UNIT 3- PROCESS MANAGEMENT

Process Synchronization was introduced to handle problems that arose while multiple process executions. Process is categorized into two types on the basis of synchronization and these are given below:

- Independent Process
- Cooperative Process

Independent Processes: Two processes are said to be independent if the execution of one process does not affect the execution of another process.

Cooperative Processes: Two processes are said to be cooperative if the execution of one process affects the execution of another process. These processes need to be synchronized so that the order of execution can be guaranteed.

Process Synchronization

It is the task phenomenon of coordinating the execution of processes in such a way that no two processes can have access to the same shared data and resources.

- It is a procedure that is involved in order to preserve the appropriate order of execution of cooperative processes.
- In order to synchronize the processes, there are various synchronization mechanisms.
- Process Synchronization is mainly needed in a multi-process system when multiple processes are running together, and more than one processes try to gain access to the same shared resource or any data at the same time.

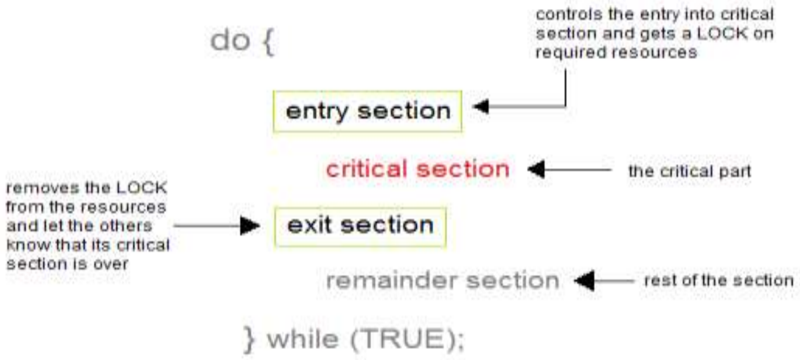
Race Condition

- It occurs when two or more processes are executed at the same time.
- It is not scheduled in proper sequence and not executed in critical section correctly, which causes ‘ Data inconsistency and Data Loss.
- At the time when more than one process is either executing the same code or accessing the same memory or any shared variable. In that condition, there is a possibility that the output or the value of the shared variable is wrong so for that purpose all the processes are doing the race to say that my output is correct. This condition is commonly known as **a race condition.** As several processes access and process the manipulations on the same data in a concurrent manner and due to which the outcome depends on the particular order in which the access of data takes place.
- Mainly this condition is a situation that may occur inside the **critical section**. Race condition in the critical section happens when the result of multiple thread execution differs according to the order in which the threads execute. But this condition is critical sections can be avoided if the critical section is treated as an atomic instruction. Proper thread synchronization using locks or atomic variables can also prevent race conditions.

Critical Section Problem

- The portion of program text where shared resources will be placed is called Critical section.It cannot be accessed easily.

- A Critical Section is a code segment that accesses shared variables and has to be executed as an atomic action. It means that in a group of cooperating processes, at a given point of time, only one process must be executing its critical section. If any other process also wants to execute its critical section, it must wait until the first one finishes.
- The entry to the critical section is mainly handled by wait() function while the exit from the critical section is controlled by the signal() function.


do {
controls the entry into critical section and gets a LOCK on required resources
entry section
critical section
the critical part
removes the LOCK from the resources and let the others know that its critical section is over
exit section
remainder section
rest of the section
} while (TRUE);

Entry Section : In this section mainly the process requests for its entry in the critical section.

Exit Section : This section is followed by the critical section.

Requirements or Criteria for solution to the Critical Section Problem

A solution to the critical section problem must satisfy the following four conditions:

1. Mutual Exclusion

Only one process is allowed in critical section at any point of time.Out of a group of cooperating processes, only one process can be in its critical section at a given point of time.

2. Progress

No process running outside the critical section should lock the other processes for entering in critical section.If no process is in its critical section, and if one or more threads want to execute their critical section then any one of these threads must be allowed to get into its critical section.

3. Bounded Waiting

After a process makes a request for getting into its critical section, there is a limit for how many other processes can get into their critical section, before this process's request is granted. So after the limit is reached, the system must grant the process permission to get into its critical section.

4. No assumption regarding no of CPUs and the speed of hardware.

Solutions for the Critical Section

The critical section plays an important role in Process Synchronization so that the problem must be solved.

Some widely used method to solve the critical section problem are as follows:

1. Peterson's Solution

Peterson’s solution provides a good algorithmic description of solving the critical-section problem and illustrates some of the complexities involved in designing software that addresses the requirements of mutual exclusion, progress, and bounded waiting. This is widely used and software-based solution to critical section problems. Peterson's solution was developed by a computer scientist Peterson that's why it is named so.

With the help of this solution whenever a process is executing in any critical state, then the other process only executes the rest of the code, and vice-versa can happen. This method also helps to make sure of the thing that only a single process can run in the critical section at a specific time.

This solution preserves all three conditions:

- Mutual Exclusion is comforted as at any time only one process can access the critical section.
- Progress is also comforted, as a process that is outside the critical section is unable to block other processes from entering into the critical section.
- Bounded Waiting is assured as every process gets a fair chance to enter the Critical section.

The above shows the structure of process **Pi in Peterson's solution.**

- Suppose there are **N processes (P1, P2, ... PN)** and as at some point of time every process requires to enter in the **Critical Section**
- A **FLAG[]** array of size N is maintained here which is by default false. Whenever a process requires to enter in the critical section, it has to set its flag as true. Example: If Pi wants to enter it will set **FLAG[i] =TRUE.**
- Another variable is called **TURN** and is used to indicate the process number that is currently waiting to enter into the critical section.
- The process that enters into the critical section while exiting would change the **TURN** to another number from the list of processes that are ready.
- Example: If the turn is 3 then P3 enters the Critical section and while exiting turn=4 and therefore P4 breaks out of the wait loop.

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

Disadvantage

- Peterson's solution works for two processes, but this solution is best scheme in user mode for critical section.
- This solution is also a busy waiting solution so CPU time is wasted. So that **"SPIN LOCK"** problem can come. And this problem can come in any of the busy waiting solution.

Semaphore

A semaphore is a variable that indicates the number of resources that are available in a system at a particular time and this semaphore variable is generally used to achieve the process synchronization. It is generally denoted by "**S**". You can use any other variable name of your choice.

