NAME: ALISTAIR SALDANHA
SAPID: 60009200024
BATCH: K1

Experiment 11

(Banker's algorithm)

Aim: Implement order scheduling in supply chain using Banker's Algorithm

Theory:

The **Banker's algorithm** is a resource allocation and deadlock avoidance algorithm developed by Edsger Dijkstra that tests for safety by simulating the allocation of predetermined maximum possible amounts of all resources, and then makes an "s-state" check to test for possible deadlock conditions for all other pending activities, before deciding whether allocation should be allowed to continue. For the Banker's algorithm to work, it needs to know three things:

- How much of each resource each process could possibly request[MAX]
- How much of each resource each process is currently holding[ALLOCATED]
- How much of each resource the system currently has available[AVAILABLE]

Resources may be allocated to a process only if it satisfies the following conditions:

1. request ≤ max, else set error condition as process has crossed maximum claim made by it.
2. request ≤ available, else process waits until resources are available.

Some of the resources that are tracked in real systems are memory, semaphores and interface access.

The Banker's Algorithm derives its name from the fact that this algorithm could be used in a banking system to ensure that the bank does not run out of resources, because the bank would never allocate its money in such a way that it can no longer satisfy the needs of all its customers. By using the Banker's algorithm, the bank ensures that when customers request money the bank never leaves a safe state. If the customer's request does not cause the bank to leave a safe state, the cash will be allocated, otherwise the customer must wait until some other customer deposits enough.

Basic data structures to be maintained to implement the Banker's Algorithm:

Let n be the number of processes in the system and m be the number of resource types. Then Following data structures is needed:

- Available: A vector of length m indicates the number of available resources of each type. If Available[j] = k, there are k instances of resource type R_j available.
- Max: An $n \times m$ matrix defines the maximum demand of each process. If Max[i,j] = k, then P_i may request at most k instances of resource type R_j.
- Allocation: An $n \times m$ matrix defines the number of resources of each type currently allocated to each process. If Allocation[i,j] = k, then process P_i is currently allocated k instances of resource type R_j.
- Need: An $n \times m$ matrix indicates the remaining resource need of each process. If Need[i,j] = k, then P_i may need k more instances of resource type R_j to complete the task.

Note: Need[i,j] = Max[i,j] - Allocation[i,j].

Example

Assuming that the system distinguishes between four types of resources, (A, B, C and D), the following is an example of how those resources could be distributed. *Note that this example shows the system at an instant before a new request for resources arrives. Also, the types and number of resources are abstracted. Real systems, for example, would deal with much larger*

quantities of each resource.
Total resources in system:
A B C D
6 5 7 6
Available system resources are:
A B C D
3 1 1 2
Processes (currently allocated resources):

	A	B	C	D
P1	1	2	2	1
P2	1	0	3	3
P3	1	2	1	0

Processes (maximum resources):

	A	B	C	D
P1	3	3	2	2
P2	1	2	3	4
P3	1	3	5	0

Need= maximum resources - currently allocated resources
Processes (need resources):

	A	B	C	D
P1	2	1	0	1
P2	0	2	0	1
P3	0	1	4	0

Safe and Unsafe States
A state (as in the above example) is considered safe if it is possible for all processes to finish executing (terminate). Since the system cannot know when a process will terminate, or how many resources it will have requested by then, the system assumes that all processes will eventually attempt to acquire their stated maximum resources and terminate soon afterward. This is a reasonable assumption in most cases since the system is not particularly concerned with how long each process runs (at least not from a deadlock avoidance perspective). Also, if a process terminates without acquiring its maximum resources, it only makes it easier on the system. A safe state is the decision maker if it is going to process ready queue. Safe State ensures Security.
Given that assumption, the algorithm determines if a state is **safe** by trying to find a hypothetical set of requests by the processes that would allow each to acquire its maximum resources and then terminate (returning its resources to the system). Any state where no such set exists is an **unsafe** state.

Example
State given in the previous example is a safe state by showing that it is possible for each process to acquire its maximum resources and then terminate.

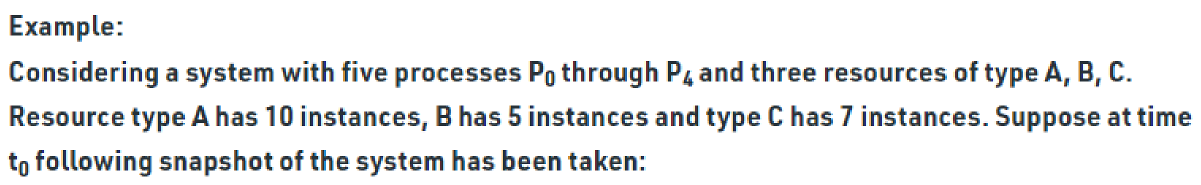
3. P1 acquires 2 A, 1 B and 1 D more resources, achieving its maximum
    - [available resource: <3 1 1 2> - <2 1 0 1> = <1 0 1 1>]
    - The system now still has 1 A, no B, 1 C and 1 D resource available
4. P1 terminates, returning 3 A, 3 B, 2 C and 2 D resources to the system
    - [available resource: <1 0 1 1> + <3 3 2 2> = <4 3 3 3>]
    - The system now has 4 A, 3 B, 3 C and 3 D resources available
5. P2 acquires 2 B and 1 D extra resources, then terminates, returning all its resources

    - [available resource: <4 3 3 3> - <0 2 0 1>+<1 2 3 4> = <5 3 6 6>]
    - The system now has 5 A, 3 B, 6 C and 6 D resources
6. P3 acquires 1 B and 4 C resources and terminates
    - [available resource: <5 3 6 6> - <0 1 4 0> + <1 3 5 0> = <6 5 7 6>
    - The system now has all resources: 6 A, 5 B, 7 C and 6 D
7. Because all processes were able to terminate, this state is safe

Note that these requests and acquisitions are *hypothetical*. The algorithm generates them to check the safety of the state, but no resources are actually given and no processes actually terminate. Also note that the order in which these requests are generated – if several can be fulfilled – doesn't matter, because all hypothetical requests let a process terminate, thereby increasing the system's free resources.

