4) **Circular Wait:**

- The fourth and final condition for deadlocks is the circular wait condition.
- One way to ensure that this condition never holds is to impose a total ordering of all resource types and to require that each process requests resources in an increasing order of enumeration.
- Let $R = \{R_1, R_2, \dots, R_m\}$ be the set of resource types.
- We assign to each resource type a unique integer number, which allows us to compare two resources and to determine whether one precedes another in our ordering.
- Formally, we define a one-to-one function $F: R \rightarrow N$, where N is the set of natural numbers.
- Each process can request resources only in an increasing order of enumeration.
- That is, a process can initially request any number of instances of a resource type, R_i .
- After that, the process can request instances of resource type R_i if and only if $F(R_j) > F(R_i)$.
- If several instances of the same resource type are needed, a single request for all of them must be issued.
- Alternatively, whenever a process requests an instance of resource type R_j , it is required that, it has released any resources R_i such that $F(R_i) \geq F(R_j)$. If these two protocols are used, then the circular-wait condition cannot hold.

Deadlock Avoidance:

The general idea behind deadlock avoidance is to prevent deadlocks from ever happening, by preventing at least one of the aforementioned conditions. This requires more information about each process, AND tends to lead to low device utilization. (I.e. it is a conservative approach.) In some algorithms the scheduler only needs to know the *maximum* number of each resource that a process might potentially use. In more complex algorithms the scheduler can also take advantage of the *schedule* of exactly what resources may be needed in what order. When a scheduler sees that starting a process or granting resource requests may lead to future deadlocks, then that process is just not started or the request is not granted. A resource allocation *state* is defined by the number of available and allocated resources, and the maximum requirements of all processes in the system.

Safe state: 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. More formally, a state is safe if there exists a *safe sequence* of processes $\{P_0, P_1, P_2, \dots, P_N\}$ such that all of the resource requests for P_i can be granted using the resources currently allocated to P_i and all processes P_j where $j < i$. (I.e. if all the processes prior to P_i finish and free up their resources, then P_i will be able to finish also, using the resources that they have freed up.) 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.)

As long as the state is safe, the OS can avoid unsafe (and deadlocked states). In an unsafe state, the OS cannot prevent processes from requesting resources such that a deadlock occurs. The behavior of the processes controls unsafe states.

The idea is simply to ensure that the system will always remain in a safe state. Initially, the system is in a safe state. Whenever a process requests a resource that is currently available, the system must decide whether the resource can be allocated immediately or whether the process must wait. The request is granted only if the allocation leaves the system in a safe state.

There exist a total of 12 resources. Each resource is used exclusively by a process. The current state looks like this.

P_0, P_1 , and P_2 are the processes. Process P_0 requires 10 tape drives, process P_1 may need as many as 4, and process P_2 may need up to 9 tape drives. Suppose that, at time t_0 , process P_0 is holding 5 tape drives, process P_1 is holding 2, and process P_2 is holding 2 tape drives. At time t_0 , the system is in a safe state.

Process	Max Needs	Allocated	Current Needs
P0	10	5	5
P1	4	2	2
P2	9	3	7

The sequence $\langle P_1, P_0, P_2 \rangle$ satisfies the safety condition, since process P_1 can immediately be allocated all its tape drives and then return them (the system will then have 5 available tape drives), then process P_0 can get all its tape drives and return them (the system will then have 10 available tape drives), and finally process P_2 could get all its tape drives and return them (the system will then have all 12 tape drives available).

Suppose that, at time t_1 , process P_2 requests and is allocated 1 more tape drive. The system is no longer in a safe state. At this point, only process P_1 can be allocated all its tape drives. When it returns them, the system will have only 4 available tape drives. Since process P_0 is allocated 5 tape drives, but has a maximum of 10, it may then request 5 more tape drives. Since they are unavailable, process P_0 must wait. Similarly, process P_2 may request an additional 6 tape drives and have to wait, resulting in a deadlock.

```

process  holding  max claims  outstanding
                                requests
    A           4           6         2
    B           3           9         6
    C           4          11         7
unallocated: 2
deadlock-free sequence: A,B,C

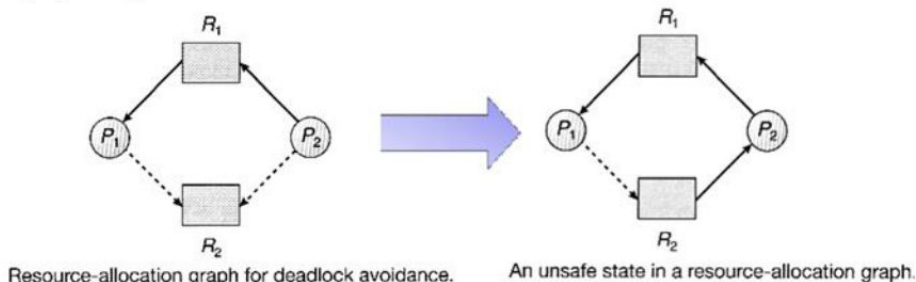
```

However, if B should have 7 instead of 6 outstanding requests, this sequence is not safe: deadlock exists.