A semaphore uses two functions i.e. ***wait()*** and ***signal()***. Both these functions are used to change the value of the semaphore but the value can be changed by only one process at a particular time and no other process should change the value simultaneously.

The ***wait()*** function is used to decrement the value of the semaphore variable "**S**" by one if the value of the semaphore variable is positive. If the value of the semaphore variable is 0, then no operation will be performed.

```
wait(S) {

   while (S == 0); //there is ";" sign here

   S--;

}
```

The ***signal()*** function is used to increment the value of the semaphore variable by one.

```
signal(S) {

   S++;

}
```

Types of Semaphores

There are two main types of semaphores i.e. counting semaphores and binary semaphores. Details about these are given as follows −

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

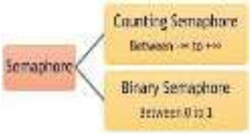
```
struct Semaphore {
   int value;
 }
P(Semaphore s)
{
   s.value = s.value - 1;
   if (s.value < 0) {

      // add process to queue
      // here p is a process which is
currently executing
      q.push(p);
      block();
   }
   else
      return;
}
```

```
V(Semaphore s)
{
   s.value = s.value + 1;
   if (s.value >= 0) {

      // remove process p from queue
      Process p=q.pop();
      wakeup(p);
   }
   else
      return;
}
```

- **Binary Semaphores**
  The binary semaphores are like counting semaphores but their value is restricted to 0 and 1. The wait operation only works when the semaphore is 1 and the signal operation succeeds when semaphore is 0. It is sometimes easier to implement binary semaphores than counting semaphores.

```
struct semaphore {
   enum value(0, 1);
 }
P(semaphore s)
{
   if (s.value == 1) {
      s.value = 0;
   }
   else {
      // add the process to the waiting
queue
      q.push(P)
      sleep();
   }
}
```

```
V(Semaphore s)
{
   if (s.q is empty) {
      s.value = 1;
   }
   else {

      // select a process from waiting
queue
      Process p=q.pop();
      wakeup(p);
   }
}
```



Types of Semaphore

Advantages of Semaphores

Some of the advantages of semaphores are as follows −

Semaphores allow only one process into the critical section. They follow the mutual exclusion principle strictly and are much more efficient than some other methods of synchronization.

There is no resource wastage because of busy waiting in semaphores as processor time is not wasted unnecessarily to check if a condition is fulfilled to allow a process to access the critical section.

Semaphores are implemented in the machine independent code of the microkernel. So they are machine independent.

Disadvantages of Semaphores

Some of the disadvantages of semaphores are as follows −

- Semaphores are complicated so the wait and signal operations must be implemented in the correct order to prevent deadlocks.
- Semaphores are impractical for last scale use as their use leads to loss of modularity. This happens because the wait and signal operations prevent the creation of a structured layout for the system.
- Semaphores may lead to a priority inversion where low priority processes may access the critical section first and high priority processes later

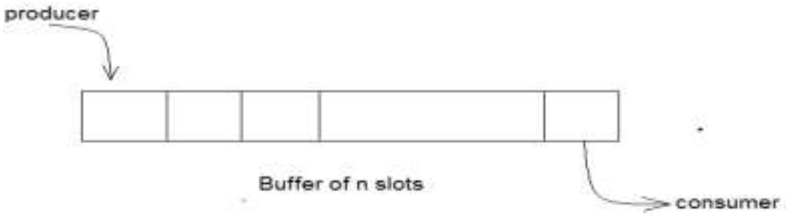
Example: A Counting Semaphore was initialized to 12. then 10P (wait) and 4V (Signal) operations were computed on this semaphore. What is the result?

1. S = 12 (initial)
2. 10 p (wait) :
3. SS = S -10 = 12 - 10 = 2
4. then 4 V :
5. SS = S + 4 =2 + 4 = 6 Answer.

Classical Problem of Synchronization

What is the Problem Statement?

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.

The producer consumer problem is a synchronization problem. There is a fixed size buffer and the producer produces items and enters them into the buffer. The consumer removes the items from the buffer and consumes them.

A producer should not produce items into the buffer when the consumer is consuming an item from the buffer and vice versa. So the buffer should only be accessed by the producer or consumer at a time.

The producer consumer problem can be resolved using semaphores.

Bounded Buffer Solution Using Semaphores

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

- m, 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.

The codes for the producer and consumer process are given as follows

Producer Process

The code that defines the producer process is given below −

```
do {
   .
   . PRODUCE ITEM
   .
   wait(empty);
   wait(mutex);
   .
   . PUT ITEM IN BUFFER
   .
   signal(mutex);
   signal(full);

} while(1);
```

In the above code, mutex, empty and full are semaphores. Here mutex is initialized to 1, empty is initialized to n (maximum size of the buffer) and full is initialized to 0.

The mutex semaphore ensures mutual exclusion. The empty and full semaphores count the number of empty and full spaces in the buffer.

After the item is produced, wait operation is carried out on empty. This indicates that the empty space in the buffer has decreased by 1. Then wait operation is carried out on mutex so that consumer process cannot interfere.

After the item is put in the buffer, signal operation is carried out on mutex and full. The former indicates that consumer process can now act and the latter shows that the buffer is full by 1.

Consumer Process

The code that defines the consumer process is given below:

```
do {


  wait(full);

  wait(mutex);

  . .

  . REMOVE ITEM FROM BUFFER
```

```
  signal(mutex);

  signal(empty);
  .
  . CONSUME ITEM
  .
} while(1);
```

The wait operation is carried out on full. This indicates that items in the buffer have decreased by 1. Then wait operation is carried out on mutex so that producer process cannot interfere.

Then the item is removed from buffer. After that, signal operation is carried out on mutex and empty. The former indicates that consumer process can now act and the latter shows that the empty space in the buffer has increased by 1.

2. Reader Writer Problem

The Problem Statement

There is a shared resource which should be accessed by multiple processes. There are two types of processes in this context. They are reader and writer. Any number of readers can read from the shared resource simultaneously, but only one writer can write to the shared resource. When a writer is writing data to the resource, no other process can access the resource. A writer cannot write to the resource if there are non zero number of readers accessing the resource at that time.

The Reader Writer Problem Solution Using Semaphores

It is evident that readers have higher priority than writer. If a writer wants to write to the resource, it must wait until there are no readers currently accessing that resource.

Here, we use one mutex m and a semaphore w. An integer variable read_count is used to maintain the number of readers currently accessing the resource. The variable read_count is initialized to 0. A value of 1 is given initially to m and w.

