OPERATING SYSTEMS

MODULE-3

Synchronization Tools: The Critical Section Problem, Peterson's Solution, Mutex Locks, Semaphores, Monitors, Classic problems of Synchronization.
Deadlocks: system Model, Deadlock characterization, Methods for handling Deadlocks, Deadlock prevention, Deadlock avoidance, Deadlock detection, Recovery from Deadlock.

THE CRITICAL SECTION PROBLEM

Consider a system consisting of n processes **{P0, P1, ..., Pn−1}.** Each process has a segment of code, called a **critical section**, in which the process may be accessing and updating data that is shared with at least one other process.

The important feature of the system is that, when one process is executing in its critical section, no other process is allowed to execute in its critical section. That is, no two processes are executing in their critical sections at the same time.

The **critical-section problem** is to design a protocol that the processes can use to **synchronize their activity** so as to cooperatively share data. Each process must request permission to enter its critical section.

The section of code implementing this request is the **entry section**. The critical section may be followed by an **exit section**. The remaining code is the **remainder section**. The entry section and exit section are enclosed in boxes to highlight these important segments of code.

```
while (true) {
    entry section
        critical section
    exit section
        remainder section
}
```

Figure General structure of a typical process.

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

1. Mutual exclusion

If process $\mathbf{P_i}$ 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 some processes wish to enter their critical sections, then only those processes that are not executing in their remainder sections can participate in deciding which will enter its critical section next, and this selection cannot be postponed indefinitely.

2. Bounded waiting

There exists a bound, or limit, 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.

Consider as an example a **kernel data structure** that maintains a list of all open files in the system. This list must be modified when a new file is opened or closed (adding the file to the list or removing it from the list). If two processes were to open files simultaneously, the separate updates to this list could result in a race condition.

The critical-section problem could be solved simply in a single-core environment if we could prevent interrupts from occurring while a shared variable was being modified. In this way, we could be sure that the current sequence of instructions would be allowed to execute in order without preemption. No other instructions would be run, so no unexpected modifications could be made to the shared variable. Unfortunately, this solution

is not as feasible in a multiprocessor environment. Disabling interrupts on a multiprocessor can be time consuming, since the message is passed to all the processors. This message passing delays entry into each critical section, and system efficiency decreases. Also consider the effect on a system's clock if the clock is kept updated by interrupts.

Two general approaches are used to handle critical sections in operating systems:

- **Preemptive Kernels**
- **Non-preemptive Kernels**

PREEMPTIVE KERNEL

A **Preemptive Kernel** allows a process to be preempted while it is running in kernel mode. Preemptive kernels, must be carefully designed to ensure that shared kernel data are free from race conditions. Preemptive kernels are especially difficult to design for SMP architectures, since in these environments it is possible for two kernel-mode processes to run simultaneously on different CPU cores.

NON-PREEMPTIVE KERNEL

A **non-preemptive kernel** does not allow a process running in kernel mode to be preempted. Obviously, a non-preemptive kernel is essentially free from race conditions on kernel data structures, as only one process is active in the kernel at a time.

Peterson's Solution

Peterson's Solution is a classic software-based algorithm used in operating systems for ensuring mutual exclusion in critical sections of two processes. It addresses the **critical section problem** and ensures that two processes do not access a shared resource simultaneously, thereby avoiding race conditions.

Key Features of Peterson's Solution

1. **Mutual Exclusion**:
   - Only one process can access the critical section at a time.
2. **Progress**:
   - If no process is in the critical section, a process waiting to enter will not be delayed unnecessarily.
3. **Bounded Waiting**:
   - There is a bound on the number of times other processes can enter their critical sections before a waiting process is granted access.
4. **Two Processes**:
   - The solution is designed for two processes, usually labeled as P0 and P1.

Mechanism

Peterson's solution uses two shared variables:

1. **flag[]**:
   - An array where each process has a flag indicating if it wants to enter the critical section.
   - flag[i] = true means process i wants to enter its critical section.
2. **turn**:
   - A shared variable indicating whose turn it is to enter the critical section.
   - It helps ensure mutual exclusion.

Algorithm

For two processes (P0 and P1):

1. **Entry Section**:
    - A process signals its intent to enter the critical section by setting its flag to true.
    - It then sets turn to the other process, indicating it is giving priority to the other process if it wants to enter the critical section.
2. **Critical Section**:
    - A process enters the critical section only if:
        - The other process has not set its flag to true, OR
        - It is the process's turn.
3. **Exit Section**:
    - After exiting the critical section, the process resets its flag to false, allowing the other process to proceed.
4. **Remainder Section**:
    - The rest of the code outside the critical and entry/exit sections.

Code Implementation

Here is a pseudocode representation:

```
c
Copy code
// Shared variables
boolean flag[2] = {false, false}; // Flags for P0 and P1
int turn; // Indicates whose turn it is

// Code for Process P0
flag[0] = true;         // P0 wants to enter critical section
turn = 1;               // Give turn to P1
while (flag[1] && turn == 1);  // Wait if P1 wants to enter and it's P1's turn

// Critical Section
<critical section>

// Exit Section
flag[0] = false;        // P0 no longer needs the critical section

// Remainder Section
<remainder section>

// Code for Process P1
flag[1] = true;         // P1 wants to enter critical section
turn = 0;               // Give turn to P0
while (flag[0] && turn == 0);  // Wait if P0 wants to enter and it's P0's turn

// Critical Section
<critical section>

// Exit Section
flag[1] = false;        // P1 no longer needs the critical section

// Remainder Section
<remainder section>
```

Explanation

1. **Entry Section**:
   - A process sets its flag to true to indicate its intent to enter the critical section.
   - It sets turn to the other process, showing willingness to allow the other process to enter if needed.
2. **Critical Section**:
   - The process enters the critical section only if the other process has not set its flag to true or if the current process's turn matches the other process.
3. **Exit Section**:
   - The process resets its flag to false, signaling that it has left the critical section.

Properties of Peterson's Solution

1. **Mutual Exclusion**:
   - Only one process can set flag[i] to true and pass the while loop to enter the critical section at a time.
2. **Progress**:
   - If one process is not interested in entering the critical section (flag[j] = false), the other process will not be delayed.
3. **Bounded Waiting**:
   - Each process gets a fair chance to execute the critical section since turn alternates between the two processes.

