UNIT-III
Inter Process Communication & Deadlocks

Introduction of Process Synchronization:

On the basis of synchronization, processes are categorized as one of the following two types:

- **Independent Process** : Execution of one process does not affects the execution of other processes.
- **Cooperative Process** : Execution of one process affects the execution of other processes.

- Process synchronization problem arises in the case of Cooperative process also because resources are shared in Cooperative processes.

Background:

- Processes can execute concurrently
  - May be interrupted at any time, partially completing execution
- Concurrent access to shared data may result in data inconsistency
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes
- Illustration of the problem:

Suppose that we wanted to provide a solution to the consumer-producer problem that fills ***all*** the buffers. We can do so by having an integer **counter** that keeps track of the number of full buffers. Initially, **counter** is set to 0. It is incremented by the producer after it produces a new buffer and is decremented by the consumer after it consumes a buffer.

Producer-Consumer Problem:

- In the producer-consumer problem, there is one Producer that is producing something and there is one Consumer that is consuming the products produced by the Producer. The producers and consumers share the same memory buffer that is of fixed-size.

- Paradigm for cooperating processes, *producer* process produces information that is consumed by a *consumer* process

  - **unbounded-buffer** places no practical limit on the size of the buffer
  - **bounded-buffer** assumes that there is a fixed buffer size

- The job of the Producer is to generate the data, put it into the buffer, and again start generating data. While the job of the Consumer is to consume the data from the buffer.

What's the problem here?

The following are the problems that might occur in the Producer-Consumer:

- The producer should produce data only when the buffer is not full. If the buffer is full, then the producer shouldn't be allowed to put any data into the buffer.
- The consumer should consume data only when the buffer is not empty. If the buffer is empty, then the consumer shouldn't be allowed to take any data from the buffer.
- The producer and consumer should not access the buffer at the same time.

Producer:

```
while (true) {
        /* produce an item in next produced */

        while (counter == BUFFER_SIZE) ;
                /* Stop-  do nothing */
        buffer[in] = next_produced;
        in = (in + 1) % BUFFER_SIZE;
        counter++;
}
```

Consumer:

```
while (true)
{
        while (counter == 0)
                ; /* do nothing */
        next_consumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
     counter--;
        /* consume the item in next consumed */
}
```

Producer consumer Problem:

- Illustration of the problem:
- Suppose that we wanted to provide a solution to the consumer-producer problem that fills all the buffers. We can do so by having an integer counter that keeps track of the number of full buffers. Initially, counter is set to 0. It is incremented by the producer after it produces a new buffer and is decremented by the consumer after it consumes a buffer.

Race conditions:

- A situation where 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 a Race condition.
- A race condition is a situation in which two or more threads or processes are reading or writing some shared data, and the final result depends on the timing of how the threads are scheduled. Race conditions can lead to unpredictable results and subtle program bugs.
- A race condition is a situation where two or more processes access shared data concurrently and final value of shared data depends on timing (race to access and modify data)
- Definition - To guard against the race condition, we need to ensure that only one process at a time can be manipulating the variable counter and this is referred as process synchronization.
- A race condition is a situation that may occur inside a critical section. This happens when the result of multiple thread execution in the critical section differs according to the order in which the threads execute. Race conditions in critical sections can be avoided if the critical section is treated as an atomic instruction. Also, proper thread synchronization using locks or atomic variables can prevent race conditions.

Example:

- **counter++** could be implemented as

  register1 = counter
  register1 = register1 + 1
  counter = register1

- **counter--** could be implemented as

  register2 = counter
  register2 = register2 - 1
  counter = register2

- Consider this execution interleaving with “count = 5” initially:

S0: producer execute **register1 = counter** {register1 = 5}
S1: producer execute **register1 = register1 + 1** {register1 = 6}
S2: consumer execute **register2 = counter** {register2 = 5}
S3: consumer execute **register2 = register2 – 1** {register2 = 4}
S4: producer execute **counter = register1** {counter = 6 }
S5: consumer execute **counter = register2** {counter = 4}

Critical Section Problem:

- Consider system of ***n*** processes {$p_0, p_1, \ldots p_{n-1}$}
- Each process has a segment of code, called a **critical section (CS**), in which the process may be changing common variables, updating a table, writing a file, and so on.
- The important feature of the system is that, when one process is executing in its critical section, no other process is to be allowed to execute in its critical section.
- That means no two processes are executing in their critical sections at the same time.
- ***The Critical section problem*** is to design protocol to solve this and the processes can use to cooperate.
- Each process must ask permission to enter critical section in **entry section**, may follow critical section with **exit section**. The remaining code is the remainder section.

General structure of a typical process Pi :

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

- Every process have the sections . The structure consists of four sections
  - Entry section
  - Critical section
  - Exit section
  - Remainder section
- Each process must request to enter into critical section. That code is implemented in entry section.

Solution to Critical-Section Problem:

A Solution to the Critical-section problem must satisfy the following three 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 the selection of the processes that will enter the critical section next cannot be postponed indefinitely

Then only those processes that are not executing in their remainder sections can participate in deciding which will enter its critical section next,

3. Bounded Waiting - A bound or limit must 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

- There is a race to execute critical sections
- Assume that each process executes at a nonzero speed
- No assumption concerning relative speed of the N processes

- At any given point of time, several Kernel-mode processes may be active in the operating system.
- As a result, the code implementing the operating system is subject to several race conditions.
- For example: if two processes were to open files simultaneously, the separate updates could result in a race condition.
- Two general approaches are used to handle critical sections in OS: depending on preemptive kernel or non-preemptive kernel.
  - **Preemptive** kernel– it allows preemption of process when running in kernel mode
  - **Non-preemptive kernel** – it does not allow a process running in kernel mode to preempted. A kernel mode process runs until exits kernel mode, blocks, or voluntarily yields control of the CPU.
    - Essentially free of race conditions in kernel mode

Peterson's Solution:

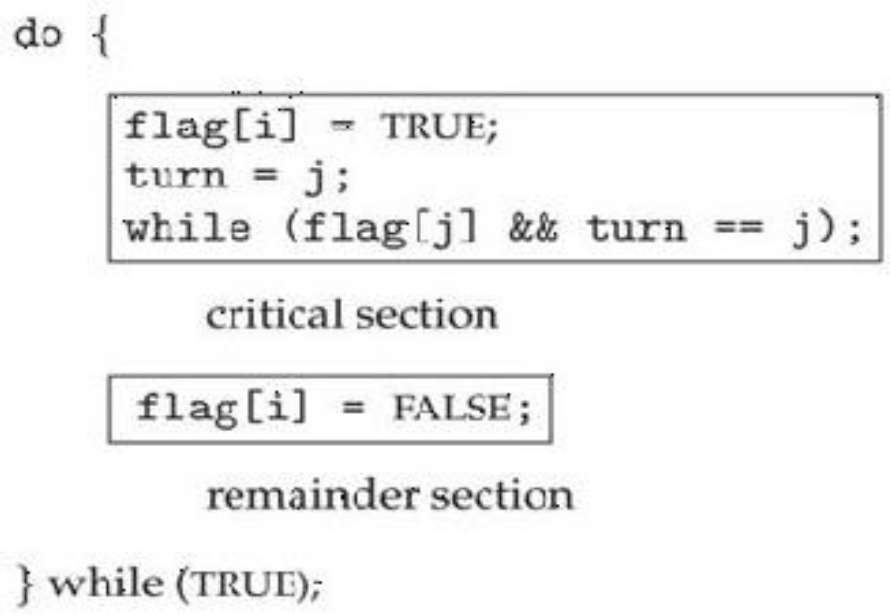
A classic software-based solution to the critical-section problem known as Peterson's solution.

It Does not require strict alternation.

- Good algorithmic description of solving the problem
- It requires Two processes that alternate execution between their critical section and remainder section.
- Assume that the **load** and **store** machine-language instructions are atomic; that is, cannot be interrupted
- Peterson's solution requires two data items to be shared between the two processes. The two processes share two variables:
  - **int turn;**
  - **Boolean flag[2]**
  - The variable **turn** indicates whose turn it is to enter the critical section
- The **flag** array is used to indicate if a process is ready to enter the critical section. If **flag[i] = *true*** implies that process $\mathbf{P_i}$ is ready to enter critical section.

Algorithm for Process Pi

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



Peterson's solution is restricted to two processes that alternate execution between their critical sections and remainder sections. The processes are numbered P0 and P1.

