UNIT – III

Syllubus:

Synchronization tools - the Critical- section problem, Peterson's solution, Semaphores, classic problems of synchronization, monitors.
Deadlocks – System model, deadlock characterization, deadlock prevention, detection and avoidance, recovery form deadlock.

The Critical-Section Problem: Consider a system consisting of n processes (P1, P2, P3…..Pn). Each process has a segment of code, called a **critical section,** 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 is, no two processes are executing in their critical sections at the same time. 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 general structure of a typical process P_i is shown below.

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

General structure of a typical process P_i.

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

- **Mutual exclusion:** If process P_i is executing in its critical section, then no other processes can be executing in their critical sections.
- **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 enter its critical section next.
- **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.

Peterson's Solution: A classic software-based solution to the critical-section problem known as **Peterson's solution.** Peterson's solution is restricted to two processes that alternate execution between their critical sections and remainder sections. Peterson's solution requires two data items to be shared between the two processes:

```
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 P_i 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 f lag[i] is true, this value indicates that P_i is ready to enter its critical section.

To enter the critical section, process P_i, 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.

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 updated value of turn decides which of the two processes is allowed to enter its critical section first.

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

        critical section

    flag[i] = FALSE;

        remainder section

} while (TRUE);
```

The structure of process P_i in Peterson's solution.

To prove property 1(Mutual Exclusion), note that each P_i 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, since the value of turn can be either 0 or 1 but cannot be both.

To prove properties 2 and 3, note that a process Pj, can be prevented from entering the critical section only if it is stuck in the while loop with the condition flag [j] == true and turn == j; this loop is the only one possible. If Pj is not ready to enter the critical section, then flag [j] == false, and Pi can enter its critical section. If *Pj* has set flag [j] to true and is also executing in its while statement, then either turn == i or turn == j . If turn == i, then P_i will enter the critical section. If turn == j, then *Pj* will enter the critical section. However, once *Pj* exits its critical section, it will reset f lag[j] to false, allowing Pi to enter its critical section. If *Pj* resets flag [j] to true, it must also set turn to i.

Semaphores: A semaphore S is an integer variable which can be incremented and decremented. The only two atomic operations that can be performed on semaphores are wait( ) and signal( ).The wait( ) operation was originally termed P; signal( ) was originally called V. The definition of wait( ) is as follows:

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

The definition of signal () is as follows:

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

All the modifications to the integer value of the semaphore in the wait( ) and signal( ) operations must be executed indivisibly. That is, when one process modifies the semaphore value, no other process can simultaneously modify that same semaphore value.

There are two types of semaphores: binary semaphore and counting semaphore. The value of a **counting semaphore** can range over an unrestricted domain. The value of a **binary semaphore** can range only between 0 and 1. On some systems, binary semaphores are known as **mutex locks,** as they are locks that provide *mutualexclusion*.

Binary semaphores are used to deal with the critical-section problem for multiple processes. The *n* processes share a semaphore, mutex, initialized to 1. Each process P_i is organized as shown below.

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

Mutual-exclusion implementation with semaphores.

Counting semaphores can be used to control access to a given resource consisting of a finite number of 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.

Semaphores are also used to solve various synchronization problems. For example, consider two concurrently running processes: *P1* with a statement S1 and *P2* with a statement *S2*. Suppose we require that *S2* be executed only after S1 has completed. We can implement this scheme readily by letting P1 and P*2* share a common semaphore **synch**, initialized to 0, and by inserting the statements

S1;

signal(synch);

in process *P1*, and the statements

wait(synch);

S2;

in process P2. 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.

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 **spinlock** because the process "spins" while waiting for the lock.

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.

The following C code implements this.

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

Each semaphore has an integer value and a list of processes list . 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 now be defined as

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

The signal( ) semaphore operation can now 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.

Deadlocks and Starvation: 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.**