Limitations

1. **Two Processes Only**:
   - Peterson’s solution works only for two processes. Extending it to more processes increases complexity and inefficiency.
2. **Assumes Atomic Operations**:
   - The solution assumes that reading/writing to shared variables (like flag[] and turn) is atomic, which may not hold in all systems.
3. **Busy Waiting**:
   - While a process waits to enter the critical section, it loops continuously, consuming CPU cycles.
4. **Modern Systems**:
   - Modern multi-core systems may reorder instructions (due to compiler optimizations or CPU caching), which can break the assumptions of Peterson's solution.

Importance

Peterson's solution is a foundational concept in operating systems and concurrent programming. While it's not commonly used in practice (due to hardware-based alternatives like semaphores and mutexes), it is a critical theoretical model for understanding synchronization and mutual exclusion.

MUTEX LOCKS

In multitasking programming, mutex locks, also referred to as mutual exclusion locks, are synchronization basic functions used to prevent simultaneous possession of resources that are shared by numerous threads or procedures. The word "mutex" means "mutual exclusion."

What are Mutex Locks?

A mutex lock makes it possible to implement mutual exclusion by limiting the number of threads or processes that can simultaneously acquire the lock. A single thread or procedure has to first try to obtain the mutex lock for something that is shared before it can access it. The seeking string or procedure gets halted and placed in a state of waiting as long as the encryption key turns into accessible if it is being organized by a different thread or process. The thread or process is able to use the resource that has been shared after acquiring the lock. When finished, it introduces the lock so that different threads or processes can take possession of it.

Components of Mutex Locks

Below we discuss some of the main components of Mutex Locks.

Mutex Variable − A mutex variable is used to represent the lock. It is a data structure that maintains the state of the lock and allows threads or processes to acquire and release it.

Lock Acquisition − Threads or processes can attempt to acquire the lock by requesting it. If the lock is available, the requesting thread or process gains ownership of the lock. Otherwise, it enters a waiting state until the lock becomes available.

Lock Release − Once a thread or process has finished using the shared resource, it releases the lock, allowing other threads or processes to acquire it.
Types of Mutex Locks

Mutex locks come in a variety of forms that offer varying levels of capabilities and behavior. Below are a few different kinds of mutex locks that are frequently used &mi nus;

Recursive Mutex

A recursive mutex enables multiple lock acquisitions without blocking an operating system or procedure. It keeps track of the number of occasions it was previously purchased and needs the same amount of discharges before it can be completely unlocked.

Example

Think about a data framework where a tree-like representation of the directory structure exists. Each node in the tree stands for a directory, and several threads are simultaneously going through the tree to carry out different operations. The use of a recursive mutex can prevent conflicts. A thread is a program that passes through a directory node, takes control of the lock, executes its actions, and then declares itself repeatedly to enter subdirectories. The recursive mutex enables multiple acquisitions of the identical lock without blocking the thread, maintaining appropriate traversal and synchronization.

Error-Checking Mutex

When lock processes, an error-checking mutex executes further error checking. By hindering looping lock appropriation, it makes a guarantee that an application or procedure doesn't take over a mutex lock it presently already holds.

Example

Consider a multi-threaded program in which multiple processes update a common contrary fluctuating. The counter is secured by a mutex lock to avoid conflicts and race conditions. A fault occurs if a thread unintentionally attempts to obtain the mutex lock it currently possesses. Such coding errors can be found and quickly fixed by programmers thanks to the error-checking mutex.

Times Mutex

For a predetermined amount of time, an algorithm or procedure can try to acquire a lock using a timed mutex. The acquiring functioning fails if the security key fails to become accessible during the allotted time, as well as if the thread/process can respond appropriately.

Example

Think about a system that operates in real-time that has a number of operations or strings that require access to a constrained number of resources that are fixed. Each assignment requires a particular resource to complete it. Nevertheless, a task might need to execute a

different course of action or indicate a mistake if it can't get the needed resource in an appropriate period of time. Each task can make an attempt to obtain the material for a set amount of time through the use of a timed mutex. The task at hand can continue with an alternate method or take the necessary action depending on the time limit circumstance if the resource in question is not accessible within the allotted time.

Priority Inheritance Mutex

A particular kind of mutex that aids in reducing the inversion of priority issues is the priority inheritance mutex (also known as the priority ceiling mutex). The priority inheritance mutex momentarily increases the ranking associated with the low-priority organization to the level of the highest-priority organization awaiting the lock to be released as a low-priority string or procedure is holding the lock and a higher-priority string or procedure requires it.

Example

Consider a system that operates in real-time with several threads operating at various priorities. These processes have access to shared assets that are mutex lock protected. A priority inheritance mutex can be used to avoid highest-priority inversion, which happens when a lower-priority string blocks a higher-priority string by retaining the lock. In order to ensure that the highest-priority string receives immediate access to the material, the mutex momentarily raises the lower-priority thread's importance to coincide with that associated with the highest-priority thread awaiting the lock.

Read-Write Mutex

A read-write lock is a synchronization system that permits several threads or procedures to access the same resource concurrently whereas implementing exclusion between them throughout write activities, though it does not constitute solely an instance of a mutex lock.

Example

Only one thread is in charge of drafting new frames into the provided buffer whereas various threads read provides and from it in an immediate time online video implementation. A read-write lock can be used to permit simultaneous reading but reserved writing. For unrestricted access to the structures, numerous reading strings can concurrently gain the read lock. Yet, the writer string develops the write lock solely when it wishes to modify the storage device, making sure that no other threads are able to read or write throughout the procedure for updating.

Use Cases of Mutex Locks

We will now talk about some of the use cases of Mutex Locks.

Shared Resource Protection − Mutex locks are commonly used to protect shared resources in multi-threaded or multi-process environments. They ensure that only one thread or process can access the shared resource at a time, preventing data corruption and race conditions.

Critical Sections − Mutex locks are used to define critical sections in a program, where only one thread at a time can execute the code inside the critical section. This ensures the integrity of shared data and prevents concurrent access issues.