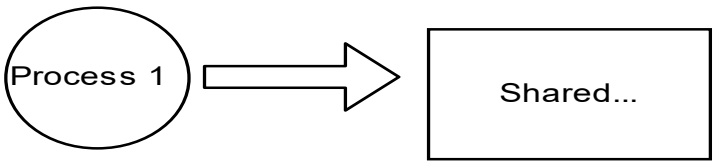
- For convenience, when presenting Pi, we use Pj to denote the other process; that is, j equals 1 − i. Peterson's solution requires the two processes to share two data items:
    - int turn;
    - boolean flag[2];
- The variable turn indicates whose turn it is to enter its critical section. That is, if turn == i, then process Pi is allowed to execute in its critical section. The flag array is used to indicate if a process is ready to enter its critical section.
- For example, if flag[i] is true, this value indicates that Pi is ready to enter its critical section. With an explanation of these data structures complete.
- To enter the critical section, process Pi first sets flag[i] to be true and then sets turn to the value j, thereby asserting that if the other process wishes to enter the critical section, it can do so.
- If both processes try to enter at the same time, turn will be set to both i and j at roughly the same time.
- Only one of these assignments will last; the other will occur but will be overwritten immediately.

- The eventual value of turn determines which of the two processes is allowed to enter its critical section first.
- We now prove that this solution is correct. We need to show that:
    - 1. Mutual exclusion is preserved.
    - 2. The progress requirement is satisfied.
    - 3. The bounded-waiting requirement is met
    - To prove property 1, we note that each Pi enters its critical section only if either flag[j] == false or turn == i.
    - Also note that, if both processes can be executing in their critical sections at the same time, then flag[0] == flag[1] == true.
    - These two observations imply that P0 and P1 could not have successfully executed their while statements at about the same time.

Mutual Exclusion with Busy waiting:

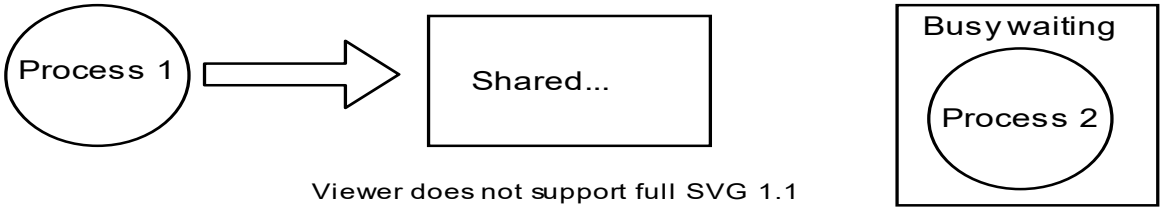
- Busy waiting, also known as spinning, or busy looping is a process synchronization technique in which a process/task waits and constantly checks for a condition to be satisfied before proceeding with its execution.
- In busy waiting, a process executes instructions that test for the entry condition to be true, such as the availability of a lock or resource in the computer system.
- For resource availability, consider a scenario where a process needs a resource for a specific program.
- However, the resource is currently in use and unavailable at the moment, therefore the process has to wait for resource availability before it can continue. This is what is known as busy waiting as illustrated below:

1. Process 1 is using the shared resource.



2. Process 2 is now in need of the shared resources.

3. Process 2 enters busy waiting till it has access to the shared...



Viewer does not support full SVG 1.1

Sleep and Wakeup:

- Busy waiting can be wasteful. Processes waiting to enter their critical sections waste processor time checking to see if they can proceed.
- A better solution to the mutual exclusion problem, which can be implemented with the addition of some new primitives, would be to **block** processes when they are denied access to their critical sections. Two primitives, **Sleep** and **Wakeup**, are often used to implement blocking in mutual exclusion.
- **How do Sleep and Wakeup Work?**
- Essentially, when a process is not permitted to access its critical section, it uses a system call known as **Sleep**, which causes that process to suspended.
- The process will not be scheduled to run again, until another process uses the **Wakeup** system call. In most cases, **Wakeup** is called by a process when it leaves its critical section if any other processes have blocked.
- The suspended process is woken by other process calling wakeup().

Synchronization Hardware:

- Process Synchronization refers to coordinating the execution of processes so that no two processes can have access to the same shared data and resources. A problem occurs when two processes running simultaneously share the same data or variable.
- There are three hardware approaches to solve process synchronization problems:

1. Swap
2. Test() and Set()
3. Unlock and lock

- Many systems provide hardware support for implementing the critical section code.
- All solutions below based on idea of **locking**
  - Protecting critical regions via locks
- Uniprocessors – could disable interrupts
  - Currently running code would execute without preemption
  - Generally too inefficient on multiprocessor systems
- Modern machines provide special atomic hardware instructions
  - Either test memory word and set value
  - Or swap contents of two memory words

Solution to Critical-section Problem Using Locks:

```
do {
    acquire lock
        critical section
```

```
    release lock
        remainder section
} while (TRUE);
```

Test and set Instruction:

Test and Set

In Test and Set the shared variable is a lock that is initialized to false.
The algorithm returns whatever value is sent to it and sets the lock to true. Mutual exclusion is ensured here, as till the lock is set to true, other processes will not be able to enter and the loop continues.
However, after one process is completed any other process can go in as no queue is maintained.

Definition:

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

1. Executed atomically
2. Returns the original value of passed parameter
3. Set the new value of passed parameter to "TRUE".
4. Shared Boolean variable lock, initialized to FALSE

Solution using test and set():

Solution:

```
do {
  while (test_and_set(&lock))
    ; /* do nothing */
       /* critical section */
  lock = false;
       /* remainder section */
} while (true);
```

Swap Instruction

In this algorithm, instead of directly setting the lock to true, the key is first set to true and then swapped with the lock.
Similar to Test and Set, when there are no processes in the critical section, the lock turns to false and allows other processes to enter. Hence, mutual exclusion and progress are ensured but the bound waiting is not ensured for the very same reason.

Definition:

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

Solution using Swap:

Shared Boolean variable lock initialized to FALSE; Each process has a local Boolean variable key

Solution:

```
do {
        key = TRUE;
        while ( key == TRUE)
        Swap (&lock, &key );
        // critical section
        lock = FALSE;
        // remainder section
} while (TRUE);
```

Bounded-waiting Mutual Exclusion with Test and Set():

```
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);
```

Mutex Locks:

- Operating-systems designers build software tools to solve the critical-section problem. The simplest of these tools is the Mutex lock.
- We use the mutex lock to protect critical regions and thus prevent race conditions.

- That is, a process must acquire the lock before entering a critical section; it releases the lock when it exits the critical section.
- The acquire()function acquires the lock, and the release() function releases the lock.
- The other solutions for critical section problem are complicated and generally inaccessible to application programmers
- Protect a critical section by first **acquire()** a lock then **release()** the lock
  - Boolean variable indicating if lock is available or not
- Calls to **acquire()** and **release()** must be atomic
  - Usually implemented via hardware atomic instructions
- But this solution requires **busy waiting**
  - This lock therefore called a **spinlock**

Solution to the critical-section problem using mutex locks.

```
do {
        acquire lock
                critical section
        release lock
                remainder section
} while (true);
```

- A mutex lock has a boolean variable available whosevalue indicates if the lock is available or not.
- If the lock is available, a call to acquire() succeeds, and the lock is then considered unavailable.
- A process that attempts to acquire an unavailable lock is blocked until the lock is released.
- The definition of acquire() is as follows:

```
acquire()
{
while (!available); /* busy wait */
available = false;;
}
```

- The definition of release() is as follows:

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

- Calls to either acquire() or release() must be performed atomically. Thus, mutex locks are often implemented using one of the hardware mechanisms.

Disadvantage of the implementation given here is that it requires busy waiting.

- While a process is in its critical section, any other process that tries to enter its critical section must loop continuously in the call to acquire().
- In fact, this type of mutex lock is also called a spinlock because the process “spins” while waiting for the lock to become available.

- This continual looping is clearly a problem in a real multiprogramming system, where a single CPU is shared among many processes. Busy waiting wastes CPU cycles that some other process might be able to use productively.

Semaphore:

- A semaphore S is an integer variable that, apart from initialization, is accessed only through two standard atomic operations: wait() and signal().
- It is a Synchronization tool that provides more sophisticated ways (than Mutex locks) for process to synchronize their activities.
- Semaphore is introduced by Dijkstra.
- The wait() operation was originally termed P (from the Dutch proberen, "to test"); signal() was originally called V (from verhogen, "to increment").
- The two indivisible (atomic) operations -**wait() and signal()**
  Which are Originally called **P()** and **V()**

Wait( ) and Signal():

The value of a semaphore is modified by wait() or signal() operation where the wait() operation decrements the value of semaphore and the signal() operation increments the value of the semaphore

The wait() operation is performed when the process wants to access the resources and the signal() operation is performed when the process want to release the resources. The semaphore can be binary semaphore or the counting semaphore.