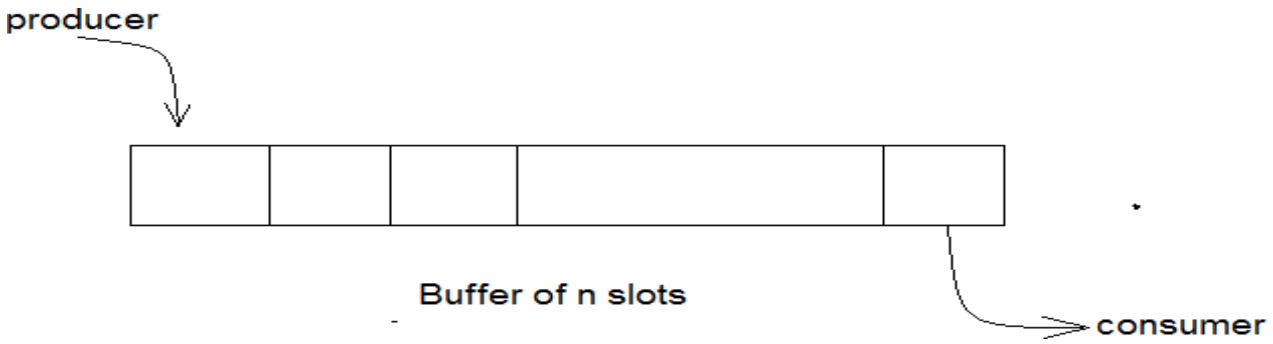
For example, consider a system consisting of two processes, P0 and P1, each accessing two semaphores, S and Q, set to the value 1: Suppose that *P0* executes wait (S) and then P1 executes wait (Q). When P0 executes wait(Q), it must wait until P1 executes signal(Q). Similarly, when P1 executes wait(S), it must wait until P0 executes signal(S). Since these signal () operations cannot be executed, P0 and P1 are deadlocked.

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

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

Classic Problems of Synchronization:

The Bounded-Buffer Problem: There is a buffer of n slots and each slot is capable of storing one unit of data. There are two processes running, namely, **producer** and **consumer**, which are operating on the buffer.



Bounded Buffer Problem

A producer tries to insert data into an empty slot of the buffer. A consumer tries to remove data from a filled slot in the buffer. As you might have guessed by now, those two processes won't produce the expected output if they are being executed concurrently. There needs to be a way to make the producer and consumer work in an independent manner.

Solution

One solution of this problem is to use semaphores. The semaphores which will be used here are:

- mutex, a **binary semaphore** which is used to acquire and release the lock.
- empty, a **counting semaphore** whose initial value is the number of slots in the buffer, since, initially all slots are empty.
- full, a **counting semaphore** whose initial value is 0.

At any instant, the current value of empty represents the number of empty slots in the buffer and full represents the number of occupied slots in the buffer.

```
do {
    . . .
   // produce an item in nextp
    . . .
   wait(empty);
   wait(mutex);
    . . .
   // add nextp to buffer
    . . .
   signal(mutex);
   signal(full);
}while (TRUE),-
```

The structure of the producer process.

```
do {
   wait(full);
   wait(mutex);
      . . .
   // remove an item from buffer to nextc
      . . .
   signal(mutex);
   signal(empty);
      . . .
   // consume the item in nextc
      . . .
}while (TRUE);
```

The structure of the consumer process.

The Readers-Writers Problem: In general, 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, to read and write) the database. If two readers access the shared data simultaneously, no problem will occur. However, if a writer and some other thread (either a reader or a writer) access the database simultaneously, problems may ensue. This synchronization problem is referred to as the *readers-writers problem.*

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. 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.

In the solution to the first readers-writers problem, the reader processes share the following data structures:

semaphore mutex, wrt;

int readcount;

The semaphores mutex and wrt are initialized to 1; readcount is initialized to 0. The semaphore wrt is common to both reader and writer processes. The mutex semaphore is used to ensure mutual exclusion when the variable readcount is updated. The readcount variable keeps track of how many processes are currently reading the object. The semaphore wrt functions as a mutual- exclusion semaphore for the writers. It is also used by the first or last reader that enters or exits the critical section. It is not used by readers who enter or exit while other readers are in their critical sections. The structure of reader and writer processes are shown below.

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

The structure of a writer 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);
```

The structure of a reader process.

Note that, if a writer is in the critical section and *n* readers are waiting, then one reader is queued on wrt, and n-1 readers are queued on mutex. Also observe that, when a writer executes signal (wrt), we may resume the execution of either the waiting readers or a single waiting writer. The selection is made by the scheduler.

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 as shown in the diagram. 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 neighbor. When a hungry philosopher has both her chopsticks at the same time, she eats without releasing her chopsticks. When she is finished eating, she puts down both of her chopsticks and starts thinking again.

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. Thus, the shared data are

semaphore chopstick[5];

where all the elements of chopstick are initialized to 1. The structure of philosopher i is shown below.