Resource Allocation Graph:

If there is a resource allocation system with only one instance of each resource type, a variant of the resource allocation graph can be used for deadlock avoidance. In addition to the request and assignment edges, a **claim edge** is also used. This edge resembles a request edge in direction but is represented in the graph by a dashed line. When a process makes a request, the request can be granted only if converting the request edge to an assignment edge does not result in the formation of a cycle in the resource allocation graph. An algorithm for detecting a cycle in this graph requires an order of n^2 operations where n is the number of processes in the system.

Consider the resource-allocation graph shown below. Suppose that P_2 requests R_2 . Although R_2 is currently free, we cannot allocate it to P_2 , since this action will create a cycle in the graph as shown below. A cycle indicates that the system is in an unsafe state. If P_1 requests R_2 , and P_2 requests R_1 , then a deadlock will occur.



Banker's algorithm:

Q) Explain in detail about Banker's algorithm.

For resource categories that contain more than one instance the resource-allocation graph method does not work. A less efficient scheme called the **Banker's Algorithm**, which gets its name because it is a method that bankers could use to assure that when they lend out resources they will still be able to satisfy all their clients. (A banker won't loan out a little money to start building a house unless they are assured that they will later be able to loan out the rest of the money to finish the house.) When a process starts up, it must state in advance the maximum allocation of resources it may request, up to the amount available on the system. When a request is made, the scheduler determines whether granting the request would leave the system in a safe state. If not, then the process must wait until the request can be granted safely.

The banker's algorithm relies on several key data structures: (where n is the number of processes and m is the number of resource categories.)

- **Available:** vector of length m indicates the number of available resources of each type. If $Available[j] = k$, there are k instances of resource type R_j available
- **Max:** an $n \times m$ matrix defines the maximum demand of each process. If $Max[i,j] = k$, then process P_i may request at most k instances of resource type R_j .
- **Allocation:** an $n \times m$ matrix defines the number of resources of each type currently allocated to each process. If $Allocation[i,j] = k$, then process P_i is currently allocated k instances of resource type R_j .
- **Need:** an $n \times m$ matrix indicates the remaining resource need of each process. If $Need[i,j] = k$, then process P_i may need k more instances of resource type R_j to complete its task. Note that $Need[i,j] = Max[i,j] - Allocation[i,j]$.
- **Request:** It is a vector size m which indicates that the process P_i has requested for some resource.

Each row in the matrices Allocation and Need are treated as vectors and refer to them as $Allocation_i$ and $Need_i$, respectively. The vector $Allocation_i$ specifies the resources currently allocated to process P_i ; the vector $Need_i$ specifies the additional resources that process P_i may still request to complete its task.

Safety Algorithm: The Algorithm for finding out whether or not a system is in a safe state.

- 1) Let *Work* and *Finish* be vectors of length *m* and *n* respectively.

Work is a working copy of the available resources, which will be modified during the analysis. *Finish* is a vector of Booleans indicating whether a particular process can finish. (Or has finished so far in the analysis.) Initialize *Work* to Available, and *Finish* to false for all elements.

Work = Available

Finish [i] = false for i = 0, 1, ..., n- 1

- 2) Find a process *i* such that,

(a) ***Finish [i] = false***

(b) ***Need_i ≤ Work***

If no such *i* exists, then go to step 4.

- 3) ***Work = Work + Allocation_i***

Finish[i] = true

Goto step 2.

- 4) If ***Finish [i] == true*** for all *i*, then the system is in a safe state, because a safe sequence has been found.

This algorithm may require an order of $m \times n^2$ operations to decide whether a state is safe.

Resource - Request Algorithm: – algorithm which determines if requests can be safely granted.

