Module 3

Process Synchronization

Process Synchronization

Background :

- A cooperating process is one that can either affect or get affected by the other processes that are in the system.
- These processes can either be authorized to share data only through files or can directly share a memory region or messages.
- In this module, we discuss several mechanisms to make sure the orderly execution of cooperating processes that share a rational address space, so that data stability is maintained.

Process Synchronization

Mutual exclusion :

- A method of making sure that if one process is accessing shared modifiable data, the other processes will be excluded from doing the similar thing.

- Formally, while one process implements the shared variable, all other processes craving at the same time must be kept waiting; when that process is done executing the shared variable, one of the processes waiting; to do so it must be allowed to proceed

Process Synchronization

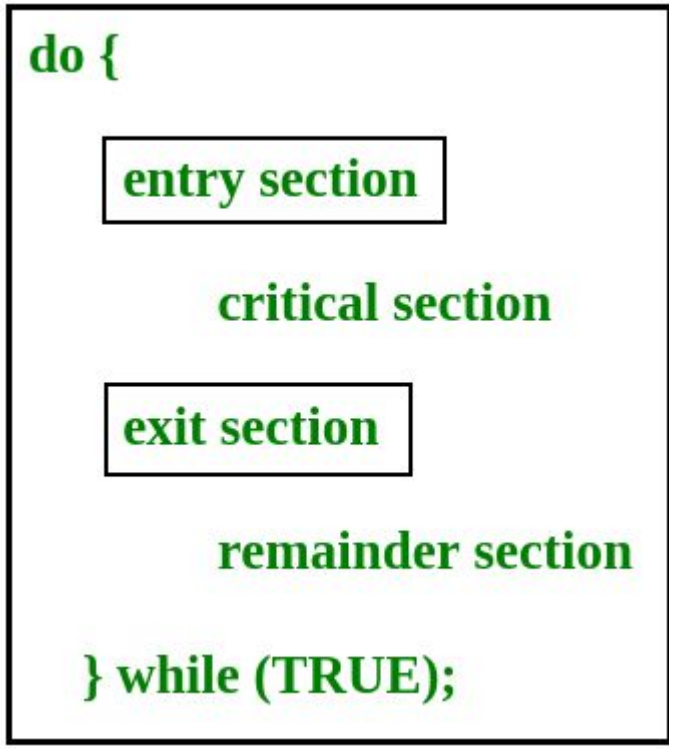
The solutions described above can be used to build the synchronization primitives below:

- Locks
- Readers–writer locks
- Recursive locks
- Semaphores
- Monitors
- Message passing
- Tuple space

The critical section problem

- Consider a system comprising of n processes {P0, P1, ..., Pn-1}.
- Each process has a section of code, called as **critical section**, in which the process may **be varying updating a table, writing a file, common variables** and so on.
- **The essential feature of the system is that, when one process is performing in its critical section, no other process is to be permitted to perform in its critical section.**
- The section of code executing this request is the entry section. The critical section is pursued by an exit section. The remaining code is the remainder section.

The critical section problem



The critical section is a code segment where the shared variables can be accessed. An atomic action is required in a critical section i.e. only one process can execute in its critical section at a time.
All the other processes have to wait to execute in their critical sections.

Solution for Critical Section Problem

- **Mutual exclusion**: If process P1 is performing in its critical section, then no other processes can be executing in their critical sections.
- **Progress**: If no process is performing in its critical section and some processes wish to enter their critical sections, then only those processes that are not performing in their remainder sections can take part in the decision on which will enter its critical section next, and this selection cannot be postponed indefinitely.
- **Bounded waiting**: There exists a limit or bound, on the number of times that other processes are permitted to enter their critical sections after a process has made a request to go in its critical section and before that request is approved

Solution for Critical Section Problem

Preemptive and Non-pre-emptive kernels:

- A pre-emptive kernel permits a process to be pre-empted while it is running in kernel mode.
- A non-preemptive kernel prevents a process running in kernel mode to be pre-empted; a kernel-mode process will run till it exits kernel mode, blocks, or willingly yields control of the CPU.

Solution for Critical Section Problem

Peterson's Solution :

- It provides a decent algorithmic description of solving the critical-section problem.
- Peterson's solution is limited to two processes that substitute execution between their remainder sections and critical sections.

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

- **Peterson's solution :** requires the two processes to share two data items –

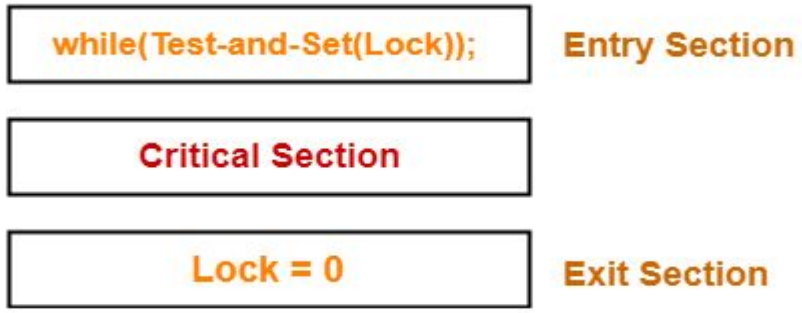
```
int turn;
boolean flag[2];
```

- Variable *turn* – indicates whose turn it is to enter its critical section, if turn == i, then process Pi is allowed to execute in its critical section
- *flag* – indicates if a process is ready to enter in its critical section, if flag[i] is true then, then process Pi is ready to enter its critical section.
- To enter the critical section, process Pi, first sets turn (turn=i) to be true and then sets flag to the value j, thereby declaring that if the other process wishes to enter the critical section, it can do so.
- If both processes try to pass in 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 take place but will be overwritten immediately.