```
do {
   wait(chopstick[i]);
   wait(chopstick[(i+1) % 5]);
      . . .
   // eat
      . . .
   signal(chopstick[i]);

   signal(chopstick[(i+1) % 5]);

   // think

}while (TRUE);
```

The structure of philosopher *i*.

Monitors: Using Sempaphores incorrectly can result in timing errors that are difficult to detect. For example,

- Suppose that a process interchanges the order in which the wait(j and signal( ) operations on the semaphore mutex are executed, resulting in the following execution:

```
signal(mutex);
        //critical section
wait(mutex);
```

  In this situation, several processes may be executing in their critical sections simultaneously, violating the rmitual-exclusion requirement.
- Suppose that a process replaces signal (mutex) with wait (mutex). That is, it executes

```
wait(mutex);
        //critical section
wait(mutex);
```

  In this case, a deadlock will occur.
- Suppose that a process omits the wait (mutex), or the signal (mutex), or both. In this case, either mutual exclusion is violated or a deadlock will occur.

These examples illustrate that various types of errors can be generated easily when programmers use semaphores incorrectly to solve the critical-section problem. To deal with such errors, ne fundamental high-level synchronization construct—the monitor type is used.

A Monitor is a type, or abstract data type presents a set of programmer-defined operations that are provided mutual exclusion within the monitor. The monitor type also contains the declaration of variables, along with the bodies of procedures or functions. The syntax of a monitor is shown below.

```
monitor monitor name
{
    // shared variable declarations
    procedure P1 ( . . . )
    {
    …….
    }
    procedure P 2 ( . . . )
    {
    ……
    }
```

```
……
……
……
procedure P n ( . . . )
{
…….
}
Initializationcode( . . . )
{
…….
}
}
```

The monitor construct ensures that only one process at a time can be active within the monitor, i.e.mutual exclusion is achieved. To overcome synchronization problems, a condition construct is used as shown below.

condition x, y;

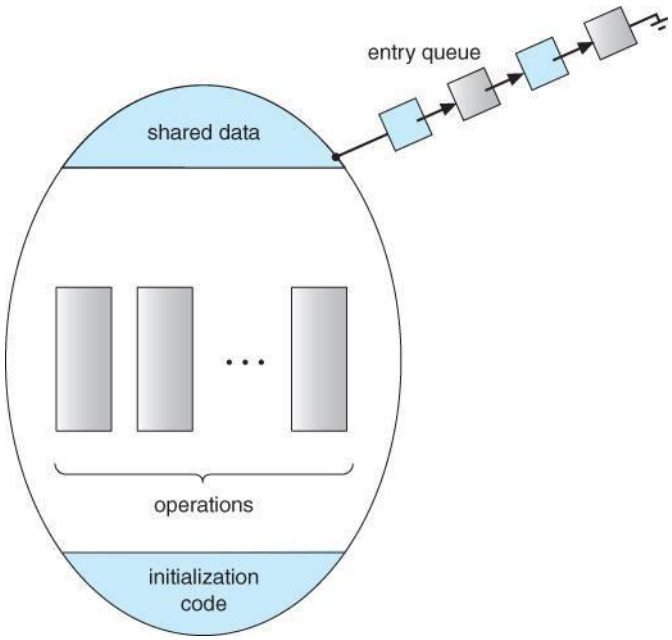
The only operations that can be invoked on a condition variable are wait( ) and signal( ). The operation

x.wait() ;

means that the process invoking this operation is suspended until another process invokes

x.signal() ;

The x. signal( ) operation resumes exactly one suspended process. If no process is suspended, then the signal( ) operation has no effect; that is, the state of x is the same as if the operation had never been executed.



Schematic View of a Monitor

For example, when the x. **s**ignal( ) operation is invoked by a process P, there is a suspended process Q associated with condition x. Clearly, if the suspended process Q is allowed to resume its execution, the signaling process P must wait. Otherwise, both P and Q would be active simultaneously within the monitor. Note, however, that both processes can conceptually continue with their execution. Two possibilities exist:

1. **Signal and wait.** *P* either waits until Q leaves the monitor or waits for another condition.
2. **Signal and continue.** *Q* either waits until *P* leaves the monitor or waits for another condition.

Dining-Philosophers Solution Using Monitors: This solution imposes the restriction that a philosopher may pick up her chopsticks only if both of them are available. To do this use the following data structure.

enum {thinking, hungry, eating} state[5] ;

Here, 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).

condition self [5];

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

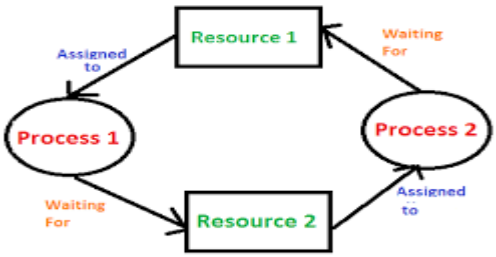
The distribution of the chopsticks is controlled by the monitor dp as shown below.

dp.pickup(i);
…….
//eat
……
dp.putdown(i);