Synchronization − Mutex locks enable synchronization between threads or processes, allowing them to coordinate their actions and access shared resources in a controlled manner. They ensure that certain operations are performed atomically, avoiding conflicts and ensuring consistency.

Deadlock Avoidance − Mutex locks can be used to prevent deadlock situations where multiple threads or processes are waiting indefinitely for resources held by each other. By following proper locking protocols and avoiding circular dependencies, deadlock situations can be avoided.
Example

Let us explore an implemented example of Mutex Locks in Python below.

In this example, multiple threads are created to increment a shared resource **(shared_resource)** by 1. The critical section where the shared resource is modified is protected by a mutex lock **(mutex)**. Each thread acquires the lock before entering the critical section and releases it after completing the critical section. The mutex lock ensures that only one thread can modify the shared resource at a time, preventing race conditions and ensuring the correctness of the final result.

```
Open Compiler
import threading

# Shared resource
shared_resource = 0

# Mutex lock
mutex = threading.Lock()

# Function to increment the shared resource
def increment():
  global shared_resource
```

```
  for _ in range(100000):
    # Acquire the lock
    mutex.acquire()

    # Critical section
    shared_resource += 1

    # Release the lock
    mutex.release()

# Create multiple threads
threads = []
for _ in range(5):
  thread = threading.Thread(target=increment)
  threads.append(thread)
# Start the threads
for thread in threads:
  thread.start()

# Wait for all threads to complete
for thread in threads:
  thread.join()

# Print the final value of the shared resource
print("Shared Resource:", shared_resource)
```

Output

Shared Resource: 500000

Benefits of Mutex Locks

Mutex locks benefit multitasking in a number of ways−

- **Mutual Exclusion** − Mutex locks ensure that only one thread or process can hold the lock at a time, preventing race conditions and ensuring predictable behavior in multi-threaded/multi-process scenarios.
- **Synchronization** − Mutex locks allow programs and threads to coordinate access to shared resources, preventing unauthorized access and enabling controlled access when an application or process attempts to acquire a locked mutex.
- **Simple and Portable** − Mutex locks are widely supported in various programming languages and OS platforms and are relatively easy to use.
- **Efficiency** − Mutex locks generally work well with little contention, generating minimal overhead while the lock is not being contended.

Drawbacks of Mutex Locks

Mutex locks have drawbacks to multitasking in a number of ways –

- **Potential Deadlocks**− Improper use of mutex locks can lead to deadlocks, where multiple threads or processes are held up indefinitely while waiting for locks that will never be released.
- **Priority Inversion**− Priority inversion can occur when a high-priority thread or process is blocked while waiting for a lock held by a lower-priority thread or process.
- **Resource Utilization**− Mutex locks can result in poor resource utilization if threads or processes are frequently blocked while waiting for locks.

Conclusion

Mutex locks are useful synchronization of basic functions in programming concurrently, to sum up. By enabling just a single thread or procedure to make use of a resource that is shared at a time, they enable mutual exclusion, synchronization, and aid in preventing race conditions. Mutex locks provide a standardized method for synchronization and are easy to operate and widely accepted.

SEMAPHORES

Semaphore was proposed by **Edsger Dijkstra** which is a very significant technique to manage concurrent processes. Semaphores are integer variables that are used to solve the critical section problem by using two atomic operations, **wait** and **signal** that are used for process synchronization.

Wait

The wait() operation was originally termed P (from the Dutch proberen, "to test"); The definition of wait() is as follows:

```
wait(S) {
        while (S <= 0)
         ; // busy wait
         S--;
         }
```

Signal

signal() was originally called V (from **verhogen, "to increment").** The definition of signal() is as follows:

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

There are two main types of semaphores. They are

- **Counting Semaphores**
- **Binary Semaphores**

COUNTING SEMAPHORES

These are integer value semaphores and have an unrestricted value domain. These semaphores are used to coordinate the resource access, where the semaphore count is the number of available resources. If the resources are added, semaphore count automatically incremented and if the resources are removed, the count is decremented.

For example, Suppose there are 4 processes P1, P2, P3, P4, and they all call wait operation on S(initialized with 4). If another process P5 wants the resource then it should wait until one of the four processes calls the signal function and the value of semaphore becomes positive.

BINARY SEMAPHORES

The binary semaphores are like counting semaphores but their value is restricted to 0 and 1. The wait operation only works when the semaphore is 1 and the signal operation succeeds when semaphore is 0. It is sometimes easier to implement binary semaphores than counting semaphores. 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 (thereby decrementing the count). When a process releases a resource, it performs a signal() operation (incrementing the count). When the count for the semaphore goes to 0, all resources are being used. After that, processes that wish to use a resource will block until the count becomes greater than 0.

Let there be two processes P1 and P2 and a semaphore s is initialized as 1. Now if suppose P1 enters in its critical section then the value of semaphore s becomes 0. Now if P2 wants to enter its critical section then it will wait until s > 0, this can only happen when P1 finishes its critical section and calls V operation on semaphore **s.**

Advantages of Semaphore

- It allows flexible management of resources.
- It is machine-independent.
- It allows multiple threads to access the critical section.
- It does not allow multiple processes to access the critical section at the same time.

Disadvantages of Semaphore

- Priority inversion.
- The OS needs to keep track of Wait and Signal operations.
- This is not a practical method for large-scale use.

It may cause deadlock if the Wait and Signal operations are executed in the wrong order.

MONITORS

Although semaphores provide a convenient and effective mechanism for process synchronization, using them incorrectly can result in timing errors that are difficult to detect. These errors happen only if particular execution sequences take place, and these sequences do not always occur. Various types of errors can be generated easily when programmers use semaphores or mutex locks incorrectly to solve the critical-section problem.

A **monitor type is an ADT** that includes a set of programmer-defined operations that are provided with mutual exclusion within the monitor. The monitor type also declares the variables whose values define the state of an instance of that type, along with the bodies of functions that operate on those variables. The representation of a monitor type cannot be used directly by the various processes. Thus, a function defined within a monitor can access only those variables declared locally within the monitor and its formal parameters.

Similarly, the local variables of a monitor can be accessed by only the local functions. The monitor construct ensures that only one process at a time is active within the monitor.