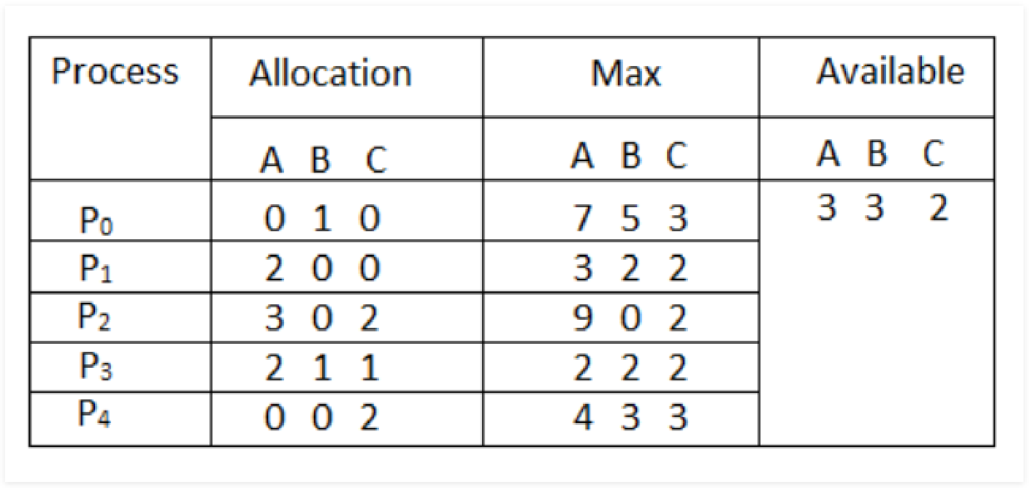
Lab Assignments to complete in this session

1.



Example:

Considering a system with five processes P_0 through P_4 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 t_0 following snapshot of the system has been taken:



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	

2. Implemnt Bankers Algorithm to have following output:

```
Enter the number of resources : 3
enter the max instances of each resources
a= 10
b= 7
c= 7

 Enter the number of processes : 3

 Enter the allocation matrix
      a b c
P[0]  3 2 1
P[1]  1 1 2
```

```
P[2]  4 1 2

Enter the MAX matrix
      a b c
P[0]  4 4 4
P[1]  3 4 5
P[2]  5 2 4

< P[2]  P[0]  P[1] >
```

NAME: ALISTAIR SALDANHA

SAPID: 60009200024

BATCH: K1

EXPERIMENT-9

In [3]:

```
import numpy as np
# To check available >= need
def availability(i):
    for j in range(no_r):
        if(needed[i][j]>available[j]):
            return 0
    return 1

# MAIN FUNCTION
no_p = 5
no_r = 3
# Sequence stores the processes after execution
Sequence = np.zeros((no_p,),dtype=int)
# Visited stores if process of executed successfully
visited = np.zeros((no_p,),dtype=int)
# Given in the Problem
allocated = np.array([[0, 1, 0 ], [2, 0, 0], [3, 0, 2], [2, 1, 1] , [0, 0, 2]])
maximum = np.array([[7, 5, 3 ], [3, 2, 2 ], [ 9, 0, 2 ], [2, 2, 2], [4, 3, 3]])
needed = maximum - allocated
available = np.array([3, 3, 2])
# count - number of processes executed successfully
count = 0
while(count < no_p) :
    temp=0
    for i in range( no_p ):
        if( visited[i] == 0 ):
            if(availability(i)):
                Sequence[count]=i;
                count+=1
                visited[i]=1
                temp=1
                for j in range(no_r):
                    available[j] += allocated[i][j]
    # Process is neither visited nor the resources available for its execution
    if(temp == 0):
        break
if(count < no_p):
    print('The system is Unsafe')
else:
    print("The system is Safe")
    print("Safe Sequence: ",Sequence)
    print("Available Resource:",available)
```

```
The system is Safe
Safe Sequence:  [1 3 4 0 2]
Available Resource: [10  5  7]
```

In [4]:

```
import numpy as np
# To check available >= need
def availability(i):
    for j in range(no_r):
        if(needed[i][j]>available[j]):
            return 0
    return 1

# MAIN FUNCTION
no_p = 3
no_r = 3
# Sequence stores the processes after execution
Sequence = np.zeros((no_p,),dtype=int)
# Visited stores if process of executed successfully
visited = np.zeros((no_p,),dtype=int)
# Given in the Problem
allocated = np.array([[3,2,1], [1,1,2], [4,1,2]])
maximum = np.array([[4,4,4], [3, 4, 5], [5, 2, 4]])
needed = maximum - allocated
available = np.array([2, 3, 2])
# count - number of processes executed successfully
count = 0
while(count < no_p) :
    temp=0
    for i in range( no_p ):
        if( visited[i] == 0 ):
            if(availability(i)):
                Sequence[count]=i;
                count+=1
                visited[i]=1
                temp=1
                for j in range(no_r):
                    available[j] += allocated[i][j]
    # Process is neither visited nor the resources available for its execution
    if(temp == 0):
        break
if(count < no_p):
    print('The system is Unsafe')
else:
    print("The system is Safe")
    print("Safe Sequence: ",Sequence)
    print("Available Resource:",available)
```

```
The system is Safe
Safe Sequence:  [2 0 1]
Available Resource: [10  7  7]
```