Definition of the wait() operation

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

Definition of the signal() operation

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

Uses of Semaphores:

- Counting semaphore – integer value can range over an unrestricted domain
- Binary semaphore – integer value can range only between 0 and 1; can be simpler to implement. Also known as mutex locks.
- Can implement a counting semaphore S as a binary semaphore

- Semaphore as General Synchronization Tool
- Usually two types of semaphores
  - Counting semaphore
  - Binary semaphore (also called mutex locks)
- Binary semaphores can be used for:
  - The binary semaphore is a integer value can range only between 0 and 1
  - On some systems binary semaphores are called, mutex locks, as they are locks that provide mutual exclusion.
  - We can use binary semaphores to Critical section problems for multiple processes
- Counting semaphores can be used for:
  - The value of counting semaphore is integer value can range over an unrestricted domain
  - They are used to Controlling access to multiple, but finite set of resources
  - Ordering the running of critical sections

Binary Semaphores:

- The value of a binary semaphore can range only between 0 and 1. Binary semaphores behave similarly to mutex locks. On systems that do not provide mutex locks, binary semaphores can be used instead for providing mutual exclusion.
- We can use binary semaphores to deal with critical section problem for multiple processes
- The n Processes share a semaphore, mutex, initialized to 1.
- Each process is organized as

```
do
 {
        wait(mutex);
              // Critical section
         signal(mutex);
                // remainder section
  }while(true);
```

Counting Semaphores:

- Counting semaphores can be used to control access to a given resource consisting of a finite number o£ instances.
- 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.
- We can also use semaphores to solve various synchronization problems.
- For example, consider two concurrently running processes: P1 with a statement S1 and P2 with a statement S2
- Consider P_1 and P_2 that require statements S_1 to happen before S_2. Create a semaphore "**synch**" initialized to 0 . In process P1, we insert the statements

P1:
S_1;
signal(synch);
P2:
wait(synch);
S_2;

Because synch is initialized to 0, P2 will execute S2 only after P1 has invoked signal(synch), which is after statement S1 has been executed.

- Can implement a counting semaphore ***S*** as a binary semaphore

Semaphore Implementation:

- The main disadvantage of the semaphore definition given here is that it requires busy waiting.
- While a process is in its critical section, any other process that tries to enter its critical section must loop continuously in the entry code. This continual looping is clearly a problem in a real multiprogramming system, where a single CPU is shared among many processes.
- Busy waiting wastes CPU cycles that some other process might be able to use productively. This type of semaphore is also called a because the process "spins" while waiting for the lock

Semaphore Implementation with no Busy waiting:

- To overcome the need for busy waiting, we can modify the definition of the wait() and signal() semaphore operations. When a process executes the wait () operation and finds that the semaphore value is not positive, it must wait. However, rather than engaging in busy waiting, the process can block itself.
- The each semaphore there is an associated waiting queue
- Each entry in a waiting queue has two data items:
    - value (of type integer)
    - pointer to next record in the list
- There are Two operations:
- **block** – place the process invoking the operation on the appropriate waiting queue
- **wakeup** – remove one of processes in the waiting queue and place it in the ready queue

- The block operation places a process into a waiting queue associated with the semaphore, and the state of the process is switched to the waiting state. Then control is transferred to the CPU scheduler, which selects another process to execute.
- A process that is blocked, waiting on a semaphore S, should be restarted when some other process executes a signal() operation.
- The process is restarted by a wakeup() operation, which changes the process from the waiting state to the ready state.
- The process is then placed in the ready queue. (The CPU may or may not be switched from the running process to the newly ready process, depending on the CPU-scheduling algorithm.)
- To implement semaphores under this definition, we define a semaphore as follows:

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

When a process must wait on a semaphore, it is added to the list of processes.

- A signal() operation removes one process from the list of waiting processes and awakens that process.
- The wait() semaphore operation can be defined as

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

- The signal() semaphore operation can be defined as

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

- The block() operation suspends the process that invokes it.
- The wakeup(P) operation resumes the execution of a blocked process P

- These two operations are provided by the operating system as basic system calls.

- When a process executes the wait () operation and finds that the semaphore value is not positive, it must wait. However, rather than engaging in busy waiting, the process can block itself.
- The block operation places a process into a waiting queue associated with the semaphore, and the state of the process is switched to the waiting state. Then control is transferred to the CPU scheduler, which selects another process to execute.
- A process that is blocked, waiting on a semaphore S, should be restarted when some other process executes a signal() operation.
- The process is restarted by a wakeup () operation, which changes the process from the waiting state to the ready state. The process is then placed in the ready queue.

Deadlock and Starvation:

Deadlock – is a situation two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes

- The implementation of a semaphore with a waiting queue may result in a situation where two or more processes are waiting indefinitely for an event that can be caused only by one of the waiting processes.
- When such a state is reached, these processes are said to be deadlocked
- To illustrate this, consider a system consisting of two processes, P0 and P1, each accessing two semaphores, S and Q, set to the value 1:

P_0	P_1
wait(S);	**wait(Q);**
wait(Q);	**wait(S);**
...	**...**
…	**….**
signal(S);	**signal(Q);**
signal(Q);	**signal(S);**

- Suppose that Po executes wait (S) and then P1 executes wait (Q). When Po executes wait (Q), it must wait until P1 executes signal (Q). Similarly, when P1 executes wait (S), it must wait until Po executes signal(S). Since these signal() operations cannot be executed, Po and P1 are deadlocked.

- We say that a set of processes is in a deadlock state when every process in the set is waiting for an event that can be caused only by another process in the set.

- Another problem related to deadlocks is indefinite blocking or starvation,

- **Starvation** – **indefinite blocking** :It is a situation in which processes wait indefinitely within the semaphore. In this, a process may never be removed from the semaphore queue in which it is suspended
- **Priority Inversion** – Scheduling problem when lower-priority process holds a lock needed by higher-priority process
- Solved via priority-inheritance protocol
- **Indefinite blocking-** may occur if we remove processes from the list associated with a semaphore in LIFO (last-in, first-out) order.

Advantages and Disadvantages:

Advantages of Semaphore :

1. Semaphore resolves the process synchronization issues.
2. Waiting list associated with each semaphore avoids busy waiting and lets CPU perform other productively.
3. It is machine-independent.
4. It allows multiple threads to access the critical section.

Disadvantages of Semaphore

1. Implementing semaphore can lead to priority inversion where the two processes get into spinlock condition.
2. If not implemented properly then semaphore can cause deadlock condition.
3. Semaphore requires busy waiting and it wastes CPU cycles.

Classical Problems of Synchronization:

- Classical problems of Synchronization are:
  - Bounded-Buffer Problem
  - Readers and Writers Problem
  - Dining-Philosophers Problem

The Bounded-Buffer Problem :

- Bounded-Buffer Problem is also called as Producer consumer problem.

- We assume that 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 the value 0.
- We can interpret this code as the producer producing full buffers for the consumer or as the consumer producing empty buffers for the producer.

- N buffers, each can hold one item
- Semaphore mutex initialized to the value 1
- Semaphore full initialized to the value 0
- Semaphore empty initialized to the value N.

- We assume that the pool consists of n buffers, each capable of holding one item.
- There are two processes running –Producer and Consumer
- The producer produces the items and consumer consumes the items.

The problem arises when

- The producer must not insert data when a buffer is full
- The consumer should not remove data when buffer is empty.
- The producer and consumer should not insert and remove data at same time.

The structure of the producer process. :

```
do {
        . . .
        /* produce an item in next produced */
        . . .
        wait(empty);
        wait(mutex);
        . . .
        /* add next produced to the buffer */
        . . .
        signal(mutex);
        signal(full);
} while (true);
```

The structure of the consumer process:

```
do {
        wait(full);
        wait(mutex);
        . . .
        /* remove an item from buffer to next consumed */
        . . .
```

```
        signal(mutex);
        signal(empty);
        . . .
        /* consume the item in next consumed */
        . . .
} while (true);
```

- We can interpret this code as the producer producing full buffers for the consumer or as the consumer producing empty buffers for the producer.