Synchronization Hardware :

- Test and Set Lock (TSL) is a synchronization mechanism.
- It uses a test and set instruction to provide the synchronization among the processes executing concurrently.

It is implemented as-



Initially, lock value is set to 0.

- Lock value = 0 means the critical section is currently vacant and no process is present inside it.
- Lock value = 1 means the critical section is currently occupied and a process is present inside it.

- **Semaphores** :A semaphore in basic form is a preserved integer variable that can assist and inhibit approach to shared resources in a multiprocessing circumstance.

- A semaphore X is an integer variable that is different from initialization, is accessed only via two standard atomic operations such as **wait ()** and **signal ()**.

- The **wait()** operation was originally named as **P** (from the Dutch problem, "to test"); **signal ()** was originally termed as **V** (from verhogen, "to increment").

Solution for Critical Section Problem Semaphores

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

The definition of signal () is as follow:

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

- All the alteration to the integer value of the semaphore in the wait () and signal() operations must be carried out inseparably.
- That is, when one process adjusts the semaphore value, no other process can concurrently change that same semaphore value.
- In the case of wait(X), the integer value of X i.e. X < 0, and its probable change (X--), can be carried out without interruption.
- The two most basic kinds of semaphores are **counting semaphores** and **binary semaphores**.
- Counting semaphores describe multiple resources which value can range over an unregulated domain.
- The binary semaphores describe two possible states and the value of this semaphore

Classic problems of synchronization

The Bounded-Buffer Problem:

- We assume that the pool contains n buffers, each capable of holding one item.
- The mutex semaphore offers mutual exclusion for accesses to the buffer pool and is initialized to the value **1**.
- The **full** and **empty** semaphores count the number of empty and full buffers.
- The semaphore empty is initialized to the value **n**; the semaphore full is reset to the value **0** for a **producer process**.
- The semaphore empty is initialized to the value **full**; the semaphore full is reset to the value **n** for a **consumer process**.

- **Producer Process**

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

Consumer Process

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

Classic problems of synchronization

The Readers-Writers Problem:

- A database is shared among various process running concurrently either to read or update.
- The processes performing read and write are called readers and writers simultaneously.
- Problems arise only when **two process write at the same time**. To solve the problems, **writers are given an exclusive access** to the shared database.
- This synchronization problem is denoted as the readers-writers problem.

- The simplest one is that none of the readers should wait for other readers to finish just because a writer is waiting.
- The second one is that, once a writer is ready, that writer performs its write as fast as possible.