Instead of having the process to acquire lock on the shared resource, we use the mutex m to make the process to acquire and release lock whenever it is updating the read_count variable.

The code for the writer process:

```
while(TRUE)

{

wait(w);

  /* perform the write operation *

signal(w);

}
```

And, the code for the reader process:

```
while(TRUE)
{
 //acquire lock
    wait(m);
    read_count++;
    if(read_count == 1)
        wait(w);
    //release lock
    signal(m);
   /* perform the reading operation */
    // acquire lock
    wait(m);
    read_count--;
    if(read_count == 0)
        signal(w);

    // release lock
    signal(m);
}
```

- As seen above in the code for the writer, the writer just waits on the w semaphore until it gets a chance to write to the resource.
- After performing the write operation, it increments w so that the next writer can access the resource.
- On the other hand, in the code for the reader, the lock is acquired whenever the read_count is updated by a process.
- When a reader wants to access the resource, first it increments the read_count value, then accesses the resource and then decrements the read_count value.
- The semaphore w is used by the first reader which enters the critical section and the last reader which exits the critical section.
- The reason for this is, when the first readers enters the critical section, the writer is blocked from the resource. Only new readers can access the resource now.

- Similarly, when the last reader exits the critical section, it signals the writer using the w semaphore because there are zero readers now and a writer can have the chance to access the resource.
-

Case	Process 1	Process 2	Allowed / Not Allowed
Case 1	Writing	Writing	Not Allowed
Case 2	Reading	Writing	Not Allowed
Case 3	Writing	Reading	Not Allowed
Case 4	Reading	Reading	Allowed

CASE 1: WRITING - WRITING → NOT ALLOWED. That is when two or more than two processes are willing to write, then it is not allowed.

Explanation:

1. The initial value of semaphore write = 1
2. Suppose two processes P0 and P1 wants to write, let P0 enter first the writer code, The moment P0 enters
3. Wait(write); will decrease semaphore write by one, now write = 0
4. And **continue** WRITE INTO THE **FILE**
5. Now suppose P1 wants to write at the same time (will it be allowed?) let's see.
6. P1 does Wait(write), since the write value is already 0, therefore from the definition of wait, it will go into an infinite loop (i.e. Trap), hence P1 can never write anything, till P0 is writing.
7. Now suppose P0 has finished the task, it will
8. signal(write); will increase semaphore write by 1, now write = 1
9. **if** now P1 wants to write it since semaphore write > 0
10. This proofs that, **if** one process is writing, no other process is allowed to write.

CASE 2: READING - WRITING → NOT ALLOWED. That is when one or more than one process is reading the file, then writing by another process is not allowed.

Explanation:

1. Initial value of semaphore mutex = 1 and variable readcount = 0
2. Suppose two processes P0 and P1 are in a system, P0 wants to read **while** P1 wants to write, P0 enter first into the reader code, the moment P0 enters
3. Wait(mutex); will decrease semaphore mutex by 1, now mutex = 0
4. Increment readcount by 1, now readcount = 1, next
5. **if** (readcount == 1)// evaluates to TRUE
6. {
7. wait (write); // decrement write by 1, i.e. write = 0(which
8. clearly proves that **if** one or more than one
9. reader is reading then no writer will be
10. allowed.
11. }
12. signal(mutex); // will increase semaphore mutex by 1, now mutex = 1 i.e. other readers are allowed to enter.
13. And reader continues to --READ THE **FILE**?
14. Suppose now any writer wants to enter into its code then:

15. As the first reader has executed wait (write); because of which write value is 0, therefore wait(writer); of the writer, code will go into a n infinite loop and no writer will be allowed.
16. This proofs that, **if** one process is reading, no other process is allowed to write.
17. Now suppose P0 wants to stop the reading and wanted to exit then
18. Following sequence of instructions will take place:
19. wait(mutex); // decrease mutex by 1, i.e. mutex = 0
20. readcount --; // readcount = 0, i.e. no one is currently reading
21. **if** (readcount == 0) // evaluates TRUE
22. {
23. signal (write); // increase write by one, i.e. write = 1
24. }
25. signal(mutex);// increase mutex by one, i.e. mutex = 1
26. Now **if** again any writer wants to write, it can **do** it now, since write > 0

CASE 3: WRITING -- READING → NOT ALLOWED. That is when if one process is writing into the file, then reading by another process is not allowed.

Explanation:

1. The initial value of semaphore write = 1
2. Suppose two processes P0 and P1 are in a system, P0 wants to write **while** P1 wants to read, P0 enter first into the writer code, The moment P0 enters
3. Wait(write); will decrease semaphore write by 1, now write = 0
4. And **continue** WRITE INTO THE **FILE**
5. Now suppose P1 wants to read the same time (will it be allowed?) let's see.
6. P1 enters reader's code
7. Initial value of semaphore mutex = 1 and variable readcount = 0
8. Wait(mutex); will decrease semaphore mutex by 1, now mutex = 0
9. Increment readcount by 1, now readcount = 1, next
10. **if** (readcount == 1)// evaluates to TRUE
11. {
12. wait (write); // since value of write is already 0, hence it
13. will enter into an infinite loop and will not be
14. allowed to proceed further (which clearly
15. proves that **if** one writer is writing then no
16. reader will be allowed.
17. }
18. The moment writer stops writing and willing to exit then
19. This proofs that, **if** one process is writing, no other process is allowed to read.
20. The moment writer stops writing and willing to exit then it will execute:
21. signal(write); will increase semaphore write by 1, now write = 1
22. **if** now P1 wants to read it can since semaphore write > 0