The Readers–Writers Problem

- A database is to be shared among several concurrent processes.
- Some of these processes may want only to read the database (readers), whereas others may want to update (that is, to read and write) the database (writers).
- If two readers access the shared data simultaneously, no adverse affectswill result. However, if a writer and some other thread (either a reader or a writer) access the database simultaneously, there could be some synchronization issues.
- To ensure that these difficulties do not arise, we require that the writers have exclusive access to the shared database. This synchronization problem is referred to as the readers-writers problem.
- A data set is shared among a number of concurrent processes
    - Readers – only read the data set; they do ***not*** perform any updates
    - Writers – can both read and write
- Problem – allow multiple readers to read at the same time
- Only one single writer can access the shared data at the same time
- Several variations of how readers and writers are considered – all involve some form of priorities
- Here we use semaphore
- The readers-writers problem has several variations, all involving priorities.
- The simplest one, referred to as the **first readers-writers problem**, requires that no reader will be kept waiting unless a writer has already obtained permission to use the shared object. In other words, no reader should wait for other readers to finish simply because a writer is waiting.
- The **second readers-writers problem** requires that, once a writer is ready, that writer performs its write as soon as possible. In other words, if a writer is waiting to access the object, no new readers may start reading.
- A solution to either problem may result in starvation.
- In the first case, writers may starve. In the second case, readers may starve.
- In the solution to the first readers–writers problem, the reader processes share the following data structures:
- Shared Data
    - Data set
    - Semaphore **wrt** initialized to 1 -common to R and W

- Semaphore **mutex** initialized to 1 -to ensure mutual exclusion
- Integer **read_count** initialized to 0 -when any Reader enters or exits CS

- **The structure of a writer process**

```
do {
  wait(wrt);
     ...
  /* writing is performed */
     ...
  signal(wrt);
} while (true);
```

- **The structure of a reader process:**

```
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);
```

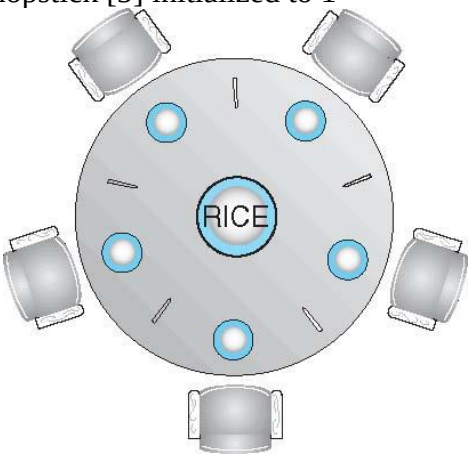
Reader-writer locks are most useful in the following situations:

- In applications where it is easy to identify which processes only read shared data and which processes only write shared data.
- In applications that have more readers than writers. This is because reader-writer locks generally require more overhead to establish than semaphores or mutual-exclusion locks. The increased concurrency of allowing multiple readers compensates for the overhead involved in setting up the reader-writer lock.

Dining-Philosophers Problem:

- The dining philosophers problem is useful for modeling processes that are competing for exclusive access to a limited number of resources, such as I/O devices.
- Consider five philosophers who spend their lives thinking and eating.
- Philosophers spend their lives alternating 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
- Don't interact with their neighbors, occasionally try to pick up 2 chopsticks (one at a time) to eat from bowl
    - Need both to eat, then release both when done
- In the case of 5 philosophers
- Shared data is
    - Bowl of rice (data set)
    - Semaphore chopstick [5] initialized to 1



The Dining Philosopher Problem –

- The Dining Philosopher Problem states that K philosophers seated around a circular table with one chopstick between each pair of philosophers.
- There is one chopstick between each philosopher.
- A philosopher may eat only if he can pick up the two chopsticks adjacent to him.
- One chopstick may be picked up by any one of its adjacent followers but not both.
- The dining philosopher is a classic synchronization problem as it demonstrates a large class of concurrency control problems.
- This dining philosopher problem is main example for Resource allocation.

Solution of Dining Philosophers Problem

- One simple solution is to represent each chopstick with a semaphore.
- A solution of the Dining Philosophers Problem is to use a semaphore to represent a chopstick. A chopstick can be picked up by executing a wait() operation on the semaphore and released by executing a signal() operation on the appropriate semaphores
- The structure of the chopstick is shown below –

```
semaphore chopstick [5];
```

• Thus, the shared data are semaphore chopstick[5] ; where all the elements of chopstick are initialized to 1

- The structure of Philosopher *i* :

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

- Semaphore provides a solution to Dining philosophers problem.
- But there exist some situation which can cause deadlock.
- Suppose that all five philosophers become hungry simultaneously 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. Several possible remedies to the deadlock problem are listed next.
- Deadlock handling
- Several possible remedies to the deadlock problem are replaced by:
  - Allow at most 4 philosophers to be sitting simultaneously at the table.
  - Allow a philosopher to pick up her chopsticks only if both chopsticks are available (to do this, she must pick them up in a critical section).
  - Use an asymmetric solution - that is an odd-numbered philosopher picks up first the left chopstick and then the right chopstick. Similarly Even-numbered philosopher picks up first the right chopstick and then the left chopstick.

Problems with Semaphores:

- Incorrect use of semaphore operations:
  - signal (mutex) …. wait (mutex)
- Deadlock and starvation are possible.

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, since these errors happen only if some particular execution sequences take place and these sequences do not always occur.

- A monitor is a high-level abstraction that provides a convenient and effective mechanism for process synchronization
- It is a *Abstract data type*, internal variables only accessible by code within the procedure
- Only one process may be active within the monitor at a time
- But not powerful enough to model some synchronization schemes
- Monitors are used for process synchronization. With the help of programming languages, we can use a monitor to achieve mutual exclusion among the processes. **Example of monitors:** ***Java Synchronized methods such as Java offers notify() and wait() constructs***
- In other words, monitors are defined as the construct of programming language, which helps in controlling shared data access.
- The Monitor is a module or package which encapsulates shared data structure, procedures, and the synchronization between the concurrent procedure invocations.

Characteristics of Monitors:

1. Inside the monitors, we can only execute one process at a time.
2. Monitors are the group of procedures, and condition variables that are merged together in a special type of module.

3. If the process is running outside the monitor, then it cannot access the monitor's internal variable. But a process can call the procedures of the monitor.
4. Monitors offer high-level of synchronization
5. Monitors were derived to simplify the complexity of synchronization problems.
6. There is only one process that can be active at a time inside the monitor.

- Monitor is a abstract data type- or ADT- encapsulates private data with public methods to operate on that data.
- A monitor type is an ADT which presents a set of programmer-defined operations that are provided mutual exclusion within the monitor.
- The monitor type also contains the declaration of variables whose values define the state of an instance of that type, along with the bodies of procedures or functions that operate on those variables
- The representation of a monitor type cannot be used directly by the various processes.

- Thus, a procedure 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 procedures.

Syntax of a monitor:

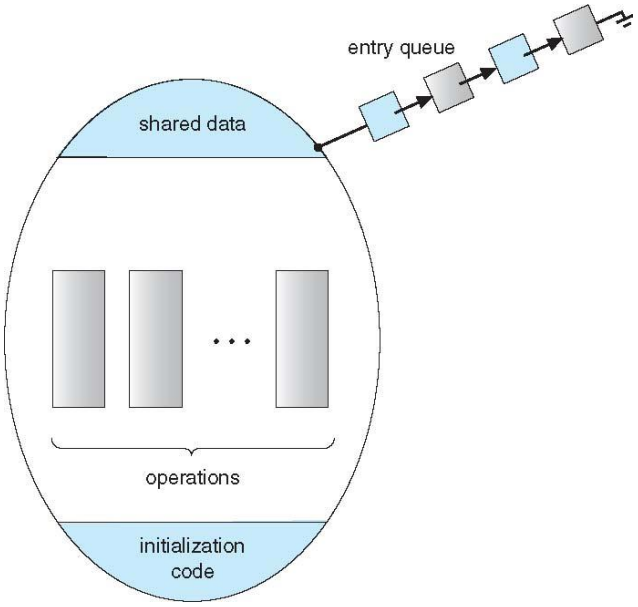
```
monitor monitor-name
{
        // shared variable declarations
        procedure P1 (…) { …. }
        ……
        procedure Pn (…) {……}
        Initialization code (…) { … }
        }
}
```

There are four main components of the monitor:

1. Initialization
2. Private data
3. Monitor procedure
4. Monitor entry queue

Initialization: – Initialization comprises the code, and when the monitors are created, we use this code exactly once.
Private Data: – Private data is another component of the monitor. It comprises all the private data, and the private data contains private procedures that can only be used within the monitor. So, outside the monitor, private data is not visible.
Monitor Procedure: – Monitors Procedures are those procedures that can be called from outside the monitor.
Monitor Entry Queue: – Monitor entry queue is another essential component of the monitor that includes all the threads, which are called procedures.

Schematic view of a Monitor:



The monitor construct is not sufficiently powerful for modeling some synchronization schemes. For this purpose, we need to define additional synchronization mechanisms. These mechanisms are provided by the condition construct.

Monitor with Condition Variables:

A programmer who needs to write a synchronization scheme can define one or more variables of type condition:

- condition x, y;
- There are only two operations that can be invoked on a condition variable are wait() and signal(). The operation

Wait operation:

x.wait() :-

The x.wait() operation means that the process invoking this operation is suspended until another process invokes

The process that performs wait operation on the condition variables are suspended and locate the suspended process in a block queue of that condition variable.