The structure of philosopher *i is*

```
while (true) {
        wait(chopstick[i]);
        wait(chopstick[(i+1) % 5]);
         . . .
         /* eat for a while */
         . . .
        signal(chopstick[i]);
        signal(chopstick[(i+1) % 5]);
         . . .
        /* think for awhile */
        . . .
      }
```

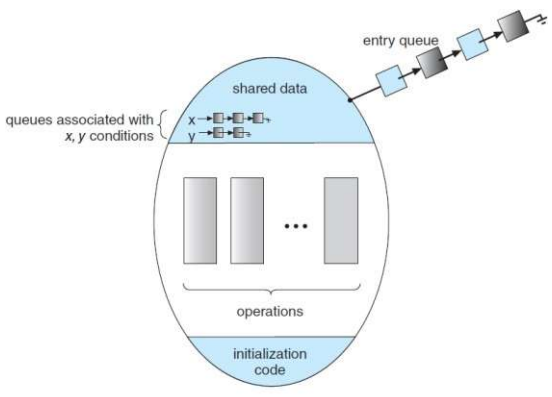

Figure Monitor with condition variables.

CLASSIC PROBLEMS OF SYNCHRONIZATION

Below are some of the classical problem depicting flaws of process synchronization in systems where cooperating processes are present.

The following problems of synchronization are considered as classical problems:

1. **The Bounded-buffer Problem**
2. **The Readers and Writers Problem**
3. **The Dining-Philosophers Problem**

THE BOUNDED-BUFFER PROBLEM

This problem is generalised in terms of the **Producer Consumer problem**, where a **finite** buffer pool is used to exchange messages between producer and consumer processes.

In our problem, the producer and consumer processes share the following data structures:

```
int n;
semaphore mutex = 1;
semaphore empty = n;
semaphore full = 0
```

We assume that the pool consists of n buffers, each capable of holding one item. The mutex binary 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 the value 0.

THE READERS-WRITERS PROBLEM

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

Obviously, if two readers access the shared data simultaneously, no adverse effects will result. However, if a writer and some other process access the database simultaneously, chaos may ensue. To ensure that these difficulties do not arise, we require that the writers have exclusive access to the shared database while writing to the database. This synchronization problem is referred to as the **readers–writers problem.**The solution to the first readers–writers problem is

One set of data is shared among a number of processes. Once a writer is ready, it

performs its write. Only one writer may write at a time. If a process is writing, no other process can read it. If at least one reader is reading, no other process can write. Readers may not write and only read.

THE DINING-PHILOSOPHERS PROBLEM

Consider five philosophers who spend their lives thinking and eating. The philosophers share a circular table surrounded by five chairs, each belonging to one philosopher. In the center of the table is a bowl of rice, and the table is laid with five single chopsticks.

Figure The situation of the dining philosophers.

When a philosopher thinks, she does not interact with her colleagues. From time to time, a philosopher gets hungry and tries to pick up the two chopsticks that are closest to her (the chopsticks that are between her and her left and right neighbors). A philosopher may pick up only one chopstick at a time. Obviously, she cannot pick up a chopstick that is already in the hand of a neighbour.

One simple solution is to represent each chopstick with a semaphore. A philosopher tries to grab a chopstick by executing a wait() operation on that semaphore. She releases her chopsticks by executing the signal() operation on the appropriate semaphores.

The structure of philosopher *i is*

```
while (true) {
        wait(chopstick[i]);
        wait(chopstick[(i+1) % 5]);
         . . .
         /* eat for a while */
         . . .
        signal(chopstick[i]);
        signal(chopstick[(i+1) % 5]);
        . . .
       /* think for awhile */
       . . .
     }
```

Although this solution guarantees that no two neighbors are eating simultaneously, it nevertheless must be rejected because it could create a **deadlock.**

Suppose that all five philosophers become hungry at the same time and each grabs her left chopstick. All the elements of chopstick will now be equal to 0. When each philosopher tries to grab her right chopstick, she will be delayed forever. When a philosopher wants to eat the rice, he will wait for the chopstick at his left and picks up that chopstick. Then he waits for the right chopstick to be available, and then picks it too. After eating, he puts both the chopsticks down. But if all five philosophers are hungry simultaneously, and each of them pickup one chopstick, then a deadlock situation occurs because they will be waiting for another chopstick forever.

The possible solutions for this are:

- A philosopher must be allowed to pick up the chopsticks only if both the left and right chopsticks are available.
- Allow only four philosophers to sit at the table. That way, if all the four philosophers pick up four chopsticks, there will be one chopstick left on the table. So, one philosopher can start eating and eventually, two chopsticks will be available. In this way, deadlocks can be avoided.

Deadlock System Model

In a computer system a **deadlock** is where two or more processes are unable to proceed because each process is waiting for the other to release a resource that it needs to continue execution. In other words, a **deadlock** occurs when two or more processes are in a circular wait state, and none of them can release the resources they hold until they receive the resources they are waiting for.

Deadlock System Model − The **Deadlock System model** is a way to describe and analyze systems that may be prone to deadlocks, which occur when two or more processes are unable to proceed because they are each waiting for the other to release a resource. Below are the components of this model −

- **Resources** − The system has a set of **resources** that are shared among processes. These **resources** can be hardware or software components, such as memory, files, printers, or network connections. Each resource is identified by a unique name or identifier.
- **Processes** − The system has a set of **processes** that request and release resources. **Processes** are units of execution that can be started, suspended, resumed, and terminated. Each process is identified by a unique process ID.