```
monitor dp
{
   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 til = THINKING;
      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;
    }
}
```

A monitor solution to the dining-philosopher problem.

Deadlock: In a multiprogramming environment, several processes may compete for a finite number of resources. A process requests resources; and 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.



A process may utilize a resource in only the following sequence:

- **Request:** 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.
- **Use:** The process can operate on the resource (for example, if the resource is a printer, the process can print on the printer).
- **Release:** The process releases the resource.

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.** A process must be 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$, Pi, ..., Pn*\ of waiting processes must exist such that *P-0* is waiting for a resource held by *P*\, *P*\ is waiting for a resource held by P?, •••, P.,--i is waiting for a resource held by *Pn*, and *P*,, is waiting for a resource held by Pn.

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: P={P1, P2, P3…..Pn}, the set consisting of all the active processes in the system, and *R = {R1, R2, • • • Rm}*, the set consisting of all resource types in the system.

A directed edge from process P_i to resource type *Rj* is denoted by $P_i \rightarrow Rj$; it signifies that process P_i has requested an instance of resource type Rj and is currently waiting for that resource. A directed edge from resource type *Rj* to process P_i is denoted by $Rj \rightarrow P_i$; it signifies that an instance of resource type *Rj* has been allocated to process P_i. A directed edge $P_i \rightarrow Rj$ is called a **request edge;** a directed edge $Rj \rightarrow P_i$; is called an **assignment edge.**

Pictorially, represent each process P_i as a circle and each resource type *Rj* as a rectangle. Since resource type *Rj* may have more than one instance, represent each such instance as a dot within the rectangle.

When process P_i requests an instance of resource type *Rj*, a request edge is inserted in the resource-allocation graph. When this request can be fulfilled, the request edge is *instantaneously* transformed to an assignment edge. When the process no longer needs access to the resource, it releases the resource; as a result, the assignment edge is deleted.

Consider the following resource allocation graph, containing

➔ The sets P, *R,* and £:

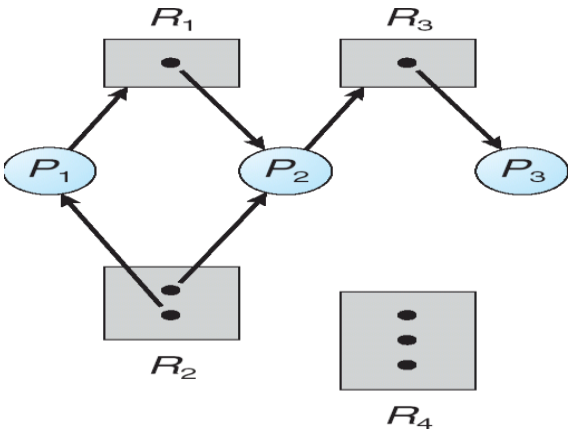
- P={P1, P2, ……PN}
- R= {R1, R2 … RM}
- £ = {P1→ R1, P2→ R3, R1→P2, R2→ P2, R2→P1,….. R3 -> P3 }

➔ 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

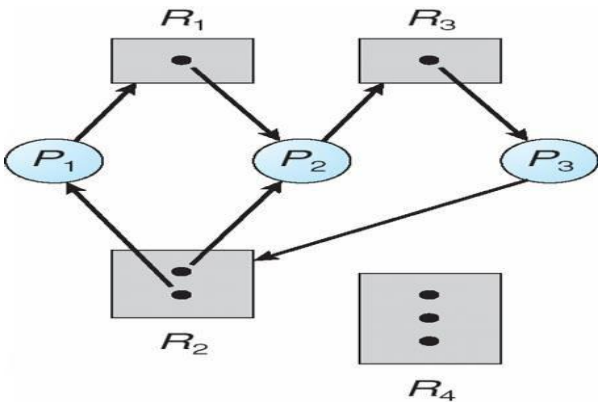
➔ Process states:

- 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.



If each resource type has exactly one instance, then a cycle implies that a deadlock has occurred. If each resource type has several instances, then a cycle does not necessarily imply that a deadlock has occurred. In this case, a cycle in the graph is a necessary but not a sufficient condition for the existence of deadlock.

Suppose that 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 as shown below.



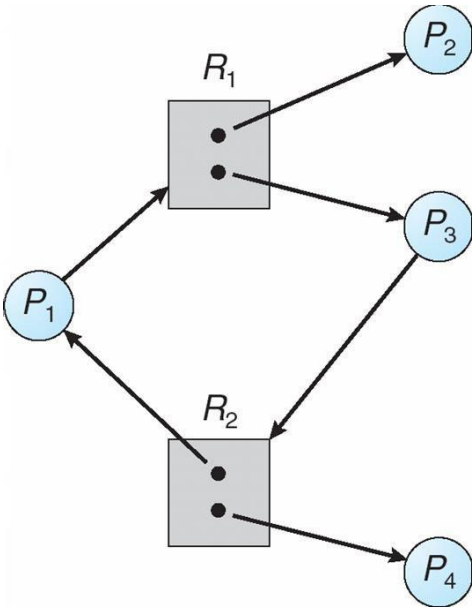
At this point, two minimal cycles exist in the svstem:

P1→R1→P2→R3→P3→R2→P1

P2→R3→P3→R2→P2

Here, 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 P2 to release resource R2. In addition, process P1 is waiting for process P2 to release resource R1.

Now consider the resource-allocation graph as shonw below, having a cycle (P1→R1→P3→R2→P1)with no dead locks. Observe that process P4 may release its instance of resource type R2. That resource can then be allocated to P3, breaking the cycle.