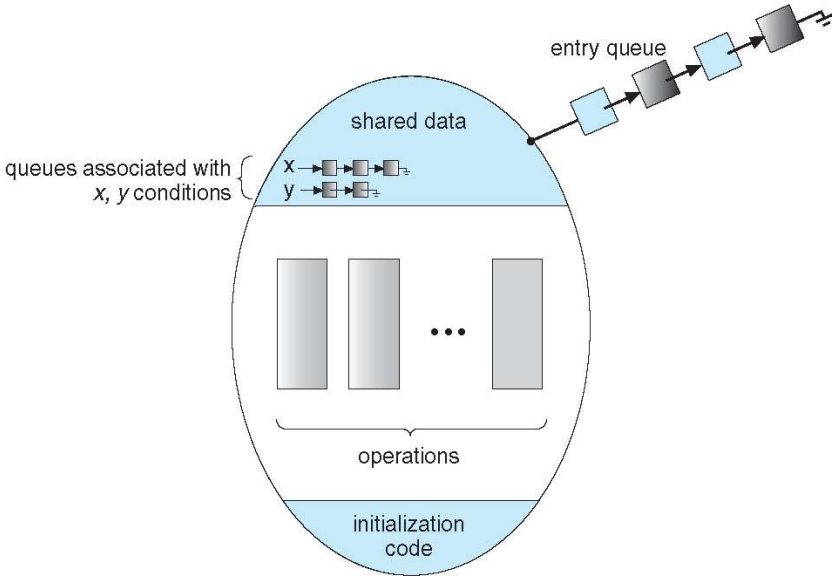
Signal Operation

x.signal() : –

The x.signal() operation resumes exactly one suspended process.

If a signal operation is performed by the process on the condition variable, then a chance is provided to one of the blocked processes.

Monitor with condition variables Diagram:



Condition Variables Choices:

- If process P invokes **x.signal(),** and process Q is suspended in **x.wait()**, what should happen next?
  - Both Q and P cannot execute in parallel. If Q is resumed, then P must wait
- Options include
  - **Signal and wait** – P waits until Q either leaves the monitor or it waits for another condition
  - **Signal and continue** – Q waits until P either leaves the monitor or it waits for another condition
  - Both have pros and cons – language implementer can decide
  - Monitors implemented in Concurrent Pascal compromise
    - P executing signal immediately leaves the monitor, Q is resumed
  - Implemented in other languages including Mesa, C#, Java

Monitor Solution to Dining Philosophers:

- We can use monitor concepts for providing a deadlock-free solution to the dining-philosophers problem.
- This solution imposes the restriction that a philosopher may pick up her chopsticks only if both of them are available. To code this solution, we need to distinguish among three states in which we may find a philosopher. For this purpose, we introduce the following data structure:

  enum {THINKING, HUNGRY, EATING} state[5];
- Philosopher i can set the variable state [i] = EATING only if her two neighbors are not eating:

  (state [(i +4) % 5] ! = EATING) and (state [(i +1) % 5] != EATING).
- We also need to declare

  condition self[5];

in which philosopher i can delay herself when she is hungry but is unable to obtain the chopsticks she needs.

- Each philosopher, before starting to eat, must invoke the operation pickup(). This act may result in the suspension of the philosopher process. After the successful completion of the operation, the philosopher may eat

A monitor solution to the dining-philosopher problem.

```
monitor DiningPhilosophers
{
        enum { THINKING, HUNGRY, EATING} state [5] ;
        condition self [5];
        void pickup (int i) {
              state[i] = HUNGRY;
              test(i);
              if (state[i] != EATING) self[i].wait;
}

  void putdown (int i) {
              state[i] = THINKING;
             // test left and right neighbors
              test((i + 4) % 5);
              test((i + 1) % 5);
}
void test (int i) {
              if ((state[(i + 4) % 5] != EATING) &&
              (state[i] == HUNGRY) &&
              (state[(i + 1) % 5] != EATING) ) {
                  state[i] = EATING ;
                     self[i].signal () ;
              }
  }

     initialization_code() {
              for (int i = 0; i < 5; i++)
              state[i] = THINKING;
            }
}
```

- Each philosopher *i* invokes the operations **pickup()** and **putdown()** in the following sequence:

```
DiningPhilosophers.pickup(i);
      EAT
DiningPhilosophers.putdown(i);
```

- No deadlock, but starvation is possible

Implementing a Monitor Using Semaphores

Now consider a possible implementation of the monitor mechanism using semaphores. For each monitor, a semaphore mutex (initialized to 1) is provided.

- A process must execute wait(mutex) before entering the monitor and must execute signal(mutex) after leaving the monitor.

- Since a signaling process must wait until the resumed process either leaves or waits, an additional semaphore, next, is introduced, initialized to 0.
- An integer variable
- next count is also provided to count the number of processes suspended on next
- Thus, each external procedure F is replaced by

```
wait(mutex);
body of  F
if (next_count > 0)
signal(next);
else
signal(mutex);
```

Mutual exclusion within a monitor is ensured. We can now describe how condition variables are implemented as well. For each condition x, we introduce a semaphore x_sem and an integer variable x_count, both initialized to 0.

- The operation x. wait() can now be implemented as

```
x_count++;
if (next_count > 0)
signal(next);
else
signal(mutex);
wait (x_sem) ;
x_count--;
```

- The operation x. signal() can be implemented as

```
if (x_count > 0) {
next_count++;
signal(x_sem);
wait(next);
next_count--;
}
```

Dead locks

Deadlocks:

Deadlock is a situation where a set of processes are blocked because each process is holding a resource and waiting for another resource acquired by some other process.

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

Consider an example when two trains are coming toward each other on the same track and there is only one track, none of the trains can move once they are in front of each other. A similar situation occurs in operating systems when there are two or more processes that hold some resources and wait for resources held by other(s). For example, in the below diagram, Process 1 is holding Resource 1 and waiting for resource 2 which is acquired by process 2, and process 2 is waiting for resource 1.

Under the normal mode of operation, a process may utilize a resource in only the following sequence:

1.Request:
2. Use:
3. Release.

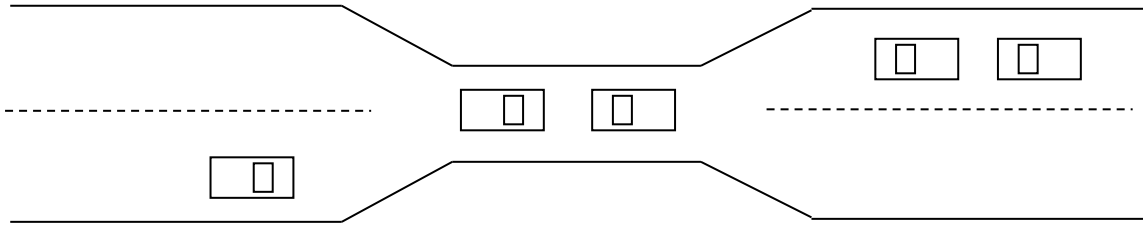
System Model:

- A system consists of a finite number of resources to be distributed among a number of competing processes.
- Resource types $R_1, R_2, \ldots, R_m$
- Memory space, CPU cycles, files, and I/0 devices (such as printers and DVD drives) are examples of resource types. If a system has two CPUs, then the resource type CPU has two instances. Similarly, the resource type printer may have five instances.
- Each resource type R_i has W_i instances.
- Each process utilizes a resource as follows:
- A process must request a resource before using it and must release the resource after using it. The following sequence of operations takes place:
  - **request**
  - **use**
  - **release**

Under the normal mode of operation ,A process may utilize a resource in only the following sequence:

1. Request: The process requests the resource. If the request cannot be granted immediately (for example, if the resource is being used by another process), then the requesting process must wait until it can acquire the resource.
2. Use: The process can operate on the resource (for example, if the resource is a printer, the process can print on the printer).
3. Release: The process releases the resource.

Bridge Crossing Example:



- Traffic only in one direction
- Each section of a bridge can be viewed as a resource
- If a deadlock occurs, it can be resolved if one car backs up (preempt resources and rollback)
- Several cars may have to be backed up if a deadlock occurs
- Starvation is possible

Deadlock Characterization:

- In a deadlock, processes never finish executing, and system resources are tied up, preventing other jobs from starting.

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

1. **Mutual exclusion:**

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

2. Hold and wait:
There must exist a process that is holding at least one resource and waiting to acquire additional resources that are currently being held by other processes.

3. No Preemption:-
Resources cannot be preempted, that is a resource can be released only voluntarily by the process holding it, after that process has completed its task.