- **Resource Allocation** − Each resource can be in one of two states , allocated or available. A resource that is allocated to a process cannot be used by any other process until it is released.
- **Request and Release** − A process can request a resource by sending a **request** to the system. If the resource is available, it will be allocated to the process. When a process is finished using a resource, it must **release** it so that it can be used by other processes.
- **Resource Dependency** − Some processes may require multiple resources to complete their tasks. A **resource dependency** graph can be used to represent the relationships between processes and resources and to detect potential deadlocks.
- **Deadlock Detection** − A deadlock can occur when two or more processes are waiting for resources that are being held by other processes, creating a circular dependency. **Deadlock detection** algorithms can be used to detect when a deadlock has occurred, so that corrective action can be taken.
- **Deadlock Resolution** − Once a deadlock has been detected, it can be resolved by breaking the circular dependency between the processes. This can be done by releasing one or more resources that are being held by a process, or by preempting one or more processes that are holding resources. The Working of some of the techniques are given below –
- **Resource preemption** is a technique used to break the circular wait condition of a deadlock. The operating system can preempt resources from one or more processes involved in the deadlock and allocate them to the processes that need them. Preemption can be done either selectively or globally. In selective preemption, only the resources that are required to resolve the deadlock are preempted, while in global preemption, all the resources held by the deadlocked processes are preempted.
- When a process is terminated, all the resources held by the **process are released**, and other processes can proceed. However, this approach can lead to data loss and inconsistency if the terminated process was in the middle of a critical task.
- **Deadlock Avoidance** − **Deadlock avoidance** is a technique used to prevent the occurrence of deadlocks in a computer system. The goal of deadlock avoidance is to ensure that all resources required by a process are available before the process starts execution, thereby avoiding the possibility of deadlock.

There are several algorithms that can be used for deadlock avoidance, including the banker's algorithm and the resource allocation graph. These algorithms use a mathematical model to analyze resource allocation and to determine whether a process should be allowed to start or wait for resources.

- **The banker's algorithm** is a widely used method for deadlock avoidance. It is a resource allocation algorithm that checks whether a requested resource can be granted to a process without causing a deadlock. The algorithm works by simulating the allocation of resources and checking whether a safe state can be reached. A safe state is a state where all processes can complete their execution without causing a deadlock.
- **The resource allocation graph** is another method for deadlock avoidance. It represents the allocation of resources as a directed graph. Each process is represented by a node, and

each resource is represented by an edge. The algorithm checks for cycles in the graph to determine whether a deadlock has occurred. If a cycle is detected, the process requesting the resource is blocked until the required resource becomes available.

Conclusion

In summary, the **deadlock system model** is a crucial idea in the design and management of operating systems. It outlines the potential causes of deadlocks as well as strategies for avoiding and resolving them. It is crucial to put into practice efficient prevention measures for deadlocks since they might result in substantial performance degradation and data loss.

DEADLOCK CHARACTERIZATION

In a multiprogramming environment, several threads may compete for a finite number of resources. A thread requests resources; if the resources are not available at that time, the thread enters a waiting state. Sometimes, a waiting thread can never again change state, because the resources it has requested are held by other waiting threads. This situation is called a **deadlock.**

Deadlock is a situation in which every process in a set of processes is waiting for an event that can be caused only by another process in the set.

A deadlock situation can arise if the following four conditions hold simultaneously in a system

- **Mutual Exclusion**
- **Hold and Wait**
- **No Preemption**
- **Circular Wait**

MUTUAL EXCLUSION

At least one resource must be held in a non-sharable mode; that is, only one thread at a time can use the resource. If another thread requests that resource, the requesting thread must be delayed until the resource has been released.

HOLD AND WAIT

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

NO PREEMPTION

Resources cannot be preempted; that is, a resource can be released only voluntarily by the thread holding it, after that thread has completed its task.

CIRCULAR WAIT

A set {T0, T1, ..., Tn } of waiting threads must exist such that T0 is waiting for a resource held by T1, T1 is waiting for a resource held by T2, ..., Tn−1 is waiting for a resource held by Tn, and Tn is waiting for a resource held by T0.

METHODS OF HANDLING DEADLOCK

Generally speaking, we can deal with the deadlock problem in one of three ways:

- We can ignore the problem altogether and pretend that deadlocks never occur in the system.
- We can use a protocol to prevent or avoid deadlocks, ensuring that the system will never enter a deadlocked state.
- We can allow the system to enter a deadlocked state, detect it, and recover.

The first solution is the one used by most operating systems, including Linux and Windows. It is then up to kernel and application developers to write programs that handle deadlocks, typically using approaches outlined in the second solution.

Some systems such as databases adopt the third solution, allowing deadlocks to occur and then managing the recovery. To ensure that deadlocks never occur, the system can use either a **deadlock prevention** or a **deadlock-avoidance** scheme.

DEADLOCK PREVENTION

Deadlock prevention provides a set of methods to ensure that at least one of the necessary conditions cannot hold. These methods prevent deadlocks by constraining how requests for resources can be made.

DEADLOCK AVOIDANCE

Deadlock avoidance requires that the OS be given additional information in advance concerning which resources a thread will request and use during its lifetime. With this additional knowledge, the OS can decide for each request whether or not the thread 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 thread**, and the **future requests** and **releases of each thread.** If a system does not employ either a deadlock-prevention or a deadlock avoidance algorithm, then a deadlock situation may arise. In this environment, the system can provide an algorithm that examines

the state of the system to determine whether a deadlock has occurred and an algorithm to recover from the deadlock.

In the absence of algorithms to **detect and recover** from deadlocks, we may arrive at a situation in which the system is in a deadlocked state yet has no way of recognizing what has happened.

DEADLOCK DETECTION AND RECOVERY

If a system does not employ either a deadlock-prevention or a deadlock avoidance algorithm, then a deadlock situation may arise. In this environment, the system can provide an algorithm that examines the state of the system and an algorithm to recover from the deadlock. In the absence of algorithms to **detect and recover** from deadlocks, we may arrive at a situation in which the system is in a deadlocked state yet has no way of recognizing what has happened.

DEADLOCK PREVENTION

For a deadlock to occur, each of the four necessary conditions must hold. They four conditions are

- **Mutual Exclusion**
- **Hold and Wait**
- **No Preemption**
- **Circular Wait**

By ensuring that at least one of these conditions cannot hold, we can prevent the occurrence of a deadlock.

MUTUAL EXCLUSION

The mutual-exclusion condition must hold. That is, at least one resource must be non-sharable. Sharable resources do not require mutually exclusive access and thus cannot be involved in a deadlock. Read-only files are a good example of a sharable resource. If several threads attempt to open a read-only file at the same time, they can be granted simultaneous access to the file.