Writer Process

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

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);
```

Classic problems of synchronization

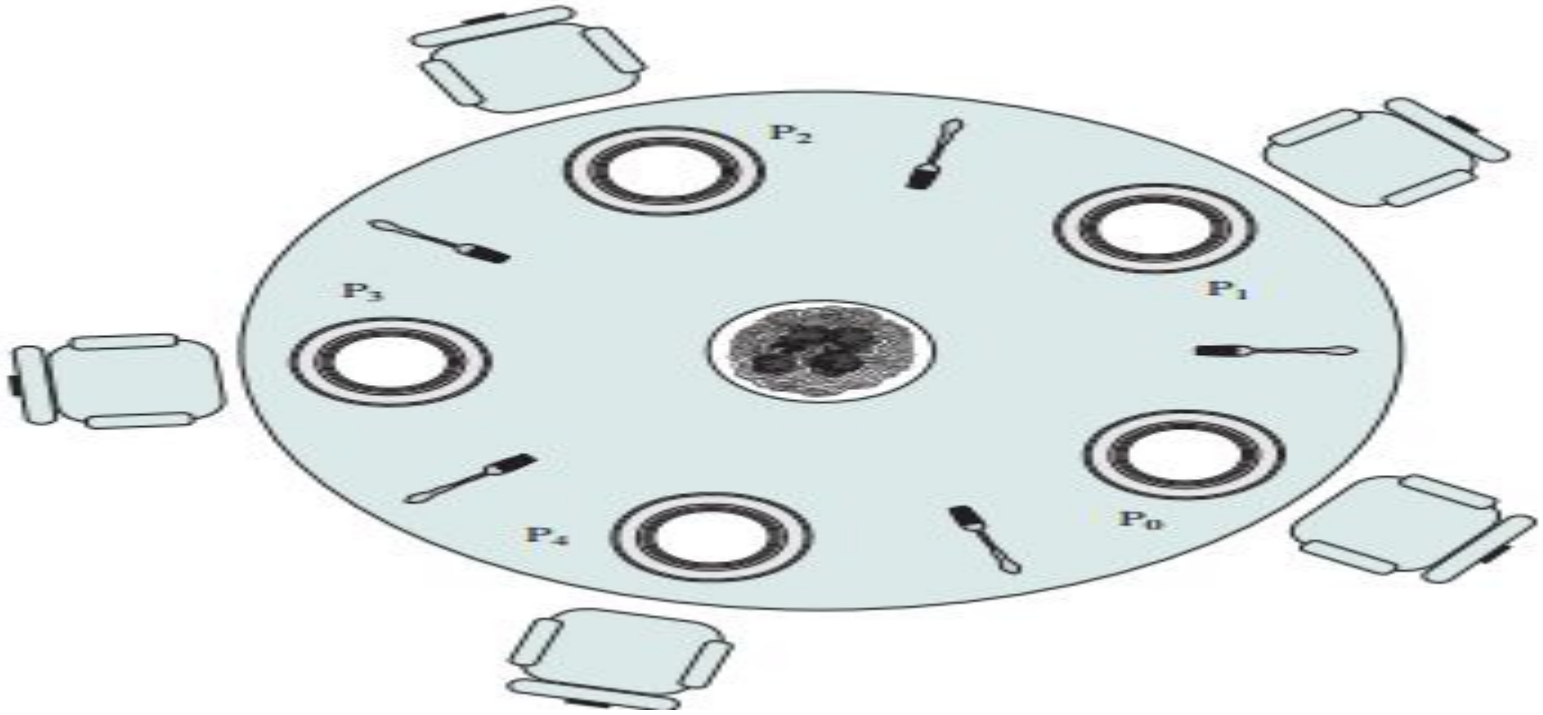
Readers-writer locks are very useful in the following situations:

a. In applications where it is easy to recognise which processes only writes shared data and which threads only reads shared data.

b. In applications that have more readers than writers. This is because reader writer locks generally need more overhead to establish than semaphores or mutual exclusion locks, and the overhead for setting up the reader-writer lock is compensated by the increased concurrency of allowing multiple readers.

Classic problems of synchronization

The Dining-Philosophers Problem:



The Dining-Philosophers Problem:

- Consider five philosophers who can eat and think, sharing a circular table with five chairs, each belonging to one philosopher.
- The table has a bowl of rice, and the table is arranged with five single chopsticks.
- When a philosopher thinks, she does not interact with her colleagues.
- From time to time, a philosopher becomes 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 and she cannot collect a chopstick that is already in the hand of a neighbor.
- When a hungry philosopher has both her chopsticks simultaneously, she eats without releasing her chopsticks.
- When she is finished eating, she puts down both of her chopsticks and again starts thinking.

- One simple solution is to signify 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 suitable semaphores.

1. The structure of philosopher is shown below:

```
do {
        wait (chopstick [a] ) ,-
        wait(chopstick [ (a + 1) % 5] ) ;
        // eat
        signal(chopstick [a]);
        signal(chopstick [(a + 1) % 5]);
        / / think
}while (TRUE);
```

- Although this solution guarantees that no two neighbors are eating simultaneously, it nevertheless must be rejected because it could create a deadlock.
- Suppose that all five philosophers become hungry at the same time and each grabs her left chopstick.
- Now all the elements of chopstick will be equal to 0. When each philosopher tries to grab her right chopstick, she will be delayed forever. This leads to a deadlock.

- Let 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).
- Using an asymmetric solution; that is, an odd philosopher picks up first her left chopstick and then her exact chopstick, whereas an even philosopher picks up her right chopstick and then her left chopstick.

Deadlocks

- 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***.

System Model

- A system consists of a fixed number of resources to be circulated among a number of competing processes.
- The resources are divided into several types, each consisting of some number of identical instances. Memory space, files, and I/O devices, CPU cycles are the examples of resource types.

Deadlocks

In the normal mode of operation, a process may use a resource in only the following sequence:

- **Request**: If the request cannot be approved immediately, then the requesting process must wait till it can obtain the resource.
- **Use**: the process can work on the resource.
- **Release**: The process releases the resource.

Deadlocks

- The request and release of resources are called system calls.
- Examples are: the open () and close file (), request () and release (), device (), and allocate () and free ().
- Request and release of resources that are not managed properly by the operating system can be achieved through the wait () and signal () operations on semaphores or over acquisition and release of a mutex lock.
- For each and every usage of a kernel managed resource through a processor thread, the operating system checks to be sure that the process has requested and has been allotted the resource.

Deadlocks

- A system table records if each resource is free or allotted for each resource that is allocated, the table records the process to which it is allocated.
- If a process requests a resource that is presently allocated to another process, it can be added to a queue of processes waiting for this resource.
- A set of processes is in a **deadlock state** when each process in the set is to come for an event that can be produced only by another process in the set.
- The events with which we are mostly concerned here are resource acquisition and release. The resources maybe either physical resources (for example, printers, tape drives) or logical resources (for example, files, semaphores).

Deadlock Example

- To explain a deadlock state, let’s consider a system with three CD RW drives. Suppose each of three processes grips one of these CD-RW drives. If each process requests another drive, the three processes will be in a deadlock state. Each is waiting for the event "CD RW is released," which can be done only by one of the other waiting processes.
- Deadlocks may also include different resource types. For an example, let’s take a system with one printer and one DVD drive. Suppose the process P1 is holding the DVD and process P2 is holding the printer. If P1 requests the printer and P2 requests the DVD drive, a deadlock happens.