CASE 4: READING - READING → ALLOWED. That is when one process is reading the file, and other process or processes is willing to read, then they all are allowed i.e. reading - reading is not mutually exclusive. Explanation :

```
Initial value of semaphore mutex = 1 and variable readcount = 0
Suppose three processes P0, P1 and P2 are in a system, all the three processes P0, P1, and P2 want to read, let P0 enter first into the reader code, the moment P0 enters
Wait( mutex ); will decrease semaphore mutex by 1, now mutex = 0
Increment readcount by 1, now readcount = 1, next
if (readcount == 1)// evaluates to TRUE
{
  wait (write); // decrement write by 1, i.e. write = 0(which
                  clearly proves that if one or more than one
                  reader is reading then no writer will be
                  allowed.
}
signal(mutex); // will increase semaphore mutex by 1, now mutex = 1 i.e. other readers are allowed to enter.
And P0 continues to --READ THE FILE?
→Now P1 wants to enter the reader code
current value of semaphore mutex = 1 and variable readcount = 1
let P1 enter into the reader code, the moment P1 enters
Wait( mutex ); will decrease semaphore mutex by 1, now mutex = 0
Increment readcount by 1, now readcount = 2, next
if (readcount == 1)// eval. to False, it will not enter if block
signal(mutex); // will increase semaphore mutex by 1, now mutex = 1 i.e. other readers are allowed to enter.
Now P0 and P1 continues to --READ THE FILE?
→Now P2 wants to enter the reader code
current value of semaphore mutex = 1 and variable readcount = 2
let P2 enter into the reader code, The moment P2 enters
Wait( mutex ); will decrease semaphore mutex by 1, now mutex = 0
Increment readcount by 1, now readcount = 3, next
if (readcount == 1)// eval. to False, it will not enter if block
signal(mutex); // will increase semaphore mutex by 1, now mutex = 1 i.e. other readers are allowed to enter.
Now P0, P1, and P2 continues to --READ THE FILE?
Suppose now any writer wants to enter into its code then:
As the first reader P0 has executed wait (write); because of which write value is 0, therefore wait(writer); of the writer, code will go into an infinite loop and no writer will be allowed.
Now suppose P0 wants to come out of system( stop reading) then
wait(mutex); //will decrease semaphore mutex by 1, now mutex = 0
readcount --; // on every exit of reader decrement readcount by
                one i.e. readcount = 2
if (readcount == 0)// eval. to FALSE it will not enter if block
signal(mutex); // will increase semaphore mutex by 1, now mutex = 1 i.e. other readers are allowed to exit
→ Now suppose P1 wants to come out of system (stop reading) then
wait(mutex); //will decrease semaphore mutex by 1, now mutex = 0
readcount --; // on every exit of reader decrement readcount by
                one i.e. readcount = 1
if (readcount == 0)// eval. to FALSE it will not enter if block
signal(mutex); // will increase semaphore mutex by 1, now mutex = 1 i.e. other readers are allowed to exit
→Now suppose P2 (last process) wants to come out of system (stop reading) then
wait(mutex); //will decrease semaphore mutex by 1, now mutex = 0
readcount --; // on every exit of reader decrement readcount by
                one i.e. readcount = 0
if (readcount == 0)// eval. to TRUE it will enter into if block
{
 signal (write); // will increment semaphore write by one, i.e.
          now write = 1, since P2 was the last process
          which was reading, since now it is going out,
          so by making write = 1 it is allowing the writer
          to write now.
}
signal(mutex); // will increase semaphore mutex by 1, now mutex = 1
```

The above explanation proves that **if** one or more than one processes are willing to read simultaneously then they are allowed.

Monitors in Operating System

- 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.*
- 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.

Components of Monitor

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.

Advantages of Monitor

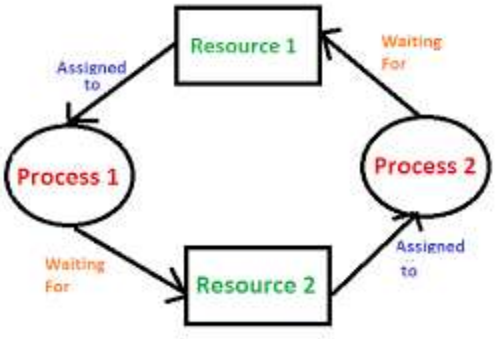
It makes the parallel programming easy, and if monitors are used, then there is less error-prone as compared to the semaphore.

Difference between Monitors and Semaphore

Monitors	Semaphore
We can use condition variables only in the monitors.	In semaphore, we can use condition variables anywhere in the program, but we cannot use conditions variables in a semaphore.
In monitors, wait always block the caller.	In semaphore, wait does not always block the caller.
The monitors are comprised of the shared variables and the procedures which operate the shared variable.	The semaphore S value means the number of shared resources that are present in the system.
Condition variables are present in the monitor.	Condition variables are not present in the semaphore.

Deadlock System Model

Deadlock is a situation that occurs in OS when any process enters a waiting state because another waiting process is holding the demanded resource.



Process 1 is holding Resource 1 while Process 2 acquires Resource 2, and Process 2 is waiting for Resource 1.

Operations :
In normal operation, a process must request a resource before using it and release it when finished, as shown below.

1. **Request –**
   If the request cannot be granted immediately, the process must wait until the resource(s) required to become available. The system, for example, uses the functions open(), malloc(), new(), and request ().

2. **Use –**
   The process makes use of the resource, such as printing to a printer or reading from a file.

3. **Release –**
   The process relinquishes the resource, allowing it to be used by other processes.

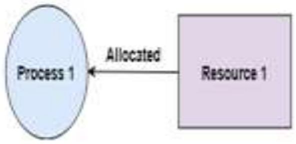
Deadlock prevention

A deadlock happens in operating system when two or more processes need some resource to complete their execution that is held by the other process.

A deadlock occurs if the four Coffman conditions hold true. But these conditions are not mutually exclusive. They are given as follows –

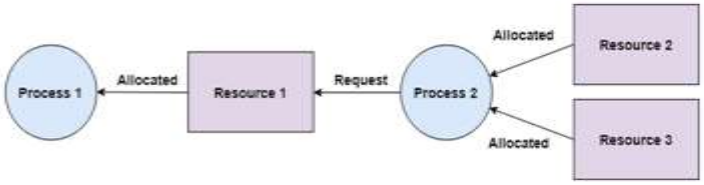
Mutual Exclusion

There should be a resource that can only be held by one process at a time. In the diagram below, there is a single instance of Resource 1 and it is held by Process 1 only.



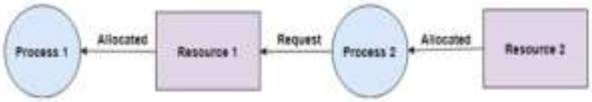
Hold and Wait

A process can hold multiple resources and still request more resources from other processes which are holding them. In the diagram given below, Process 2 holds Resource 2 and Resource 3 and is requesting the Resource 1 which is held by Process 1.