Deadlock Prevention: By ensuring that at least one of the following conditions cannot hold, we can *prevent* the occurrence of a deadlock.

1. **Mutual Exclusion:** The mutual-exclusion condition must hold for nonsharable resources. For example, a printer cannot be simultaneously shared by several processes. Sharable resources, in contrast, 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.
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. For this, release all the resources held by it whenever it requests a resource which is not available.
3. No Preemption: this condition can be prevented in several ways.
   - If a process holding certain resources is denied a further request. That process must release its original resources and if necessary request them again, together with an additional resource.
   - If a process requests a resource that is currently held by another process, the operating system may preempt the second process and require it to release its resources.
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. For example, process hold resource type R1, then it can only request resource of class R2 or R3 etc…

Advantages of Deadlock Prevention:

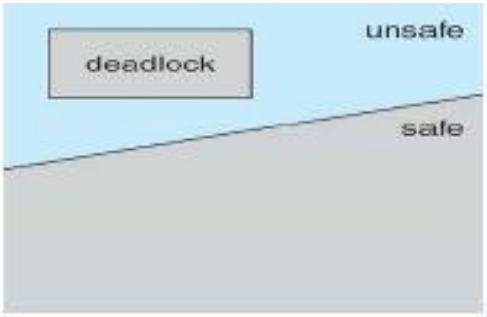
- No preemption necessary
- Needs no runtime computation
- Feasible to enforce via compile time checks.
- Works well processes that perform a single burst of activity.

Disadvantages of Deadlock Prevention:

- Inefficient
- Delays process initiation
- Disallows incremental resource requests.

Deadlock Avoidance: A deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that a circularwait 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: A state is *safe* if the system can allocate resources to each process in some order and still avoid a deadlock. More formally, a system is in a safe state only if there exists a safe sequence. A sequence of processes <P1, P2, ..., Pn> is a safe sequence for the current allocation state if, for each *Pi*, the resource requests that Pi can be satisfied by the currently available resources plus the resources held by all *Pi*, with j < i. In this situation, if the resources that *Pi* needs are not immediately available, then Pi can wait until all *Pj* have finished. When they have finished, Pj can obtain all of its needed resources, complete its designated task, return its allocated resources, and terminate. When Pi terminates, Pi+l can obtain its needed resources, and so on. If no such sequence exists, then the system state is said to be *unsafe*.



Safe, unsafe, and deadlock state spaces.

A safe state is not a deadlocked state. Conversely, a deadlocked state is an unsafe state. Not all unsafe states are deadlocks, however. An unsafe state *may* lead to a deadlock. As long as the state is safe, the operating system can avoid unsafe (and deadlocked) states. In an unsafe state, the operating system cannot prevent processes from requesting resources such that a deadlock occurs: The behavior of the processes controls unsafe states.

Consider a system with 12 magnetic tape drives and three processes: P0, P1, and P2. Process P0 requires 10 tape drives, process P1 may need as many as 4 tape drives, and process P2 may need up to 9 tape drives. Suppose that, at time *to*, process P0 is holding 5 tape drives, process *P1* is holding *2* tape drives, and process *P2* is holding 2 tape drives. (Thus, there are 3 free tape drives.)

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

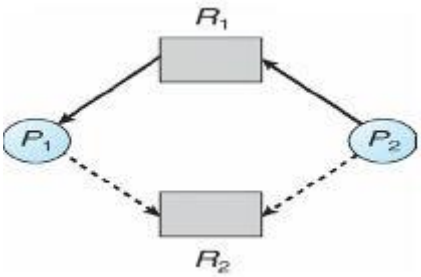