Deadlock Characterization

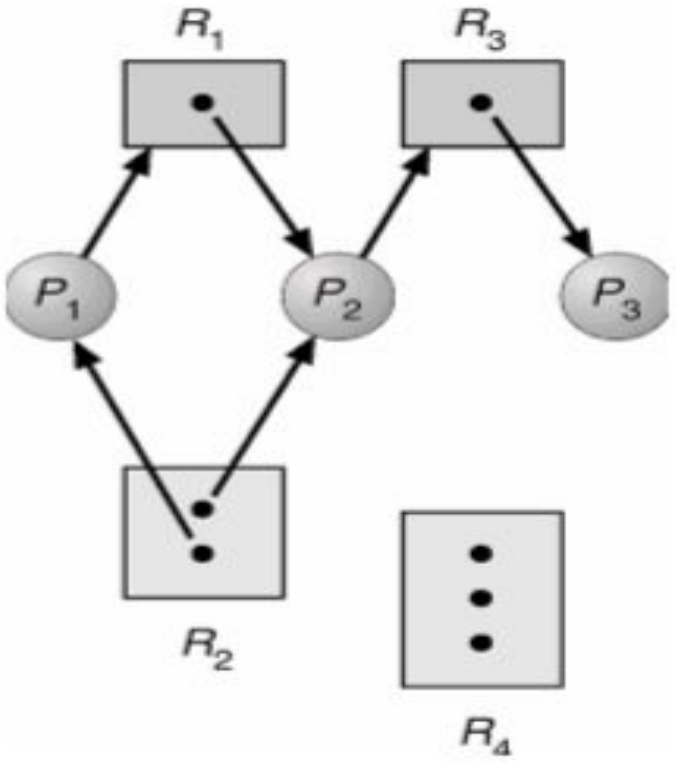
Essential Conditions: A deadlock situation can arise if the following four conditions hold simultaneously in a system:

- **Mutual exclusion**: At least one resource should be held in a non-sharable mode; that means, only one process can use the resource at a time. If another process requests that resource, the requesting process must be delayed till the resource has been released.
- **Hold and wait:** A process should hold at least one resource and should wait to get additional resources that are presently being held by other processes.
- **No pre-emption**: Resources cannot be pre-empted; that is, a resource can be released only freely by the process holding it, after that process has completed its task.
- **Circular wait**: A set $\{P_0, P_1, ..., P_n\}$ of waiting processes must exist such that P_0 is waiting for a resource held by P_1, P_1 is waiting for a resource held by P_2,…., P_{n-1} is waiting for a resource held by P_n, and P_n is waiting for a resource held by P_0

Resource-Allocation Graph

- Deadlocks can be described more specifically in terms of a directed graph called a system resource-allocation graph.
- This graph includes a set of vertices V and a set of edges E.
- The set of vertices V is divided into two different types of nodes: P - {Pi, Pi,,..,Pn}, the set involving of all the active processes in the system, and R = {R[, R2,…,Rm }, the set involving of all resource types in the system.

Resource-Allocation Graph



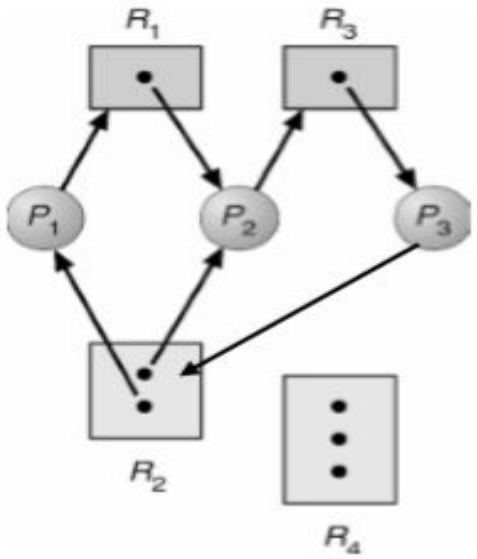
Resource-Allocation Graph

- **REQUEST**: If the process Pi does not have any outstanding request, then it can simultaneously request any number of resources R1...Rm.
- **ACQUISITION**: If the process Pi comprises of any outstanding requests, then all those requests can be satisfied simultaneously.
- **RELEASE**: If the process Pi does not have any outstanding request, then the held resources can be released.
- Here, in the figure two minimal cycles exist

in the system:

P1-> R1-> P2 -> R3 -> P3 -> R2->P1

P2-> R3-> P3-> R2-> P2



Resource Allocation with a cycle and no deadlock

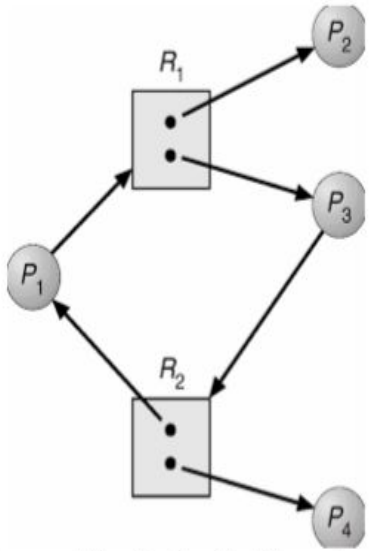
- The resource-allocation graph in the below diagram. In this example, we also have a cycle. P1-> R1-> P3->R2->P1
- However, there is no deadlock. See that process P4 may release its instance of resource type R2. That resource can then be assigned to P3, breaking the cycle.
- In *short, if a resource-allocation graph*

does not have a cycle, then the system is

not in a deadlocked state.

- *If there is a cycle, then the system may or may*

not be in a deadlocked state.



Methods of handling deadlocks

- **Deadlock Prevention and Avoidance**:

Make sure that the system never enters into the deadlock state.

- **Deadlock Detection and Recovery**:

If any deadlock has occurred, detect it and make sure to recover that deadlock.

- **Deadlock Ignorance**:

Ignoring the fact that there would not be a deadlock.

Deadlock Prevention

- For a deadlock to occur, the four necessary conditions i.e. Mutual exclusion, No preemption, Hold and wait, and Circular wait must hold. By ensuring that minimum one of these conditions does not hold, we can prevent the incident of a deadlock.