Let *Request_i* be the request vector for process *P_i*. If *Request_i[j] = k*, then process *P_i* wants *k* instances of resource type *R_j*. When a request for resources is made by process *P_i*, the following actions are taken:

- 1) If ***Request_i ≤ Need_i*** go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim.
- 2) If ***Request_i ≤ Available***, go to step 3. Otherwise *P_i* must wait, since resources are not available.
- 3) Pretend to allocate requested resources to *P_i* by modifying the state as follows:

Available = Available – Request_i

Allocation_i = Allocation_i + Request_i

Need_i = Need_i – Request_i

If the resulting resource-allocation state is safe, the transaction is completed and process *P_i* is allocated its resources. But, if the new state is unsafe, then *P_i* must wait for *Request_i* and the old resource-allocation state is restored.

Example of Banker's Algorithm:

Five processes- *P₀*, *P₁*, *P₂*, *P₃*, *P₄* with three resource types: A, B, C with 10, 5, 7 instances. The content of the matrix *Need* is defined to be *Max – Allocation* (*Max = Allocation + Need*).

A snapshot of the system taken at time *T₀* is shown below

	<u>Allocation</u>	<u>Max</u>	<u>Available</u>	<u>Need</u>
	A B C	A B C	A B C	A B C
<i>P₀</i>	0 1 0	7 5 3	3 3 2	7 4 3
<i>P₁</i>	2 0 0	3 2 2		1 2 2
<i>P₂</i>	3 0 2	9 0 2		6 0 0
<i>P₃</i>	2 1 1	2 2 2		0 1 1
<i>P₄</i>	0 0 2	4 3 3		4 3 1

The system is in a safe state since the sequence **< *P₁*, *P₃*, *P₄*, *P₂*, *P₀* >** satisfies safety criteria

Now, if process *P₁* requests 1 instance of A and 2 instances of C. (*Request*[1] = (1, 0, 2))

To decide whether this request can be immediately granted, we first check that

Request ≤ *Available* (that is, (1,0,2) ≤ (3,3,2) ⇒ true.

Now we arrive at the following new state:

	<u>Allocation</u>	<u>Need</u>	<u>Available</u>
	A B C	A B C	A B C
<i>P₀</i>	0 1 0	7 4 3	2 3 0
<i>P₁</i>	3 0 2	0 2 0	
<i>P₂</i>	3 0 2	6 0 0	
<i>P₃</i>	2 1 1	0 1 1	
<i>P₄</i>	0 0 2	4 3 1	

Executing safety algorithm shows that sequence **< *P₁*, *P₃*, *P₄*, *P₀*, *P₂* >** satisfies safety requirement. When the system is in this state, a request for (3,3,0) by *P₄* cannot be granted, since the resources are not available. A request for (0,2,0) by *P₀* cannot be granted, even though the resources are available, since the resulting state is unsafe.

Disadvantages of the Banker's Algorithm:

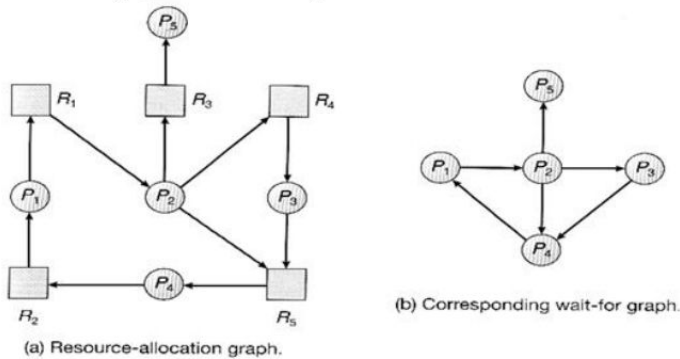
- It requires the number of processes to be fixed; no additional processes can start while it is executing.
- It requires that the number of resources remain fixed; no resource may go down for any reason without the possibility of deadlock occurring.
- It allows all requests to be granted in finite time, but one year is a finite amount of time. Similarly, all of the processes guarantee that the resources loaned to them will be repaid in a finite amount of time. While this prevents absolute starvation, some pretty hungry processes might develop.
- All processes must know and state their maximum resource need in advance.

Deadlock Detection:

If deadlocks are not avoided, then another approach is to detect when they have occurred and recover somehow. 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 type

If all resources have only a single instance, then we can define a deadlock detection algorithm that uses a variant of the resource allocation graph called a wait-for graph. 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.



A deadlock exists in the system if and only if the wait-for graph contains a cycle. To detect deadlocks, the system needs to maintain the wait-for graph and periodically invoke an algorithm that searches for a cycle in the graph. An algorithm to detect a cycle in the graph requires an order of n^2 operations where n is the number of vertices in the graph.