At time T0, the system is in a safe state. The sequence < P1, *P0, P2*> satisfies the safety condition. Process P1 can immediately be allocated all its tape drives and then return them (the system will then have 5 available tape drives); then process P0 can get all its tape drives and return them (the system will then have 10 available tape drives); and finally process P2 can get all its tape drives and return them (the system will then have all 12 tape drives available).

A system can go from a safe state to an unsafe state. Suppose that, at time T1, process *P2* requests and is allocated one more tape drive. The system is no longer in a safe state. At this point, only process P1 can be allocated all its tape drives. When it returns them, the system will have only 4 available tape drives. Since process P0 is allocated 5 tape drives but has a maximum of 10, it may request 5 more tape drives. Since they are unavailable, process P0 must wait. Similarly, process P2 may request an additional 6 tape drives and have to wait, resulting in a deadlock. Here, mistake was in granting the request from process P2 for one more tape drive. If we had made P2 wait until either of the other processes had finished and released its resources, then we could have avoided the deadlock. The following are two different types of deadlock avoidance algorithms.

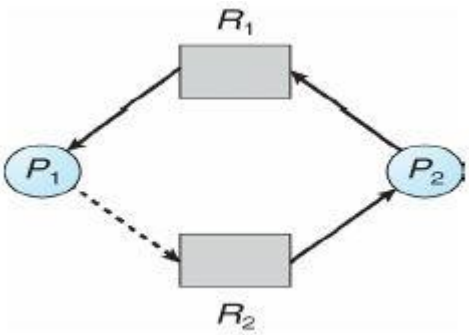
1. **Resource-Allocation-Graph Algorithm:** A claim edge Pi → *Rj* indicates that process Pi may request resource *Rj* at some time in the future. This edge resembles a request edge in direction but is represented in the graph by a dashed line. When process Pi requests resource *Rj*, the claim edge Pi→ *Rj* is converted to a request edge. Similarly, when a resource *Rj* is released by *Pj*, the assignment edge *Rj*→ Pi is reconverted to a claim edge *Pi*→ *Rj*.

   Suppose that process Pi requests resource *Rj*. The request can be granted only if converting the request edge Pi → *Rj* to an assignment edge *Rj*→ *Pi* does not result in the formation of a cycle in the resource-allocation graph. An algorithm for detecting a cycle in this graph requires an order of *n2* operations, where *n* is the number of processes 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. Therefore, process *Pi* will have to wait for its requests to be satisfied. For example,



Resource-allocation graph for deadlock avoidance.



An unsafe state in a resource-allocation graph

Suppose that P2 requests *R2*. Although *R2* is currently free, cannot allocate it to P2, since this action will create a cycle in the graph. A cycle indicates that the system is in an unsafe state. If P1 requests *R2,* and *P2* requests *R1,* then a deadlock will occur.

2. **Banker's Algorithm:** This algorithm is suitable for resource types with multiple instances. 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.

Several data structures must be maintained to implement the banker's algorithm. These data structures encode the state of the resource-allocation system. Let *n* be the number of processes in the system and *m* be the number of resource types. Data structures are

- **Available:** A vector of length *m* indicates the number of available resources of each type. If *Available[j]* equals *k*, there are *k* instances of resource type Rj available.
- **Max:** An *n* x *m* matrix defines the maximum demand of each process. If Max[i][j] equals *k*, then process *Pi* may request at most *k* instances of resource type Rj.
- **Allocation**: An *n* x m matrix defines the number of resources of each type currently allocated to each process. If *Allocation[i][j]* equals *k*, then process *Pi* is currently allocated *k* instances of resource type Rj.
- **Need:** An *n* x *m* matrix indicates the remaining resource need of each process. If *Need[i][j]* equals *k*, then process Pi may need *k* more instances of resource type *Rj* to complete its task. Note that *Need*[i][j] equals *Max[i][j]- Allocntion[i][j].*