- **Mutual Exclusion**: The mutual-exclusion situation must hold for non-sharable resources. For instance, a printer cannot be concurrently shared by several processes. Sharable resources, indifference, do not need mutually exclusive access and thus cannot be engaged in a deadlock.

Hold and Wait: To guarantee that the hold-and-wait condition never happens in the system, we must ensure that, whenever a process demands a resource, it does not hold any other extra resources.

- One protocol that can be used needs each process to demand and be allocated all its resources before it initiates execution.
- An alternative protocol permits a process to request resources only when it has none.
- *A process may demand some resources and also use them. Before it can demand any additional resources, still, it must release resources entirely that it is presently allocated.*

No Preemption:

- The third essential condition for deadlocks is that there should not be any preemption of resources that have previously been allocated.
- To confirm that this condition does not hold, we can use the below protocol.
- If a process is holding few resources and needs another resource that cannot be instantly allocated to it (that is, the process need to wait), then all resources presently being held are preempted

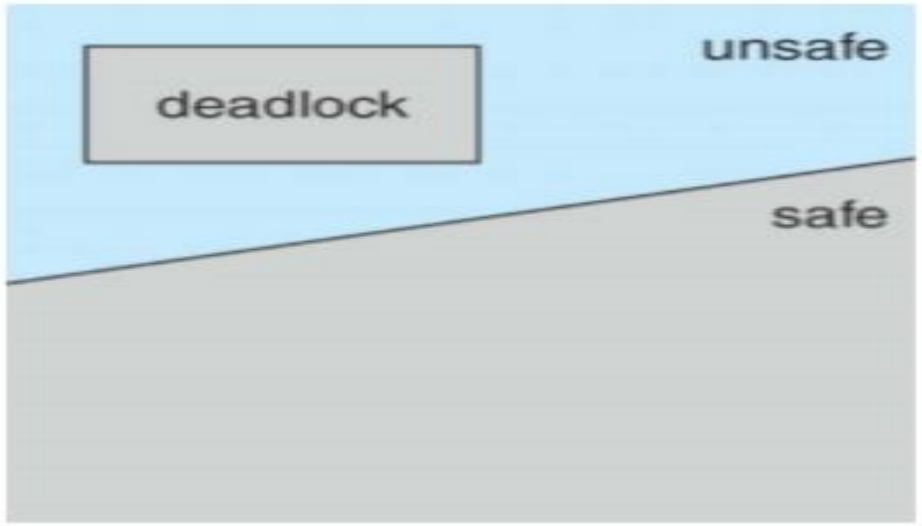
Circular Wait: To prevent circular wait, order the resources and make sure that each process can utilize those resources in the order

Deadlock Avoidance

- As most of the prevention algorithms have poor utilization of the resources, it reduces the throughputs.
- A better way is to avoid the deadlocks by knowing the resource usage by the processes and also knowing about the resources allocated and available, and future releases and requests by all the processes.
- A system that is in a **safe state** avoids to get into an **unsafe state** and thereby avoiding the deadlock situation. In an unsafe state, the deadlocks are bound to happen.
- *To ensure a safe state, there should be a safe sequence of the processes and the resources must be properly allocated.*

Banker's Algorithm for deadlock avoidance

- When there are multiple instances for the resources then RAG is not useful. For such cases, Banker's algorithm is used.



Banker's Algorithm for deadlock avoidance

- When a new process arrives the system, it must state the maximum number of instances of every resource type that it may need.
- This number may not surpass the total number of resources in the system.
- When a user demands a group of resources, the system must examine whether the distribution of these resources will leave the system in a safe state.
- If it will, the resources are assigned; otherwise, the process must wait till some other process releases sufficient resources.

Banker’s Algorithm for deadlock avoidance

- Several data structures must be managed to implement the banker's algorithm. These data structures encrypt the state of the resource-allocation system.
- Consider n be the number of processes in the system and m is the number of resource types. We need the subsequent data structures:
- **Available**: A vector of dimension m indicates the number of obtainable resources of every type. If Available[j] equals to k, there are k instances of resource type Ri obtainable.

Banker's Algorithm for deadlock avoidance

- **Allocation**: An n x m matrix describes the number of resources of every type currently assigned to each process. If Allocation[i][j] equals to k, then process Pi is presently allocated k instances of Rj resource type.