A thread never needs to wait for a sharable resource. In general, however, we cannot prevent deadlocks by denying the mutual-exclusion condition, because some resources are intrinsically non-sharable. For example, a mutex lock cannot be simultaneously shared by several threads.

HOLD AND WAIT

To ensure that the hold-and-wait condition never occurs in the system, we must guarantee that, whenever a thread requests a resource, it does not hold any other resources. One protocol that we can use requires each thread to request and be allocated all its resources before it begins execution. This is, of course, impractical for most applications due to the dynamic nature of requesting resources.

An alternative protocol allows a thread to request resources only when it has none. A thread 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.

First, resource utilization may be low, since resources may be allocated but unused for a long period. For example, a thread may be allocated a mutex lock for its entire execution, yet only require it for a short duration.

Second, starvation is possible. A thread 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 thread.

NO PREEMPTION

The third necessary condition for deadlocks is that there is no preemption of resources that have already been allocated. To ensure that this condition does not hold, we can use the following protocol. If a thread is holding some resources and requests another resource that cannot be immediately allocated to it then all resources the thread is currently holding are preempted. In other words, these resources are implicitly released. The preempted resources are added to the list of resources forwhich the thread is waiting. The thread will be restarted only when it can regain its old resources, as well as the new ones that it is requesting.

CIRCULAR WAIT

The three options presented thus far for deadlock prevention are generally impractical in most situations. However, the fourth and final condition for deadlocks the circular-wait condition presents an opportunity for a practical solution by invalidating one of the necessary conditions.

To illustrate, we let R = {R1, R2, ..., Rm} 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. We can accomplish this

scheme in an application program by developing an ordering among all synchronization objects in the system.

DEADLOCK AVOIDANCE

Deadlock-prevention algorithms prevent deadlocks by limiting how requests can be made. The limits ensure that at least one of the necessary conditions for deadlock cannot occur. Possible side effects of preventing deadlocks by this method, however, are low device utilization and reduced system throughput.

An alternative method for avoiding deadlocks is to require additional information about how resources are to be requested. For example, in a system with resources R1 and R2, the system might need to know that thread P will request first R1 and then R2 before releasing both resources, whereas thread Q will request R2 and then R1.

With this knowledge of the complete sequence of requests and releases for each thread, the system can decide for each request whether or not the thread should wait in order to avoid a possible future deadlock. Each request requires that in making this decision the system consider the resources currently available, the resources currently allocated to each thread, and the future requests and releases of each thread.

RESOURCE-ALLOCATION GRAPH ALGORITHM

We use a variant of the resource-allocation graph for deadlock avoidance. In addition to the request and assignment edges, we introduce a new type of edge, called a **claim edge.** A claim edge Ti→Rj indicates that thread Ti may request resource Rj at some time in the future. This edge resembles a request edge in direction but is represented in the graph by a dashed line.

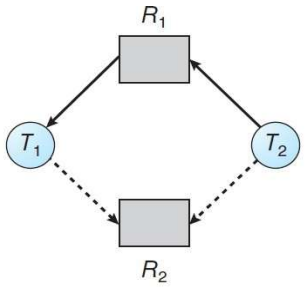

Figure Resource-allocation graph for deadlock avoidance.

When thread Ti requests resource Rj, the claim edge Ti → Rj is converted to a request edge. Similarly, when a resource Rj is released by Ti, the assignment edge Rj →Ti is reconverted to a claim edge Ti→Rj. Note that the resources must be claimed a priori in the system. That is, before thread Ti starts executing, all its claim edges must already appear in the resource-allocation graph.

Now suppose that thread T_i requests resource R_j. The request can be granted only if converting the request edge $T_i \rightarrow R_j$ to an assignment edge $R_j \rightarrow T_i$ does not result in the formation of a cycle in the resource- allocation graph. We check for safety by using a **cycle-detection algorithm.** An algorithm for detecting a cycle in this graph requires an order of n^2 operations, where n is the number of threads in the system. If no cycle exists, then the allocation of the resource will leave the system in a safe state. If a **cycle is found**, then the allocation will put the system in an **unsafe state**. In that case, thread T_i will have to wait for its requests to be satisfied.

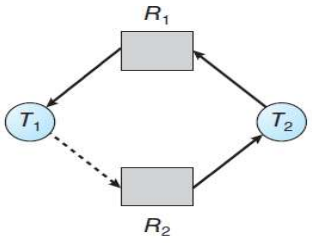

Figure An unsafe state in a resource-allocation graph.

To illustrate this algorithm, we consider the resource-allocation graph of figure. Suppose that T_2 requests R_2. Although R_2 is currently free, we cannot allocate it to T_2, since this action will create a cycle in the graph. A cycle, as mentioned, indicates that the system is in an unsafe state. If T_1 requests R_2, and T_2 requests R_1, then a deadlock will occur.

BANKER'S ALGORITHM

The resource-allocation-graph algorithm is not applicable to a resource allocation system with multiple instances of each resource type. **Banker's deadlock avoidance algorithm** is applicable to such a system but is less efficient than the resource-allocation graph scheme. The name was chosen because the algorithm could be used in a banking system to ensure that the bank never allocated its available cash in such a way that it could no longer satisfy the needs of all its customers.

When a new thread enters the system, it must declare the **maximum number of instances of each resource type that it may need.** This number may not exceed the total number of resources in the system. When a user requests a set of resources, the system must determine whether the allocation of these resources will leave the system in a safe state. If it will, the resources are allocated; otherwise, the thread must wait until some other thread releases enough resources.

We need the following data structures, where *n* is the number of threads in the system and *m* is the number of resource types:

Available

A vector of length *m* indicates the number of available resources of each type. If Available[j] equals k, then k instances of resource type R_j are available.

Max5

An n × m matrix defines the maximum demand of each thread. If Max[i][j] equals k, then thread T_i may request at most k instances of resource type R_j.

Allocation

An n × m matrix defines the number of resources of each type currently allocated to each thread. If Allocation[i] [j] equals k, then thread T_i is currently allocated k instances of resource type R_j.

Need

An n × m matrix indicates the remaining resource need of each thread. If Need[i][j] equals k, then thread T_i may need k more instances of resource type R_j to complete its task.