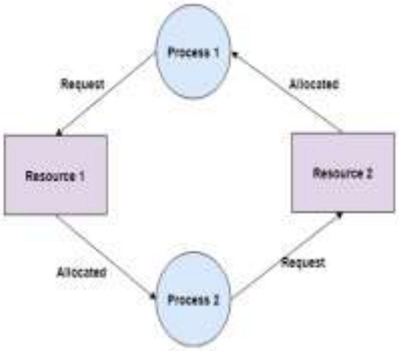
No Preemption

A resource cannot be preempted from a process by force. A process can only release a resource voluntarily. In the diagram below, Process 2 cannot preempt Resource 1 fromProcess 1. It will only be released when Process 1 relinquishes it voluntarily after its execution is complete.



Circular Wait

A process is waiting for the resource held by the second process, which is waiting for the resource held by the third process and so on, till the last process is waiting for a resource held by the first process. This forms a circular chain. For example: Process 1 is allocated Resource2 and it is requesting Resource 1. Similarly, Process 2 is allocated Resource 1 and it is requesting Resource 2. This forms a circular wait loop.



Deadlock Prevention:

Deadlock prevention is a set of methods for ensuring that at least one of these necessary conditions cannot hold.

Mutual Exclusion:

The mutual exclusion condition holds for non sharable. The example is a printer cannot be simultaneously shared by several processes. Sharable resources do not require mutual exclusive access and thus cannot be involved in a dead lock. The example is read only files which are in sharing condition. If several processes attempt to open the read only file at the same time they can be guaranteed simultaneous access.

Hold and wait:

To ensure that the hold and wait condition never occurs in the system, we must guaranty that whenever a process requests a resource it does not hold any other resources. There are two protocols to handle these problems such as one protocol that can be used requires each process to request and be allocated all its resources before it begins execution. The other protocol allows a process to request resources only when the process has no resource. These protocols have two main disadvantages. First, resource utilization may be low, since many of the resources may be allocated but unused for a long period. 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.

No Preemption:

To ensure that this condition does not hold, a protocol is used. If a process is holding some resources and request another resource that cannot be immediately allocated to it. The preempted one added to a list of resources for which the process is waiting. The process will restart only when it can regain its old resources, as well as the new ones that it is requesting. Alternatively if a process requests some resources, we first check whether they are available. If they are, we allocate them. If they are not available, we check whether they are allocated to some other process that is waiting for additional resources. If so, we preempt the desired resources from the waiting process and allocate them to the requesting process. If the resources are not either available or held by a waiting process, the requesting process must wait.

Circular Wait:

We can ensure that this condition never holds by ordering of all resource type and to require that each process requests resource in an increasing order of enumeration.

Deadlock Avoidance

Deadlock Avoidance Requires additional information about how resources are to be used. Simplest and most useful model requires that each process declare the maximum number of resources of each type that it may need. The deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that there can never be a circular-wait condition. Resource-allocation state is defined by the number of available and allocated resources, and the maximum demands of the processes

Safe and Unsafe States

The resource allocation state of a system can be defined by the instances of available and allocated resources, and the maximum instance of the resources demanded by the processes.A state of a system recorded at some random time is shown below.

<u>Resources Assigned (table1)</u>

Process	Type 1	Type 2	Type 3	Type 4
A	3	0	2	2
B	0	0	1	1
C	1	1	1	0
D	2	1	4	0

<u>Resources still needed (table2)</u>

Process	Type 1	Type 2	Type 3	Type 4
A	1	1	0	0
B	0	1	1	2
C	1	2	1	0
D	2	1	1	2

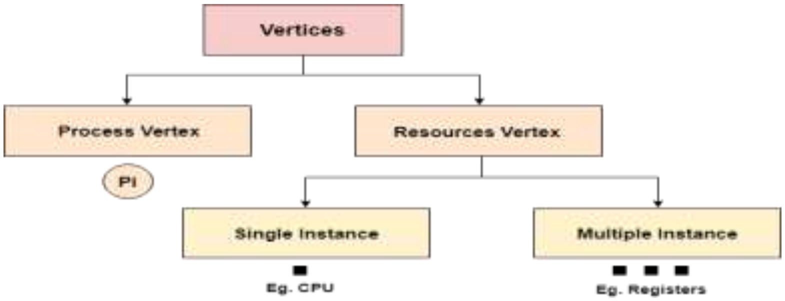
1. E = (7 6 8 4)
2. P = (6 2 8 3)
3. A = (1 4 0 1)

Above tables and vector E, P and A describes the resource allocation state of a system. There are 4 processes and 4 types of the resources in a system. Table 1 shows the instances of each resource assigned to each process. Table 2 shows the instances of the resources, each process still needs. Vector E is the representation of total instances of each resource in the system. Vector P represents the instances of resources that have been assigned to processes. Vector A represents the number of resources that are not in use.

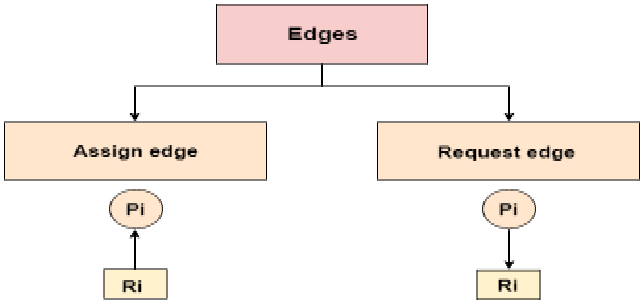
A state of the system is called safe if the system can allocate all the resources requested by all the processes without entering into deadlock.If the system cannot fulfill the request of all processes then the state of the system is called unsafe.The key of Deadlock avoidance approach is when the request is made for resources then the request must only be approved in the case if the resulting state is also a safe state

Resource Allocation Graph

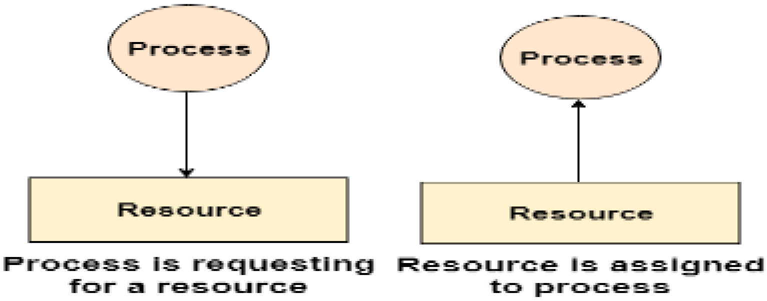
- The resource allocation graph is the pictorial representation of the state of a system. As its name suggests, the resource allocation graph is the complete information about all the processes which are holding some resources or waiting for some resources.
- It also contains the information about all the instances of all the resources whether they are available or being used by the processes .In Resource allocation graph, the process is represented by a Circle while the Resource is represented by a rectangle.