4) Circular Wait:-

A set $\{P_0, P_1, \ldots, P_n\}$ of waiting processes must exist such that P_0 is waiting for a resource that is held by P_1, P_1 is waiting for a resource that is held by P_2, …, P_{n-1} is waiting for a resource that is held by P_n, and P_n is waiting for a resource that is held by P_0.

We emphasize that all four conditions must hold for a deadlock to occur. The circular-wait condition implies the hold-and-wait condition, so the four conditions are not completely independent.

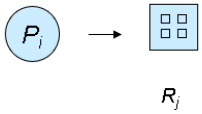
Resource-Allocation Graph:

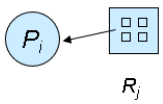
- Deadlocks can be described more precisely in terms of a directed graph called a system Resource allocation graph.
- This graph consists of a set of vertices V and a set of edges E. The set of vertices V is partitioned into two different types of nodes:
- V is partitioned into two types:
  - $P = \{P_1, P_2, \ldots, P_n\}$, the set consisting of all the active processes in the system
  - $R = \{R_1, R_2, \ldots, R_m\}$, the set consisting of all resource types in the system
- A directed edge from process Pi to resource type Rj is denoted by Pi-> Rj; it signifies that process Pi has requested an instance of resource type Rj and is currently waiting for that resource.

  request edge – directed edge $P_i \rightarrow R_j$
- A directed edge from resource type Rj to process Pi is denoted by Rj --> Pi . It signifies that an instance of resource type Rj has been allocated to process Pi

  assignment edge – directed edge $R_j \rightarrow P_i$

Symbols used in Resource allocation graph:

- Process 
- Resource Type with 4 instances 
- P_i requests instance of R_j 

- P_i is holding an instance of R_j 

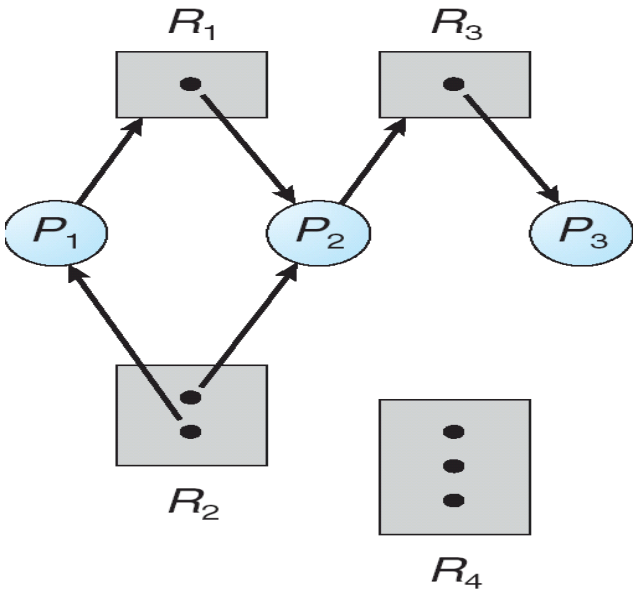
Example:

The resource-allocation graph shown in Figure 7.2 depicts the following situation.

- The sets P, R, and E:
  - $P = \{P_1, P_2, P_3\}$
  - $R = \{R_1, R_2, R_3, R_4\}$
  - $E = \{P_1 \rightarrow R_1, P_2 \rightarrow R_3, R_1 \rightarrow P_2, R_2 \rightarrow P_2, R_2 \rightarrow P_1, R_3 \rightarrow P_3\}$

Resource instances:

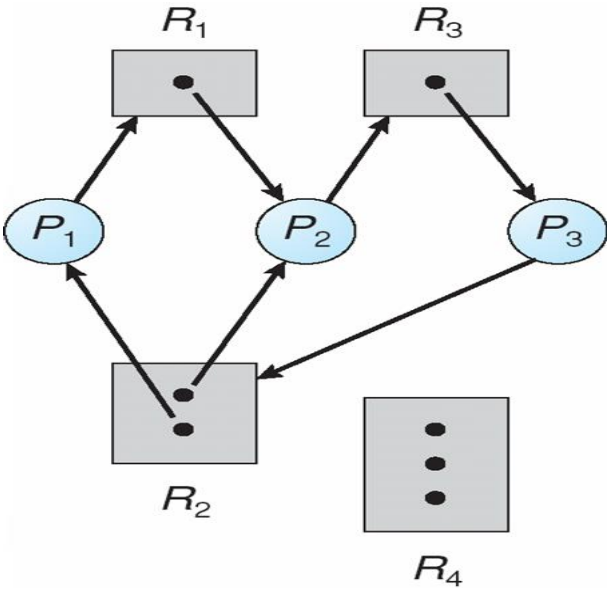
- One instance of resource type R1
- Two instances of resource type R2
- One instance of resource type R3
- Three instances of resource type R4



Process states:

- Process P1 is holding an instance of resource type R2 and is waiting for an instance of resource type R1.
- Process P2 is holding an instance of R1 and an instance of R2 and is waiting for an instance of R3.
- Process P3 is holding an instance of R3.

Resource Allocation Graph With A Deadlock:



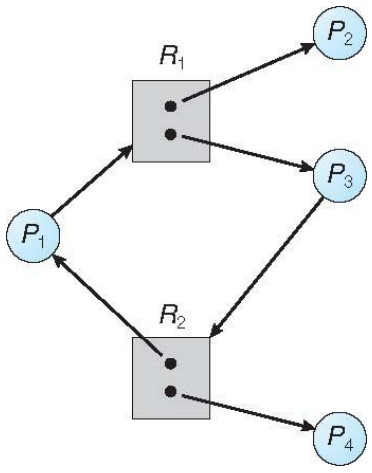
When the process P3 requests an instance of resource type R2. Since no resource instance is currently available, a request edge P3→ R2 is added to the graph .

Processes P1, P2, and P3 are deadlocked. Process P2 is waiting for the resource R3, which is held by process P3. Process P3 is waiting for either process P1 or process P2 to release resource R2. In addition, process P1 is waiting for process P2 to release resource R1

At this point, two minimal cycles exist in the system:

$$P_1 \rightarrow R_1 \rightarrow P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_1$$
$$P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_2$$

Graph With A Cycle But No Deadlock:



Sometimes even there is a cycle in Resource Allocation Graph, there is no deadlock. Observe that process P4 may release its instance of resource type R2. That resource can then be allocated to P3, breaking the cycle.

Basic Facts about Resource Allocation Graph:

- If graph contains no cycles $\Rightarrow$ no deadlock
- If graph contains a cycle $\Rightarrow$
  - if only one instance per resource type, then deadlock
  - if several instances per resource type, possibility of deadlock

Methods for Handling Deadlocks:

- We can deal with the deadlock problem in one of following ways:
- Ensure that the system will ***never*** enter a deadlock state:
  - Deadlock prevention
  - Deadlock avoidance
  - Allow the system to enter a deadlock state and then recover
  - Ignore the problem and pretend that deadlocks never occur in the system; used by most operating systems, including UNIX

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 –

- This condition means not required for sharable resources (e.g., read-only files); must hold for non-sharable resources
- 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 processes attempt to open a read-only file at the same time, they can be granted simultaneous access to the file.
- A process never needs to wait for a sharable resource.
- 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, however, it must release all the resources that it is currently allocated.
- To illustrate the difference between these two protocols, we consider a process that copies data from a DVD drive to a file on disk, sorts the file, and then prints the results to a printer.
- Both these protocols have two main disadvantages. o First, resource utilization may be low, since resources may be allocated but unused for a long period. o 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

- The third necessary condition for deadlocks is that there be 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 process is holding some resources and requests another resource that cannot be immediately allocated to it (that is, the process must wait), then all resources currently being held are preempted. (released)
- The preempted resources are added to the list of resources for which the process is waiting.
- The process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting

4. Circular Wait

- 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.
- 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.
- Each process can request resources only in an increasing order of enumeration.
- If these two protocols are used, then the circular-wait condition cannot hold.

Deadlock Avoidance

- An alternative method for avoiding deadlocks is to require additional information about how resources are to be requested.
- Each request requires that in making this decision the system consider
    - the resources currently available,
    - the resources currently allocated to each process,
    - the future requests and releases of each process.
- The simplest and most useful model requires that each process declare the maximum number of resources of each type that it may need.
- Given this a priori information, it is possible to construct an algorithm that ensures that the system will never enter a deadlocked state.
- A deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that a circular-wait condition can never exist.
- The resource-allocation state is defined by the number of available and allocated resources and the maximum demands of the processes.