Need[i][j] = Max[i][j]− Allocation[i] [j]

Banker's algorithm comprises of two algorithms:

- **Safety Algorithm**
- **Resource Request Algorithm**

Safety algorithm

We can now present the algorithm for finding out whether or not a system

is in a safe state. This algorithm can be described as follows:

1. Let *Work* and *Finish* be vectors of length *m* and *n*, respectively. Initialize
   Work = *Available* and *Finish*[*i*] = *false* for *i* = 0, 1, ..., *n* − 1.
2. Find an index *i* such that both
   a. *Finish*[*i*] == *false*
   b. $Need_i \leq Work$

   If no such *i* exists, go to step 4.
3. $Work = Work + Allocation_i$
   Finish[*i*] = *true*
   Go to step 2.
4. If *Finish*[*i*] == *true* for all *i*, then the system is in a safe state.

Resource Request Algorithm

Next, we describe the algorithm for determining whether requests can be safely granted. Let $Request_i$ be the request vector for thread T_i. If $Request_i[j] == k$, then thread T_i wants k instances of resource type Rj.

When a request for resources is made by thread T_i the following actions are taken:

1. If $Request_i \leq Need_i$, go to step 2. Otherwise, raise an error condition, since the thread has exceeded its maximum claim.
2. If $Request_i \leq Available$, go to step 3. Otherwise, T_i must wait, since the resources are not available.
3. Have the system pretend to have allocated the requested resources to thread T_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 thread T_i is allocated its resources. However, if the new state is unsafe, then T_i must wait for $Request_i$, and the old resource allocation state is restored.

EXAMPLE:

Assume that there are 5 processes, P0 through P4, and 4 types of resources. At T0 we have the following system state:

Given Matrices												
	Allocation Matrix (N0 of the allocated resources By a process)				**Max Matrix** Max resources that may be used by a process				**Available Matrix** Not Allocated Resources			
	A	**B**	**C**	**D**	**A**	**B**	**C**	**D**	**A**	**B**	**C**	**D**
P_0	0	1	1	0	0	2	1	0	1	5	2	0
P_1	1	2	3	1	1	6	5	2				
P_2	1	3	6	5	2	3	6	6				
P_3	0	6	3	2	0	6	5	2				
P_4	0	0	1	4	0	6	5	6				
Total	**2**	**12**	**14**	**12**								

1. **Create the need matrix (max-allocation)**

Need(i) = Max(i) – Allocated(i)
(i=0) (0,2,1,0) - (0,1,1,0) = (0,1,0,0)
(i=1) (1,6,5,2) - (1,2,3,1) = (0,4,2,1)
(i=2) (2,3,6,6) - (1,3,6,5) = (1,0,0,1)
(i=3) (0,6,5,2) - (0,6,3,2) = (0,0,2,0)
(i=4) (0,6,5,6) - (0,0,1,4) = (0,6,4,2)
Ex. Process P1 has max of (1,6,5,2) and allocated by (1,2,3,1)
Need(p1) = max(p1)- allocated(p1) = (1,6,5,2) – (1,2,3,1) = (0,4,2,1)

Need Matrix = Max Matrix – Allocation Matrix				
	A	**B**	**C**	**D**
P_0	0	1	0	0
P_1	0	4	2	1
P_2	1	0	0	1
P_3	0	0	2	0
P_4	0	6	4	2

2. **Use the safety algorithm to test if the system is in a safe state or not?**

a.We will first define work and finish:

Initially work = available = (1, 5 , 2, 0)

Finish = False for all processes

Finish matrix	
P_0	False
P_1	False
P_2	False
P_3	False
P_4	False

Work vector			
1	**5**	**2**	**0**

b.Check the needs of each process [needs(P_i) <= Max(P_i)], if this condition is true:

- **Execute the process , Change Finish[i] =True**
- **Release the allocated Resources by this process**
- **Change The Work Variable = Allocated (pi) + Work**

- **need0 (0,1,0,0) <= work(1,5,2,0)**

- P0 will be executed because need(P0) <= Work
- P0 will be True
- P0 will release the allocated resources(0,1,1,0)
- Work = Work (1,5,2,0)+Allocated(P0) (0,1,1,0)
  = 1,6,3,0

Finish matrix	
P_0 – 1	**True**
P_1	False
P_2	False
P_3	False
P_4	False

Work vector			
1	**6**	**3**	**0**

- **Need1 (0,4,2,1) <= work(1,6,3,0) Condition Is False P1 will Not be executed**
- **Need2 (1,0,0,1) <= work(1,6,3,0) Condition Is False P2 will Not be executed**
- **Need3 (0,0,2,0) <= work(1,6,3,0) P3 will be executed**

- P3 will be executed because need(P3) <= Work
  - P3 will be True
  - P3 will release the allocated resources (0,6,3,2)
  - Work = Work (1,6,3,0)+Allocated(P3) (0,6,3,2)
    = 1,12,6,2

Finish matrix	
P_0 – 1	**True**
P1	False
P_2	False
P_3 -2	**True**
P_4	False

Work vector			
1	12	7	6

- **Need4 (0,6,4,2) <= work(1,12,6,2) P4 will be executed**
  - P4 will be executed because need(P4) <= Work
  - P4 will be True
  - P4 will release the allocated resources (0,0,1,4)
  - Work = Work (1,12,6,2) + Allocated(P4) (0,0,1,4)
    = 1,12,7,6

Finish matrix	
P_0 – 1	**True**
P1	False
P_2	False
P_3 -2	**True**
P_4 -3	**True**

Work vector			
1	12	7	6

- **Need1 (0,4,2,1) <= work(1,12,7,6) P1 will be executed**
  - P1 will be executed because need(P1) <= Work
  - P1 will be True
  - P1 will release the allocated resources (1,2,3,1)
  - Work = Work (1,12,7,6) + Allocated(P1) (1,2,3,1)
    = 2,14,10,7

Finish matrix	
P_0 – 1	**True**
P_1 -4	**True**
P_2	False
P_3 -2	**True**
P_4 -3	**True**

Work vector			
2	14	10	7

- **Need2 (1,0,0,1) <= work(2,14,10,7) P2 will be executed**
  - P2 will be executed because need(P2) <= Work
  - P2 will be True
  - P2 will release the allocated resources (1,3,6,5)
  - Work = Work (2,14,10,7) + Allocated(P1) (1,3,6,5)
    = 3,17,16,12

Finish matrix	
P_0 – 1	**True**
P_1 -4	**True**
P_2 -5	**True**
P_3 -2	**True**
P_4 -3	**True**

Work vector			
3	17	16	12

The system is in a safe state and the processes will be executed in the following order:

P0,P3,P4,P1,P2

DEADLOCK DETECTION

If a system does not employ either a **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.