- Vertices are mainly of two types, Resource and process. Each of them will be represented by a different shape. Circle represents process while rectangle represents resource. A resource can have more than one instance. Each instance will be represented by a dot inside the rectangle.

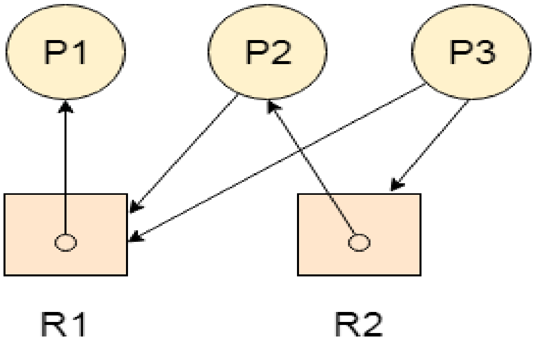


- Edges in RAG are also of two types, one represents assignment and other represents the wait of a process for a resource. The above image shows each of them.
- A resource is shown as assigned to a process if the tail of the arrow is attached to an instance to the resource and the head is attached to a process.
- A process is shown as waiting for a resource if the tail of an arrow is attached to the process while the head is pointing towards the resource.


Process
Resource
Process is requesting for a resource
Process
Resource
Resource is assigned to process

Example

Let's consider 3 processes P1, P2 and P3, and two types of resources R1 and R2. The resources are having 1 instance each. According to the graph, R1 is being used by P1, P2 is holding R2 and waiting for R1, P3 is waiting for R1 as well as R2.The graph is deadlock free since no cycle is being formed in the graph.



Banker's Algorithm

The banker's algorithm is a resource allocation and deadlock avoidance algorithm that tests for safety by simulating the allocation for predetermined maximum possible amounts of all resources, then makes an "s-state" check to test for possible activities, before deciding whether allocation should be allowed to continue. anker's algorithm is named so because the bank would never allocate its money in such a way that it can no longer satisfy the needs of all its customers. The bank would try to be in safe state always.
Following Data structures are used to implement the Banker's Algorithm:
Let 'n' be the number of processes in the system and 'm' be the number of resources types.

Available :

- It is a 1-d array of size 'm' indicating the number of available resources of each type.
- Available[j] = k means there are 'k' instances of resource type Rj

Max :

- It is a 2-d array of size 'n*m' that defines the maximum demand of each process in a system.
- Max[i, j] = k means process Pi may request at most 'k' instances of resource type Rj.

Allocation :

- It is a 2-d array of size ‘n*m’ that defines the number of resources of each type currently allocated to each process.
- Allocation[i, j] = k means process Pi is currently allocated ‘k’ instances of resource type Rj

Need :

- It is a 2-d array of size ‘n*m’ that indicates the remaining resource need of each process.
- Need [i, j] = k means process Pi currently need ‘k’ instances of resource type Rj
- Need [i, j] = Max [i, j] – Allocation [i, j]

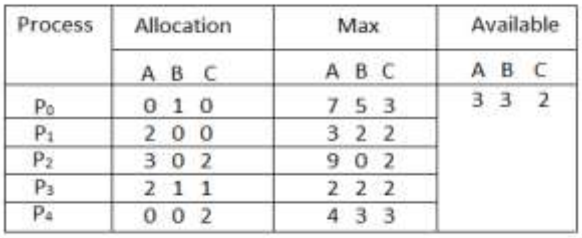
Safety Algorithm

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

1. Let Workand Finish be vectors of length m and n, respectively. Initialize: Work = Available Finish [i] = false for i = 0, 1, …,n- 1.
2. Find and i such that both:
   (a) Finish [i] = false (b) Needi≤Work If no such i exists,
   go to step 4.
3. Work = Work + Allocationi Finish[i] = true
   go to step 2.
4. If Finish [i] == true for all i, then the system is in a safe state.

Example

Considering a system with five processes P0 through P4 and three resources of type A, B, C. Resource type A has 10 instances, B has 5 instances and type C has 7 instances. Suppose at time t0 following snapshot of the systemhasbeentaken:

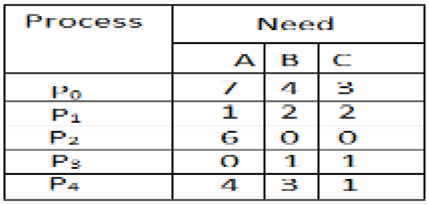


Process	Allocation	Max	Available
	A B C	A B C	A B C
P_0	0 1 0	7 5 3	3 3 2
P_1	2 0 0	3 2 2	
P_2	3 0 2	9 0 2	
P_3	2 1 1	2 2 2	
P_4	0 0 2	4 3 3	

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

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

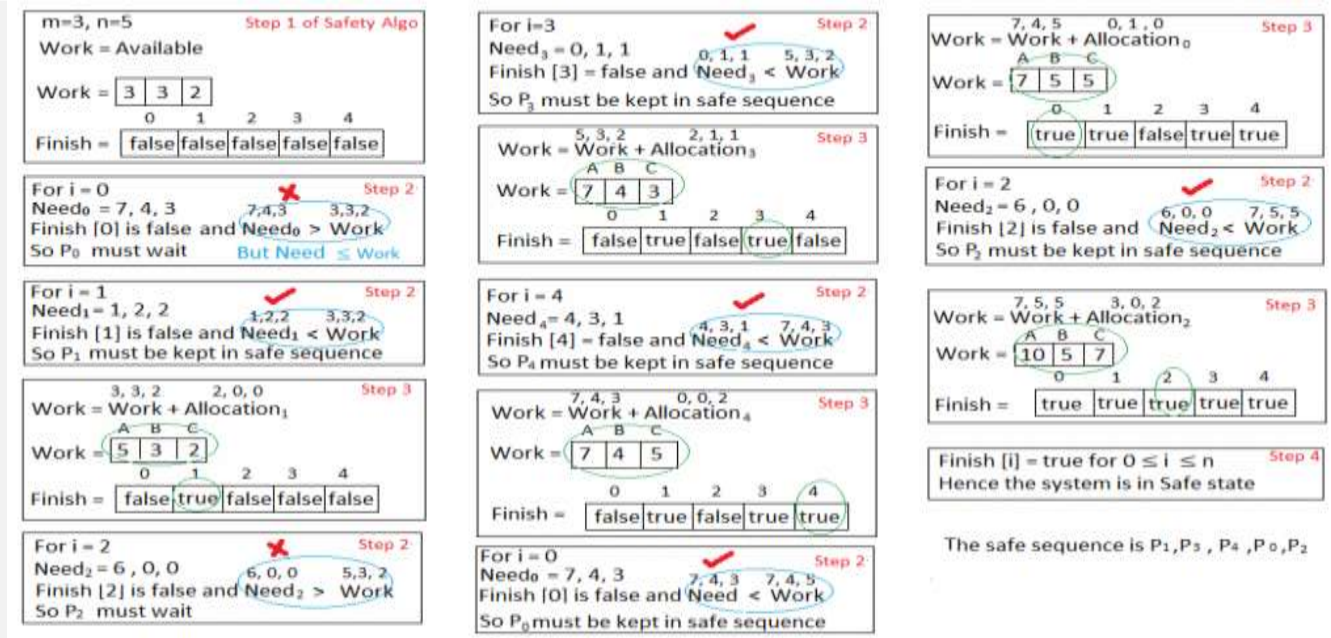
So, the content of Need Matrix is:



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



m=3, n=5 Step 1 of Safety Algo

Work = Available

Work = 3 | 3 | 2

	0	1	2	3	4
Finish =	false	false	false	false	false

For i = 0 Step 2

$Need_0$ = 7, 4, 3

Finish [0] is false and $\overset{7,4,3}{Need_0} > \overset{3,3,2}{Work}$ ✗

So P_0 must wait But Need ≤ Work

For i = 1 Step 2

$Need_1$ = 1, 2, 2

Finish [1] is false and $\overset{1,2,2}{Need_1} < \overset{3,3,2}{Work}$ ✓

So P_1 must be kept in safe sequence

Step 3

Work = $\overset{3,\ 3,\ 2}{Work} + \overset{2,\ 0,\ 0}{Allocation_1}$

	A	B	C
Work =	5	3	2

	0	1	2	3	4
Finish =	false	true	false	false	false

For i = 2 Step 2

$Need_2$ = 6 , 0, 0

Finish [2] is false and $\overset{6,\ 0,\ 0}{Need_2} > \overset{5,3,\ 2}{Work}$ ✗

So P_2 must wait

For i=3 Step 2

$Need_3$ = 0, 1, 1

Finish [3] = false and $\overset{0,\ 1,\ 1}{Need_3} < \overset{5,\ 3,\ 2}{Work}$ ✓

So P_3 must be kept in safe sequence

Step 3

Work = $\overset{5,\ 3,\ 2}{Work} + \overset{2,\ 1,\ 1}{Allocation_3}$

	A	B	C
Work =	7	4	3

	0	1	2	3	4
Finish =	false	true	false	true	false

For i = 4 Step 2

$Need_4$ = 4, 3, 1

Finish [4] = false and $\overset{4,\ 3,\ 1}{Need_4} < \overset{7,\ 4,\ 3}{Work}$ ✓

So P_4 must be kept in safe sequence

Step 3

Work = $\overset{7,\ 4,\ 3}{Work} + \overset{0,\ 0,\ 2}{Allocation_4}$

	A	B	C
Work =	7	4	5

	0	1	2	3	4
Finish =	false	true	false	true	true

For i = 0 Step 2

$Need_0$ = 7, 4, 3

Finish [0] is false and $\overset{7,\ 4,\ 3}{Need} < \overset{7,\ 4,\ 5}{Work}$ ✓

So P_0 must be kept in safe sequence

Step 3

Work = $\overset{7,\ 4,\ 5}{Work} + \overset{0,\ 1\ ,\ 0}{Allocation_0}$

	A	B	C
Work =	7	5	5

	0	1	2	3	4
Finish =	true	true	false	true	true

For i = 2 Step 2

$Need_2$ = 6 , 0, 0

Finish [2] is false and $\overset{6,\ 0,\ 0}{Need_2} < \overset{7,\ 5,\ 5}{Work}$ ✓

So P_2 must be kept in safe sequence

Step 3

Work = $\overset{7,\ 5,\ 5}{Work} + \overset{3,\ 0,\ 2}{Allocation_2}$

	A	B	C
Work =	10	5	7

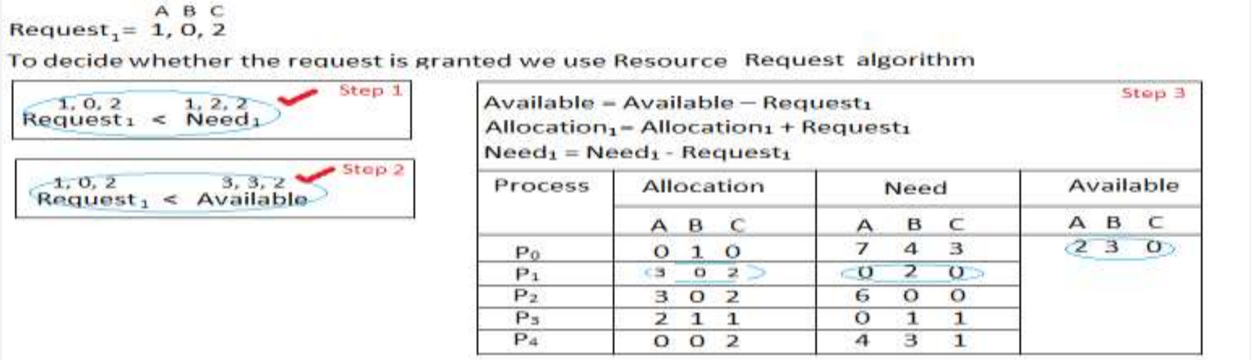
	0	1	2	3	4
Finish =	true	true	true	true	true

Finish [i] = true for 0 ≤ i ≤ n Step 4

Hence the system is in Safe state

The safe sequence is P_1, P_3, P_4, P_0, P_2

Question3. What will happen if process P1 requests one additional instance of resource type A and two instances of resource type C?