Several instances of a resource type:

The wait-for graph scheme is not applicable to resource allocation system with multiple instances of each resource type. For several instances of resource type, the algorithm employs several time varying data structures. They are:

- Available – a vector of length m indicates the number of available resources of each type
- Allocation – an $n \times m$ matrix defines the number of resources of each type currently allocated to each process
- Request – an $n \times m$ matrix indicates the current request of each process.

The detection algorithm outlined here is essentially the same as the Banker's algorithm, with two subtle differences:

Step 1) Let *Work* and *Finish* be vectors of length m and n respectively.

Initialize **Work=Available**.

For $i = 0, 1, \dots, n-1$, if *Allocation* $\neq 0$, then *Finish*[i] = *false*;
otherwise *Finish*[i] = *true*..

Step 2) Find a process i such that,

Finish*[i] = *false

Need* $_i \leq$ *Work

If no such i exists, then go to step 4.

Step 3) ***Work* = *Work* + *Allocation* $_i$**

Finish*[i] = *true

Go to step 2.

Step 4) The basic Banker's Algorithm says that if *Finish*[i] == *true* for all i , that there is no deadlock.

This algorithm is more specific, by stating that if *Finish*[i] == *false* for any process P_i , $0 \leq i < n$ then that process is specifically involved in the deadlock which has been detected.

This algorithm requires an order of $m \times n^2$ operations to detect whether the system is in a deadlocked state.

Five processes P_0 through P_4 and three resource types A (7 instances), B (2 instances), and C (6 instances). Snapshot at time T_0 :

	<u>Allocation</u>			<u>Request</u>			<u>Total</u>		
	A	B	C	A	B	C	A	B	C
P_0	0	1	0	0	0	0	7	2	6
P_1	2	0	0	2	0	2	<u>Allocated</u>		
P_2	3	0	3	0	0	0	7	2	6
P_3	2	1	1	1	0	0	<u>Available</u>		
P_4	0	0	2	0	0	2	0	0	0

Sequence $\langle P_0, P_2, P_3, P_1, P_4 \rangle$ will result in *Finish*[i] = *true* for all i .

Detection algorithm usage:

Two factors decide when to invoke the detection algorithm.

- How often is a deadlock likely to occur?
- How many processes will be affected by deadlock when it happens?

There are two obvious approaches, each with trade-offs:

- 1) Do deadlock detection after every resource allocation which cannot be immediately granted. This has the advantage of detecting the deadlock right away, while the minimum number of processes are involved in the deadlock. The down side of this approach is the extensive overhead and performance hit caused by checking for deadlocks so frequently.
- 2) Do deadlock detection only when there is some clue that a deadlock may have occurred, such as when CPU utilization reduces to 40% or some other magic number. The advantage is that deadlock detection is done much less frequently, but the down side is that it becomes impossible to detect the processes involved in the original deadlock, and so deadlock recovery can be more complicated and damaging to more processes.

Recovery from Deadlock:

There are three basic approaches to recovery from deadlock:

- 1) Inform the system operator, and allow him/her to take manual intervention.
- 2) Terminate one or more processes involved in the deadlock
- 3) Preempt resources

Process Termination:

To eliminate deadlocks by aborting a process, use one of the two methods. In both methods, the system reclaims all resources allocated to the terminated processes.

- ❖ **Abort all deadlocked processes** – Breaks the deadlock cycle, the deadlocked processes may have computed for a long time and the results of partial computations must be discarded and will have to be recomputed later.
- ❖ **Abort one process at a time until the deadlock cycle is eliminated** – Incurs considerable overhead, since after each process is aborted, a deadlock detection algorithm must be invoked to determine whether any processes are still deadlocked.

If the partial termination method is used, then we must determine which deadlocked process should be terminated. Abort those processes whose termination will incur minimum costs.

Many factors may affect which process is chosen including:

- 1) What the priority of the process is?
- 2) How long the process has computed and how much longer the process will compute before completing its designated task?
- 3) How many and what type of resources the process has used?
- 4) How many more resources the process needs in order to complete?
- 5) How many processes will need to be terminated?
- 6) Whether the process is interactive or batch?

Resource Preemption:

To eliminate deadlocks using resource preemption, preempt some resources from processes and give these resources to other processes until the deadlock cycle is broken.

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

- **Selecting a victim** – Which resources and which processes are to be preempted?
- **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.
- **Starvation** – How do you guarantee that a process won't starve because its resources are constantly being preempted? One option would be to us.

In a system where victim selection is based primarily on cost factors, it may happen that the same process is always picked as a victim. So this process never completes its designated task, a starvation situation that must be dealt with in any practical system. A process can be picked as a victim only a finite number of times.