Safety Algorithm:

1. Let *Work* and *Finish* be vectors of length *m* and *n*, respectively. Initialize: *Work* = *Available*
   Finish [*i*] = *false* for *i* = 0, 1, …, *n*- 1
2. Find an index *i* such that both: *Finish* [*i*] = *false* and $Need_i$ <= *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

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

1. If $Request_i$ < *Need*, go to step 2. Otherwise, raise an error condition, since the process has exceeded its maximum claim.
2. If $Request_i$ < *Available*, go to step 3. Otherwise, P_i must wait, since the resources are not available.
3. *Available = Available – Reques*$_i$*;*
   $Allocation_i = Allocation_i + Request_i$
   $Need_i = Necd_i – Request_i$

Consider a system with five processes *P0* through P4 and three resource types *A*, *B*, and C. Resource type A has 10 instances, resource type *B* has 5 instances, and resource type *C* has 7 instances.

	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 be *Max - Allocation* and is as follows:

	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

Suppose now that process *P0* requests 7 additional instance of resource type *A* , 4 instances of resource type B and 4 instances of resource type B. Now check for Need <= Work(Avaialable) for all the values of i.

→For process P0, Since (7, 4, 3) <= (3, 3, 2) is false. Make process P0 to wait.

→For process P1, Since (1, 2, 2) <= (3, 3, 2) is true. Process it. After completion of P1, it releases all the releases it held. So, available is updated as follows:

Aaialable = available + Allocation$_i$

Aaialable = (3, 3, 2) + (2, 0, 0) = (5, 3, 2)

➔ For process P2, Since (6, 0 ,0) <= (5, 3, 2) is False, Make P2 to wait.

➔ For process P3, Since (0, 1 ,1) <= (5, 3, 2) is True, Process it. After completion of P3, it releases all the resources it held. So, Available is updated as follows.

Aaialable = (5, 3, 2) + (2, 1, 1) = (7, 4, 3)

➔ For process P4, Since (4, 3 ,1) <= (7, 4, 3) is True, Process it. After completion of P4, it releases all the resources it held. So, Available is updated as follows.

Aaialable = (7, 4, 3) + (0, 0, 2) = (7, 4, 5)

➔ For process P0, Since (7, 4 ,3) <= (7, 4, 5) is True, Process it. After completion of P0, it releases all the resources it held. So, Available is updated as follows.

Aaialable = (7, 4, 5) + (0, 1, 0) = (7, 5, 5)

➔ For process P2, Since (6, 0 ,0) <= (7, 5, 5) is True, Process it. After completion of P2, it releases all the resources it held. So, Available is updated as follows.

Aaialable = (7, 5, 5) + (3, 0, 2) = (10, 5, 7)

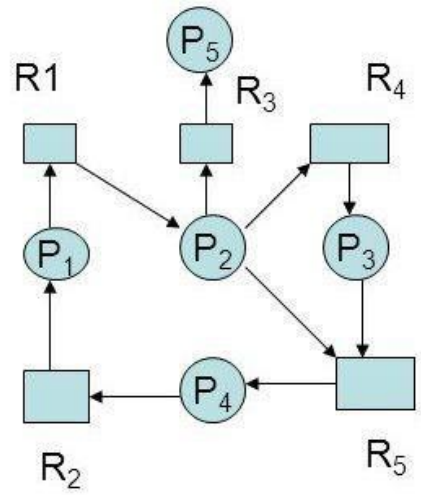
Deadlock detection: If a system does not empliy either a deadlock prevention or a deadlock avoidance algorithm, then a deadlock situation may occur. In this environment, the system must provide:

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

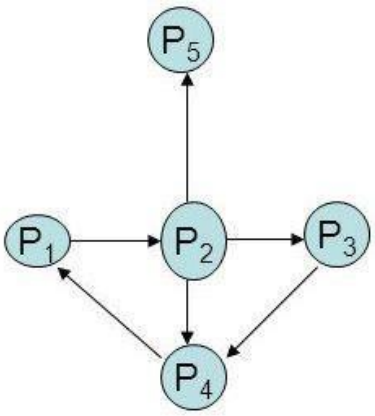
The following are different types of deadlock detection algorithms:

1. Single Instance of Each Resource Type (Wait-For Graph): Wait-For graph is an sibling of Resource-Allocation Graph algorithm. More precisely, an edge from P_i to P_j in a Wait-For graph implies that process P_i is waiting for process P_j to release a resource that P_i needs. An

edge $P_i \rightarrow P_j$ exists in a wait-fpr graph if and only the corresponding resource allocation graph contains two edges $P_i \rightarrow R_j$ and $R_j \rightarrow P_j$ for more resource. For example,



Resource allocation graph



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 wai-for graph and periodically invoke an algorithm that's earches for a cycle in the graph. An algorithm to detect a cyckle in a graph requires an order of n^2 operstions, where n is the number of vertices in the graph.