- **Max**: An n x m matrix describes the maximum request of each process. If Max[i][j] equals to k, then process Pi may request maximum k instances of resource type Rj.
- **Need**: An n x m matrix specifies the remaining resource need of each process. If Need[i][j] equals to k, then process Pi, may need k more instances of Rj resource type to complete its task.
- Note that **Need[i][j] = Max[i][j] - Allocation[i][j].**

Safety Algorithm

- The algorithm for finding out whether or not a system is in a safe state can be described as follows.

1) Let Work and Finish be vectors of length 'm' and 'n' respectively.
Initialize: Work = Available
Finish[i] = false; for i=1, 2, 3, 4....n

2) Find an i such that both
a) Finish[i] = false
b) $Need_i$ <= Work
if no such i exists goto step (4)

3) Work = Work + Allocation[i]
Finish[i] = true
goto step (2)

4) if Finish [i] = true for all i
then the system is in a safe state

Resource-Request Algorithm

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

1) If $Request_i <= Need_i$
Goto step (2) ; otherwise, raise an error condition, since the process has exceeded its maximum claim.

2) If $Request_i <= Available$
Goto step (3); otherwise, P_i must wait, since the resources are not available.

3) Have the system pretend to have allocated the requested resources to process Pi by modifying the state as follows:
Available = Available − Requesti
$Allocation_i = Allocation_i + Request_i$
$Need_i = Need_i - Request_i$

Banker's Algorithm for deadlock avoidance

- Considering a system with five processes P_0 through P_4 and three resources of type A, B, C.

Process	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	

Banker's Algorithm for deadlock avoidance

Question1. What will be the content of the Need matrix?

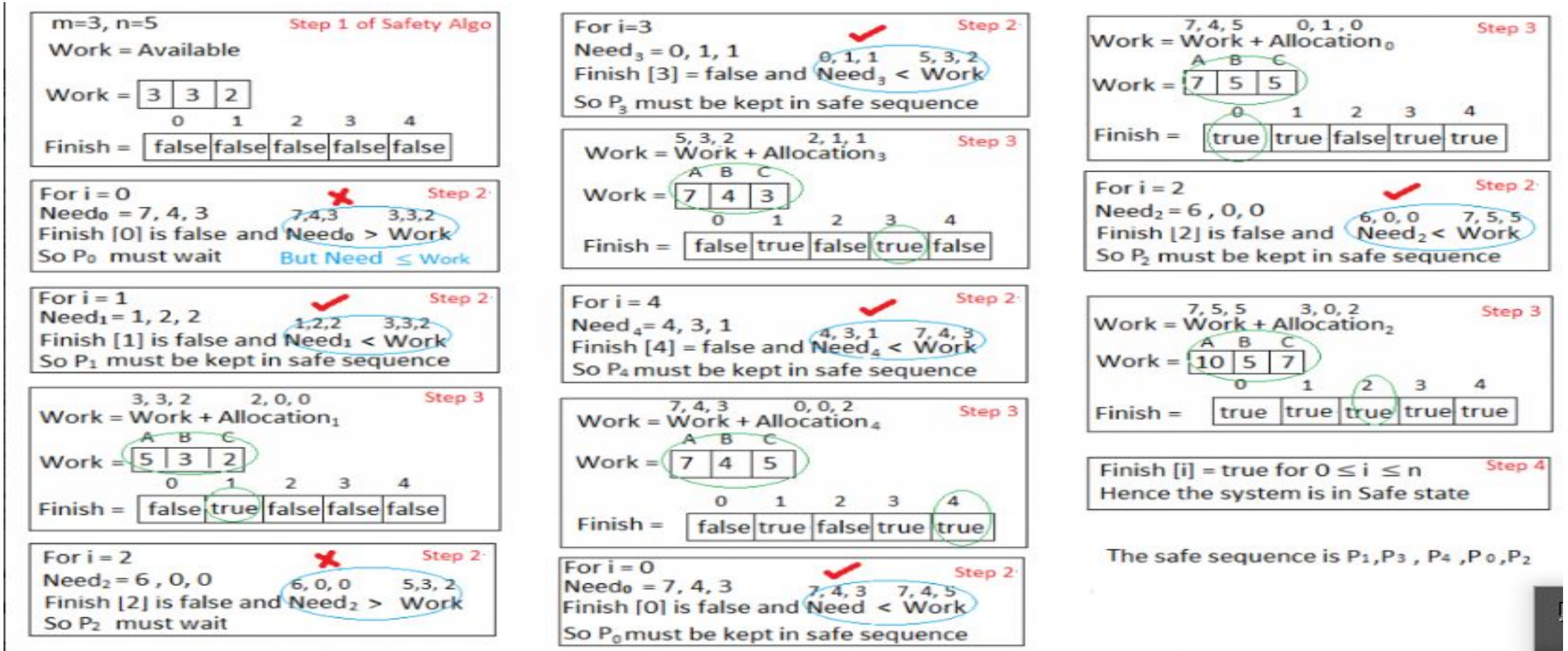
Need [i, j] = Max [i, j] – Allocation [i, j]

So, the content of Need Matrix is:

Process	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

Question2. Is the system in a safe state? If Yes, then what is the safe sequence?

Applying the Safety algorithm on the given system,



Question3. What will happen if process P_1 requests one additional instance of resource type A and two instances of resource type C?

A B C

$Request_1 = 1, 0, 2$

To decide whether the request is granted we use Resource Request algorithm

Step 1

$Request_1 (1, 0, 2) < Need_1 (1, 2, 2)$ ✓

Step 2

$Request_1 (1, 0, 2) < Available (3, 3, 2)$ ✓

Step 3

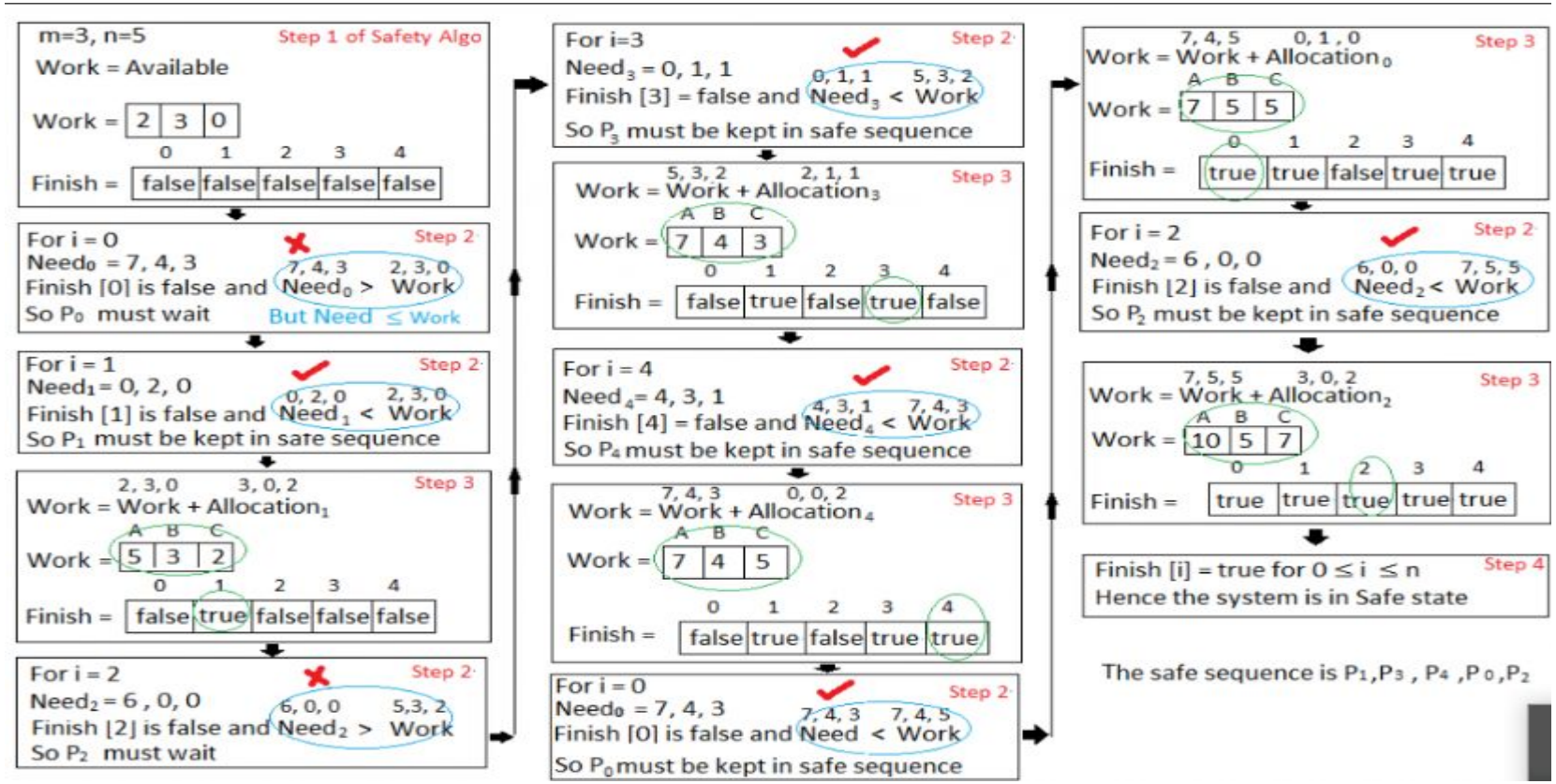
$Available = Available - Request_1$

$Allocation_1 = Allocation_1 + Request_1$

$Need_1 = Need_1 - Request_1$

Process	Allocation	Need	Available
	A B C	A B C	A B C
P_0	0 1 0	7 4 3	2 3 0
P_1	3 0 2	0 2 0	
P_2	3 0 2	6 0 0	
P_3	2 1 1	0 1 1	
P_4	0 0 2	4 3 1	

- We must determine whether this new system state is safe. To do so, we again execute Safety algorithm on the above data structures.



Deadlock Detection

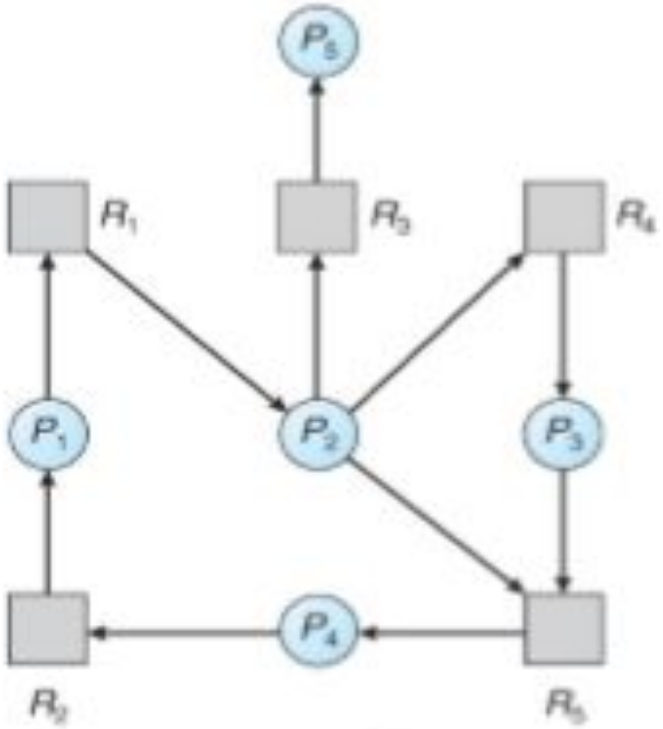
- If a system does not work either a deadlock-prevention or a deadlock avoidance algorithm, then a deadlock situation may arise. In this situation, the system must provide:
  - An algorithm that inspects the state of the system to determine if a deadlock has occurred
  - An algorithm to recover from the deadlock .

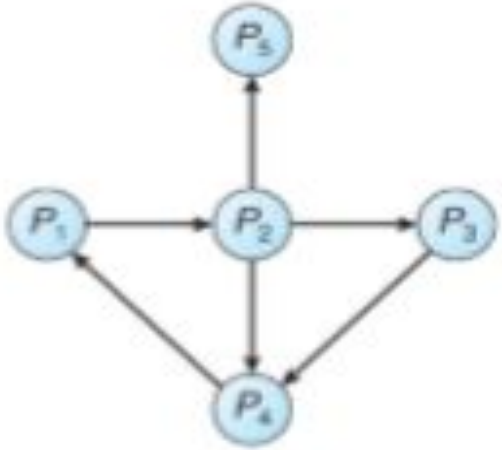
Deadlock Detection

Single Instance of Each Resource Type:

- If all resources have only a **single instance**, then we can describe a deadlock detection algorithm that employs a variant of the resource allocation graph, called a wait-for graph.