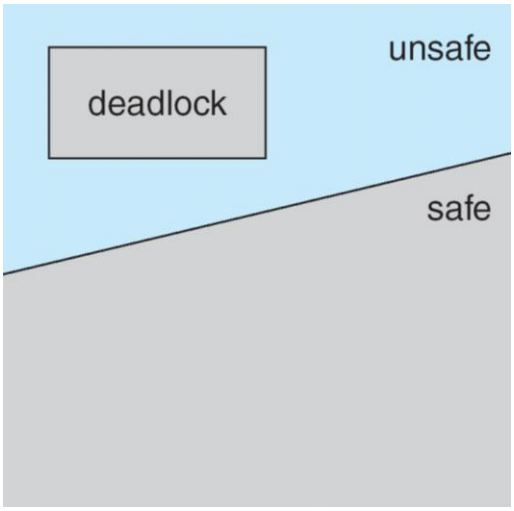
Safe State:

- When a process requests an available resource, system must decide if immediate allocation leaves the system in a safe state
- System is in **safe state** if there exists a sequence $<P_1, P_2, \ldots, P_n>$ of ALL the processes in the systems such that for each P_i, the resources that P_i can still request can be satisfied by currently available resources + resources held by all the P_j, with $j < i$
- This sequence is called a safe sequence.
- A state is **safe state** if the system can allocate resources to each process(up to its maximum requirement) in some order and still avoid a deadlock. Formally, a system is in a safe state only, if there exists a safe sequence. So a safe state is not a deadlocked state and conversely a deadlocked state is an unsafe state.
- In an **Unsafe state**, the operating system cannot prevent processes from requesting resources in such a way that any deadlock occurs. It is not necessary that all unsafe states are deadlocks; an unsafe state may lead to a deadlock.

- In this situation,
  - If P_i resource needs are not immediately available, then P_i can wait until all P_j have finished
  - When P_j is finished, P_i can obtain needed resources, execute, return allocated resources, and terminate
  - When P_i terminates, P_{i+1} can obtain its needed resources, and so on

- If no such sequence exists, then the system state is said to be unsafe.
- A safe state is not a deadlocked state. Conversely, a deadlocked state is an unsafe state. Not all unsafe states are deadlocks

Safe, Unsafe, Deadlock State



- If a system is in safe state ⇒ no deadlocks
- If a system is in unsafe state ⇒ possibility of deadlock
- Avoidance ⇒ ensure that a system will never enter an unsafe state.
- 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 difference between a safe state and an unsafe state is that from a safe state the system can guarantee that all processes will finish; from an unsafe state, no such guarantee can be given.

Deadlock Avoidance Algorithms:

- Single instance of a resource type
  - Use a Resource-allocation graph algorithm
- Multiple instances of a resource type
  - Use the Banker's algorithm

Example:

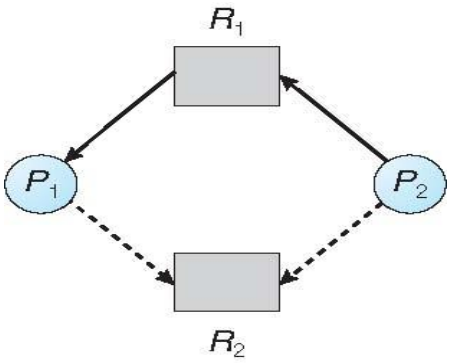
To illustrate, we consider a system with twelve magnetic tape drives and three processes: P_0, P_1, and P_2. Process P_0 requires ten tape drives, process P_1 may need as many as four tape drives, and process P_2 may need up to nine tape drives. Suppose that, at time t_0, process P_0 is holding five tape drives, process P_1 is holding two tape drives, and process P_2 is holding two tape drives. (Thus, there are three free tape drives.)

	Maximum Needs	Current Needs
P_0	10	5
P_1	4	2
P_2	9	2

At time t_0, the system is in a safe state. The sequence $<P_1, P_0, P_2>$ satisfies the safety condition. Process P_1 can immediately be allocated all its tape drives and then return them (the system will then have five available tape drives); then process P_0 can get all its tape drives and return them (the system will then have ten available tape drives); and finally process P_2 can get all its tape drives and return them (the system will then have all twelve tape drives available).

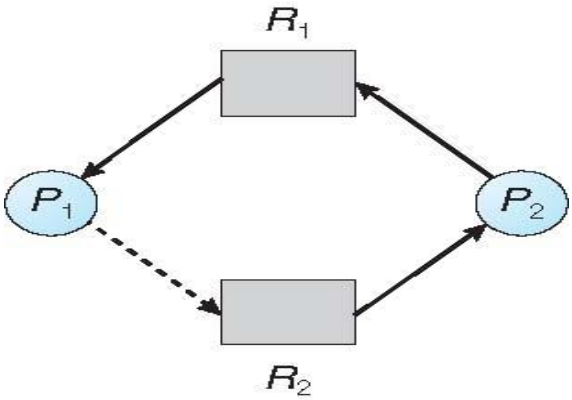
Resource-Allocation Graph Scheme:

- A new type of edge, called a claim edge is introduced.
- **Claim edge** $P_i \rightarrow R_j$ indicated that process P_j may request resource R_j; represented by a dashed line
- Claimed edge means process may request resource in future.
- Claim edge converts to request edge when a process requests a resource
- Request edge converted to an assignment edge when the resource is allocated to the process
- When a resource is released by a process, assignment edge reconverts to a claim edge
- Resources must be claimed *a priori* in the system
- A claim edge Pi → Rj indicates that process Pi 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. • When process Pi requests resource Rj , the claim edge Pi → Rj is converted to a request edge. Similarly, when a resource Rj is released by Pi , the assignment edge Rj → Pi is reconverted to a claim edge Pi → Rj . • That is, before process Pi starts executing, all its claim edges must already appear in the resource-allocation graph.
- Resource-allocation graph for deadlock avoidance.



- 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, process Pi will have to wait for its requests to be satisfied.

Unsafe State In Resource-Allocation Graph:



Resource-Allocation Graph Algorithm:

- Suppose that process P_i requests a resource R_j
- 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

Banker's Algorithm

- The resource-allocation-graph algorithm is not applicable to a resource allocation system with multiple instances of each resource type.
- The algorithm is commonly known as the banker's algorithm. 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.
- It is used for resources with Multiple instances
- Each process must a priori claim maximum use
- When a process requests a resource it may have to wait
- When a process gets all its resources it must return them in a finite amount of time

- When a new process 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 process must wait until some other process releases enough resources.

Data Structures for the Banker's Algorithm:

Let *n* = number of processes, and *m* = number of resources types.

- **Available***:*

Vector of length *m*. If Available[*j*] = *k*, there are *k* instances of resource type R_j available

- **Max***:*
  - It is a n x m matrix. If Max[i,j] = k, then process P_i may request at most *k* instances of resource type R_j
- **Allocation***:*
  - This is n x m matrix. If Allocation[i,j] = k then P_i is currently allocated *k* instances of R_j
- **Need***:*
  - This is n x m matrix. If Need[i,j] = *k*, then P_i may need *k* more instances of R_j to complete its task

    Need [i,j] = Max[i,j] – Allocation [i,j]

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

Resource-Request Algorithm for Process P_i:

We describe the algorithm for determining whether requests can be safely granted

$Request_i$ = request vector for process P_i. If $Request_i[j] = k$ then process P_i wants k instances of resource type R_j

1. If $Request_i \leq Need_i$ go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim
2. If $Request_i \leq Available$, 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 safe $\Rightarrow$ the resources are allocated to P_i

   If unsafe $\Rightarrow$ P_i must wait, and the old resource-allocation state is restored

If the resulting resource-allocation state is safe, the transaction is completed, and process Pi is allocated its resources. However, if the new state is unsafe, then Pi must wait for Request-i, and the old resource-allocation state is restored.

Example of Banker's Algorithm

n=5 processes P0 through P4

- 5 processes P_0 through P_4;
- 3 resource types:

A (10 instances), *B* (5instances), and *C* (7 instances)

- Snapshot at time T_0:

	Allocation	*Max*	*Available*
	A B C	*A B C*	*A B C*
P_0	0 1 0	7 5 3	3 3 2
P_1	2 0 0	3 2 2	
P_2	3 0 2	9 0 2	
P_3	2 1 1	2 2 2	
P_4	0 0 2	4 3 3	

- The content of the matrix *Need* is defined to beNeed= *Max – Allocation*

	Need
	A B C
P_0	7 4 3
P_1	1 2 2
P_2	6 0 0
P_3	0 1 1

P_4 4 3 1

- The system is in a safe state since the sequence < P_1, P_3, P_4, P_2, P_0> satisfies safety criteria
- We should check the condition Need<= Available . Which process satisfies the condition will be executed.