2. Several Instances of resource Type (Banker's Algorithm) : The Wait-For graph is not applicable to a resource allocation system with multiple instances of each resource type. The algorithm employs several time varying data structures that are similar to those used in the banker's algorithm.
   - Available : 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]* equals *k*, then process P_i is requesting *k* more instances of resource type *Rj.*

Algorithm:

1. Let *Work* and *Finish* be vectors of length *m* and *n*, respectively. Initialize *Work= Available. Fori*= 0, 1, ... , *n-1,* if $Allocation_i$ # 0, then *Finish[i] =false;* otherwise, *Finish[i] = true.*
2. Find an index *i* such that both
   Finish[i] ==false
   $Request_i$ < *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] ==false* for some *i,* 0 < *i* < *n,* then the system is in a deadlocked state. Moreover, if *Finish[i] ==false,* then process P_i is deadlocked.

This algorithm requires an order of m x n^2 operations to detect whether the system is in a deadlocked state. Consider a system with 5 processes P0 through P4 and three resource types A, B

and C. Resource type A has 7 instances, B has two instances, and C has 6 instances. Suppose that, at time T_0, we have the following resoutce allocation state.

	Allocation	Request	Available
	A B C	A B C	A B C
P0	0 1 0	0 0 0	0 0 0
P1	2 0 0	2 0 2	
P2	3 0 3	0 0 0	
P3	2 1 1	1 0 0	
P4	0 0 2	0 0 2	

For above request algorithm leaves system in safe state with safe sequence <P0, P2, P3, P4, P1>. Suppose now that process P_2 makes one additional request for an instance of type C. The Request matrix is modified as follows, which leaves the system in unsafe state.

Request
A B C
0 0 0
2 0 2
0 0 1
1 0 0
0 0 2

	Allocation	Request	Available
	A B C	A B C	A B C
P0	0 1 0	0 0 0	0 0 0
P1	2 0 0	2 0 2	
P2	3 0 3	0 0 0	
P3	2 1 1	1 0 0	
P4	0 0 2	0 0 2	

1. In this, Work = [0, 0, 0] &
   Finish = [false, false, false, false, false]
2. *i=0* is selected as both Finish[0] = false and [0, 0, 0]<=[0, 0, 0].
3. Work =[0, 0, 0]+[0, 1, 0] =>[0, 1, 0] &
   Finish = [true, false, false, false, false].
4. *i=2* is selected as both Finish[2] = false and [0, 0, 0]<=[0, 1, 0].
5. Work =[0, 1, 0]+[3, 0, 3] =>[3, 1, 3] &
   Finish = [true, false, true, false, false].
6. *i=1* is selected as both Finish[1] = false and [2, 0, 2]<=[3, 1, 3].
7. Work =[3, 1, 3]+[2, 0, 0] =>[5, 1, 3] &
   Finish = [true, true, true, false, false].
8. *i=3* is selected as both Finish[3] = false and [1, 0, 0]<=[5, 1, 3].
9. Work =[5, 1, 3]+[2, 1, 1] =>[7, 2, 4] &
   Finish = [true, true, true, true, false].
10. *i=4* is selected as both Finish[4] = false and [0, 0, 2]<=[7, 2, 4].
11. Work =[7, 2, 4]+[0, 0, 2] =>[7, 2, 6] &
    Finish = [true, true, true, true, true].
12. Since Finish is a vector of all true it means **there is no deadlock** in this example.

Recovery from Deadlock: When a detection algorithm determines that a deadlock exists, several alternativs are available to recover from deadlock. 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 One is simply to abort one or more processes to break the circular wait. The other is to preempt some resources from one or more of the deadlocked processes.

1. **Process Termination:**
   - Abort all deadlocked processes: This method clearly will break the deadlock cycle, but at great expense; the deadlocked processes may have computed for long time, and result 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.
2. **Resource Preemption:** To eliminate deadlocks using resource preemption, successively preempt some resources from processes and give these resources to othee processes until the deadlock cycle is broken.