At this point, however, a detection-and-recovery scheme requires

- **overhead** that includes not only the **run-time costs** of maintaining the necessary information
- **executing the detection algorithm** but also the potential losses inherent in **recovering from a deadlock**

Now, the main task of the OS is to detect the deadlocks and this is done with the help of Resource Allocation Graph.

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.** We obtain this graph from the resource- allocation graph by removing the resource nodes and collapsing the appropriate edges.

More precisely, an edge from T_i to T_j in a wait-for graph implies that thread T_i is waiting for thread T_j to release a resource that T_i needs. An edge $T_i \rightarrow T_j$ exists in a wait-for graph if and only if the corresponding resource-allocation graph contains two edges $T_i \rightarrow R_q$ and $R_q \rightarrow T_j$ for some resource R_q.

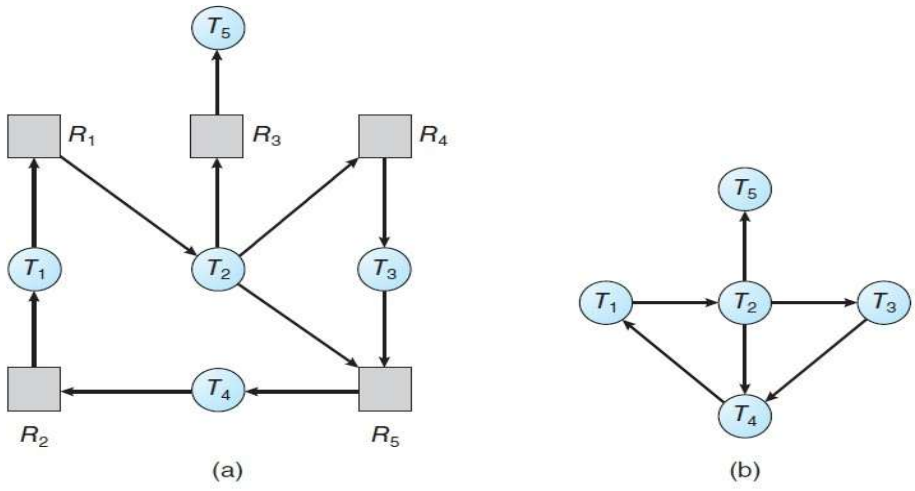

Figure (a) Resource-allocation graph. (b) Corresponding wait-for graph.

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 a graph requires $O(n^2)$ 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 a resource-allocation system with multiple instances of each resource type. We turn now to a deadlock detection algorithm that is applicable to such a system. This algorithm mainly uses several time-varying data structures that are similar to those used in Banker's Algorithm.

To simplify notation, we again treat the rows in the matrices Allocation and Request as vectors; we refer to them as $Allocation_i$ and $Request_i$. The detection algorithm described here simply investigates every possible allocation sequence for the threads that remain to be completed.

ALGORITHM

1. Let *Work* and *Finish* be vectors of length m and n, respectively. Initialize $Work = Available$. For $i = 0, 1, ..., n-1$, if $Allocation_i \neq 0$, then $Finish[i] = false$. Otherwise, $Finish[i] = true$.
2. Find an index i such that both
   a. $Finish[i] == false$
   b. $Request_i \leq Work$

   If no such i exists, go to step 4.
3. $Work = Work + Allocation_i$
   $Finish[i] = true$
   Go to step 2.
4. If $Finish[i] == false$ for some i, $0 \leq i < n$, then the system is in a deadlocked state. Moreover, if $Finish[i] == false$, then thread T_i is deadlocked.

RECOVERY FROM DEADLOCK

When a detection algorithm determines that a deadlock exists, several alternatives are available. One possibility is to inform the operator that a deadlock has occurred and to let the **operator deal with the deadlock manually.** Another possibility is to let the **system recover from the deadlock automatically.** There are two options for breaking a deadlock.

- **Process and Thread Termination**
- **Resource Preemption**

Process and Thread Termination

Simply abort one or more threads to break the circular wait. To eliminate deadlocks by aborting a process or thread, we use one of two methods.

- **Abort all deadlocked processes**
- **Abort one process at a time until the deadlock cycle is eliminated**

In both methods, the system reclaims all resources allocated to the terminated processes.

Abort one process at a time until the deadlock cycle is eliminated

This method incurs considerable overhead, since after each process is aborted, a deadlock-detection algorithm must be invoked to determine whether any processes are still deadlocked. Aborting a process may not be easy. If the partial termination method is used, then we must determine which deadlocked process (or processes) should be terminated. The question is basically an economic one; we should abort those processes whose termination will incur the minimum cost.

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 types of resources the process has used (for example, whether the resources are simple to preempt)?
4. How many more resources the process needs in order to complete?
5. How many processes will need to be terminated?

Resource Preemption

Preempt some resources from one or more of the deadlocked threads. To eliminate deadlocks using resource preemption, we successively 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, then three issues need to be addressed:

- **Selecting a victim**
- **Rollback**
- **Starvation**

Selecting a victim

Which resources and which processes are to be preempted? As in process termination, we must determine the order of preemption to minimize cost. Cost factors

may include such parameters as the **number of resources** a deadlocked process is holding and the **amount of time the process** has thus far consumed.

Rollback

If we preempt a resource from a process, what should be done with that process? Clearly, it cannot continue with its normal execution; it is missing some needed resource. We must roll back the process to some safe state and restart it from that state.

Starvation

How do we ensure that starvation will not occur? That is, how can we guarantee that resources will not always be preempted from the same process? 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. We must ensure that a process can be picked as a victim only a (small) **finite number of times.** The most common solution is to include the **number of rollbacks in the cost factor.**

END OF MODULE-3