	Allocation	*Max*	*Available*	*Need*
	A B C	*A B C*	*A B C*	*A B C*
P_0	0 1 0	7 5 3	3 3 2	7 4 3
P_1	2 0 0	3 2 2	5 3 2	1 2 2
P_2	3 0 2	9 0 2	7 4 3	6 0 0
P_3	2 1 1	2 2 2	7 4 5	0 1 1
P_4	0 0 2	4 3 3	10 4 7	4 3 1
			10 5 7	

- The system is in a safe state since the sequence < P_1, P_3, P_4, P_2, P_0> satisfies safety criteria

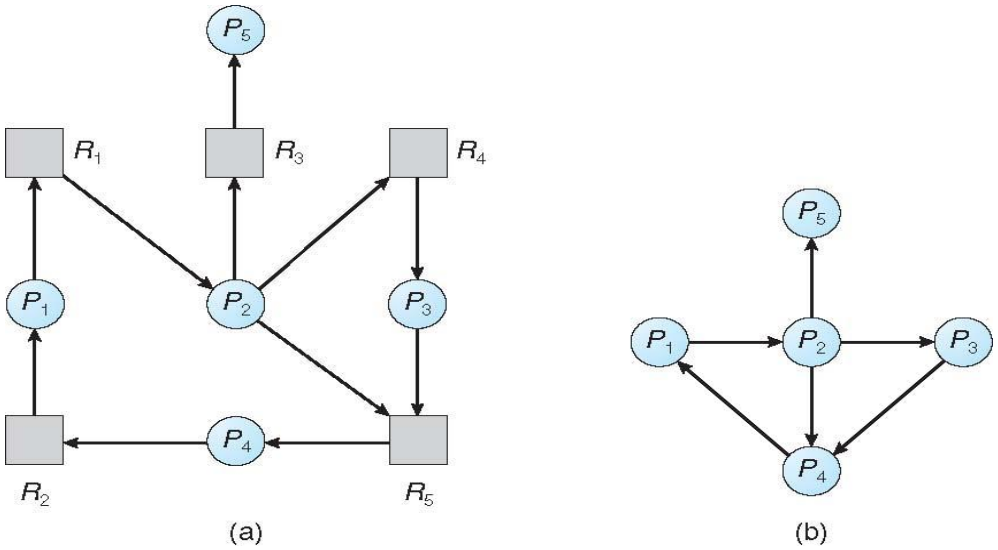
Deadlock Detection

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

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.
- It Maintain **wait-for** graph
  - Nodes are processes
  - $P_i \rightarrow P_j$ if P_i is waiting for P_j do release a resource that Pi needs.
- An edge from Pi to Pj in a wait-for graph implies that process Pi is waiting for process Pj to release a resource that Pi needs.
- Periodically invoke an algorithm that searches for a cycle in the graph. If there is a cycle, there exists a deadlock.
- An edge Pi → Pj exists in a wait-for graph if and only if the corresponding resource
- allocation graph contains two edges Pi → Rq and Rq → Pj for some resource Rq .
- A deadlock exists in the system if and only if the wait-for graph contains a cycle.

- An algorithm to detect a cycle in a graph requires an order of n^2 operations, where n is the number of vertices in the graph



(a)Resource-Allocation Graph and (b) Corresponding wait-for graph

- 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 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 a resource-allocation system with multiple instances of each resource type.

Data Structures :

• **Available:**

It is a vector of length "m" indicates the number of available resources of each type.

• **Allocation**. :

An n × m matrix defines the number of resources of each type currently allocated to each process.

• **Request.** An n × m matrix indicates the current request of each process. If Request[i][j] = k, then process Pi is requesting k more instances of resource type Rj .

Detection Algorithm

1. Let ***Work*** and ***Finish*** be vectors of length ***m*** and ***n***, respectively Initialize:
   (a) ***Work* = *Available***
   (b) For ***i* = 1,2, …, *n***, if ***Allocation***$_i \neq$ **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***, $1 \leq i \leq n$, then the system is in deadlock state. Moreover, if ***Finish*[*i*] == *false***, then P_i is deadlocked

Algorithm requires an order of O(m x n2) operations to detect whether the system is in deadlocked state

Example of Detection Algorithm:

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

	Allocation	Request	Available
	A B C	*A B C*	*A B C*
P_0	0 1 0	0 0 0	0 0 0
P_1	2 0 0	2 0 2	
P_2	3 0 3	0 0 0	
P_3	2 1 1	1 0 0	
P_4	0 0 2	0 0 2	

- Sequence <P_0, P_2, P_3, P_1, P_4> will result in ***Finish[i] = true*** for all ***i***

Solution:

	Allocation	Request	Available
	A B C	*A B C*	*A B C*
P_0	0 1 0	0 0 0	0 0 0
P_1	2 0 0	2 0 2	0 1 0
P_2	3 0 3	0 0 0	3 1 3
P_3	2 1 1	1 0 0	5 2 4
P_4	0 0 2	0 0 2	7 2 4
			7 2 6

Safe Sequence <P_0, P_2, P_3, P_1, P_4> will result in ***Finish[i] = true*** for all ***i***

A situation when a deadlock exists:

- In the above example: if the request of P2 is 0 0 1, then check the deadlock
- Five processes P_0 through P_4; three resource types
  A (7 instances), *B* (2 instances), and *C* (6 instances)
- Snapshot at time T_0:

	Allocation	Request	Available

	A B C	*A B C*	*A B C*
P_0	0 1 0	0 0 0	0 0 0
P_1	2 0 0	2 0 2	0 1 0
P_2	3 0 3	0 0 1	
P_3	2 1 1	1 0 0	
P_4	0 0 2	0 0 2	

- The process $\mathbf{P_0}$ can execute. But after P0, available is only 0,1,0 , so P1 cannot execute and no other process can execute. We claim that the system is now deadlocked. Although we can reclaim the resources held by process Po, the number of available resources is not sufficient to fulfill the requests of the other processes. Thus, a deadlock exists, consisting of processes P1, P2, P3, and P4.

Detection-Algorithm Usage

- When should we invoke the detection algorithm? The answer depends on two factors:
  1. How often is a deadlock likely to occur?
  2. How many processes will be affected by deadlock when it happens?
- If deadlocks occur frequently, then the detection algorithm should be invoked frequently.
- Resources allocated to deadlocked processes will be idle until the deadlock can be broken. In addition, the number of processes involved in the deadlock cycle may grow.
- Deadlocks occur only when some process makes a request that cannot be granted immediately. This request may be the final request that completes a chain of waiting processes.
- Invoking the deadlock-detection algorithm for every resource request will incur considerable overhead in computation time.
- A less expensive alternative is simply to invoke the algorithm at defined intervals-for example, once per hour or whenever CPU utilization drops below 40 percent.
- If detection algorithm is invoked arbitrarily, there may be many cycles in the resource graph and so we would not be able to tell which of the many deadlocked processes “caused” the deadlock.

Recovery from Deadlock :

Recovery from Deadlock can be done in two ways

To eliminate deadlocks by aborting a process, we use one of two methods. In both methods, the system reclaims all resources allocated to the terminated processes.
The two methods are 1) Process termination and 2) Resource Preemption.

Process Termination:

When a detection algorithm determines that a deadlock exists, several alternatives are available.

1. One possibility is to inform the operator that a deadlock has occurred and to let the operator deal with the deadlock manually.
2. Another possibility is to let the system recover from the deadlock automatically.

- There are two options for breaking a deadlock.

1. Abort all deadlocked processes
2. Abort one process at a time until the deadlock cycle is eliminatedProcess Termination :

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

• **Abort all deadlocked processes**.

This method clearly will break the deadlock cycle, but
at great expense. The deadlocked processes may have computed for a long time, and the results of these partial computations must be discarded and probably will have to be recomputed later.

• **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.

Many factors may affect which process is chosen, including:

1. What is the Priority of the process
2. How long process has computed, and how much longer to completion
3. Resources the process has used- How many and what types of resources the process has used(for example, whether the resources are simple to preempt)
4. How many Resources process needs to complete
5. How many processes will need to be terminated
6. Whether the process interactive or batch?

Resource Preemption:

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.

Issues

1. **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. As in process termination, we must determine the order of preemption to minimize cost.

2. **Rollback.**

- If we preempt a resource from a process, what should be done with that process?
- We must roll back the process to some safe state and restart it from that state.
- Since, in general, it is difficult to determine what a safe state is, the simplest solution is a total rollback: abort the process and then restart it. Although it is more effective to roll back the process only as far as necessary to break the deadlock, this method requires the system to keep more information about the state of all running processes.

3. **Starvation.**

- How do we ensure that starvation will not occur?
- 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. As a result, this process never completes its designated task, a starvation situation that must be dealt with in any practical system. Clearly, 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.