$Request_1 = \overset{A\ B\ C}{1,\ 0,\ 2}$

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

Step 1

$\overset{1,\ 0,\ 2}{Request_1} < \overset{1,\ 2,\ 2}{Need_1}$ ✓

Step 2

$\overset{1,\ 0,\ 2}{Request_1} < \overset{3,\ 3,\ 2}{Available}$ ✓

Step 3

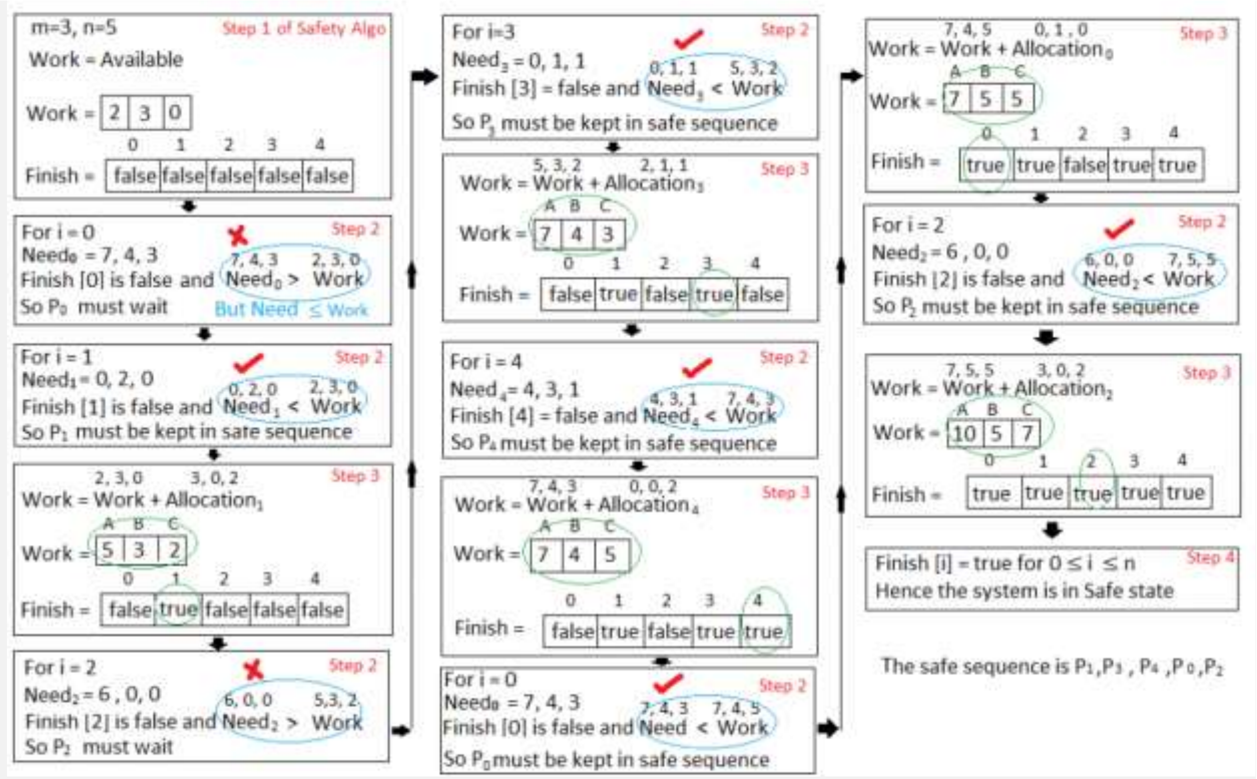
Available = Available − $Request_1$

$Allocation_1$ = $Allocation_1$ + $Request_1$

$Need_1$ = $Need_1$ - $Request_1$

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.



Step 1 of Safety Algo

m=3, n=5

Work = Available

Work = | 2 | 3 | 0 |

0	1	2	3	4
false	false	false	false	false

Finish =

Step 2

For i = 0

Need$_0$ = 7, 4, 3

Finish [0] is false and Need$_0$ > Work (7, 4, 3 > 2, 3, 0)

So P$_0$ must wait

But Need ≤ Work

Step 2

For i = 1

Need$_1$ = 0, 2, 0

Finish [1] is false and Need$_1$ < Work (0, 2, 0 < 2, 3, 0)

So P$_1$ must be kept in safe sequence

Step 3

Work = Work + Allocation$_1$ (2, 3, 0 + 3, 0, 2)

A	B	C
5	3	2

Work =

0	1	2	3	4
false	true	false	false	false

Finish =

Step 2

For i = 2

Need$_2$ = 6 , 0, 0

Finish [2] is false and Need$_2$ > Work (6, 0, 0 > 5,3, 2)

So P$_2$ must wait

Step 2

For i=3

Need$_3$ = 0, 1, 1

Finish [3] = false and Need$_3$ < Work (0, 1, 1 < 5, 3, 2)

So P$_3$ must be kept in safe sequence

Step 3

Work = Work + Allocation$_3$ (5, 3, 2 + 2, 1, 1)

A	B	C
7	4	3

Work =

0	1	2	3	4
false	true	false	true	false

Finish =

Step 2

For i = 4

Need$_4$ = 4, 3, 1

Finish [4] = false and Need$_4$ < Work (4, 3, 1 < 7, 4, 3)

So P$_4$ must be kept in safe sequence

Step 3

Work = Work + Allocation$_4$ (7, 4, 3 + 0, 0, 2)

A	B	C
7	4	5

Work =

0	1	2	3	4
false	true	false	true	true

Finish =

Step 2

For i = 0

Need$_0$ = 7, 4, 3

Finish [0] is false and Need < Work (7, 4, 3 < 7, 4, 5)

So P$_0$ must be kept in safe sequence

Step 3

Work = Work + Allocation$_0$ (7, 4, 5 + 0, 1, 0)

A	B	C
7	5	5

Work =

0	1	2	3	4
true	true	false	true	true

Finish =

Step 2

For i = 2

Need$_2$ = 6 , 0, 0

Finish [2] is false and Need$_2$ < Work (6, 0, 0 < 7, 5, 5)

So P$_2$ must be kept in safe sequence

Step 3

Work = Work + Allocation$_2$ (7, 5, 5 + 3, 0, 2)

A	B	C
10	5	7

Work =

0	1	2	3	4
true	true	true	true	true

Finish =

Step 4

Finish [i] = true for 0 ≤ i ≤ n

Hence the system is in Safe state

The safe sequence is P$_1$, P$_3$, P$_4$, P$_0$, P$_2$

Hence the new system state is safe, so we can immediately grant the request for process **P$_1$** .