- We can develop this graph from the resource allocation graph by eliminating the resource nodes and collapsing the proper edges, as shown in Figure 2.2.6.
- More specifically, an edge from Pi, to Pj, in a wait-for graph indicates that process Pi is waiting for process Pj to release a resource that Pi requires.
- An edge Pi->Pj remains in a wait-for graph if and only if the corresponding resource allocation graph includes two edges Pi —>Rq and Rq-> Pj for some resource Rq.
- For example, we show a resource allocation graph and the corresponding wait-for graph.

RAG Vs WFG in deadlock detection





Deadlock Detection

Several Instances of a Resource Type: The wait-for graph pattern is not valid to a resource-allocation system with multiple instances of each resource type. Now we turn to a deadlock detection algorithm that is appropriate to such system. The algorithm works several time-varying data structures that are similar to those used in the algorithm of bankers.

a. **Available**: A vector of length m shows the number of available resources of each type.

b. **Allocation**: An n x m matrix describes the number of resources of each type presently allocated to each process.

Recovery from Deadlocks

- When a detection algorithm decides that a deadlock exists, some alternatives are available.
- One option is to inform the operator that a deadlock has been occurred and to operator deals with the deadlock manually.
- Another one is the possibility to let the system recuperate from the deadlock automatically.

There are two options to break a deadlock.

- One is simply to terminate one or more processes to break the circular wait.
- The other is to pre-empt some resources from one or more of the deadlocked processes

Recovery from Deadlocks

Process Termination

- **Abort all deadlocked processes:** This method breaks the deadlock cycle clearly, but is expensive; the deadlocked processes may have calculated for a long time, and the end results of these partial computations should be cancelled and probably will have to compute it later.
- **Abort one process at a time till the deadlock cycle is eliminated**: This method gains considerable overhead, then, after each process is aborted, a deadlock-detection algorithm must be raised to determine or regulate whether any processes are still deadlocked.

Recovery from Deadlocks

Resource Pre-emption:

If pre-emption is essential to deal with deadlocks, then three issues need to be spoken.

- ***Selecting a victim***: Which processes and which resources are to be pre-empted? As in process termination, we should dictate the order of pre-emption to reduce cost. Cost factors may involve such parameters as the number of resources a deadlocked process is holding and the time the process has consumed during its execution.
- ***Rollback*:** If we pre-empt a resource from a process, what must be done with that process? It cannot endure with its normal execution. It is missing some necessary resource. We must roll back the process and restart it from the safe state.

Recovery from Deadlocks

- ***Starvation***: How do we confirm that starvation will not occur? That is how can we guarantee that resources will not always be pre-empted from the same process?
- We must confirm that a process can be chosen as a victim an only finite number of times. The most common solution is to involve the number of rollbacks in the cost factor.