OS PROJECT

SAFE SEQUNCE USIING BANKER’S ALGORITHM

Submitted by Yusuf Al Naiem

Reg : 11812720

Section : K18PA

Roll : 18

Git hub link : https://github.com/Yusuf-al/OS-project

Submitted to

Suruchi Talwani

UID:21629

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

Deadlock can arise if following four conditions hold simultaneously :
Mutual Exclusion: One or more than one resource are non-sharable (Only one process can use at a time)
Hold and Wait: A process is holding at least one resource and waiting for resources.
No Preemption: A resource cannot be taken from a process unless the process releases the resource.
Circular Wait: A set of processes are waiting for each other in circular form.

Methods for handling deadlock :

There are three ways to handle deadlock
1) Deadlock prevention or avoidance: The idea is to not let the system into deadlock state.
One can zoom into each category individually, Prevention is done by negating one of above mentioned necessary conditions for deadlock.
Avoidance is kind of futuristic in nature. By using strategy of "Avoidance", we have to make an assumption. We need to ensure that all information about resources which process WILL need are known to us prior to execution of the process. We use Banker's algorithm (Which is in-turn a gift from Dijkstra) in order to avoid deadlock.

2) Deadlock detection and recovery: Let deadlock occur, then do preemption to handle it once occurred.

3) Ignore the problem all together: If deadlock is very rare, then let it happen and reboot the system. This is the approach that both Windows and UNIX take.

Banker's Algorithm : Banker's Algorithm is a resource allocation and deadlock avoidance algorithm. This algorithm test for safety 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. bIn simple terms, it checks if allocation of any resource will lead to deadlock or not, OR is it safe to allocate a resource to a process and if not then resource is not allocated to that process. Determining a safe sequence(even if there is only 1) will assure that system will not go into deadlock. Banker's algorithm is generally used to find if a safe sequence exist or not. But here we will determine the total number of safe sequences and print all safe sequences

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 $\mathbf{R_j}$

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 $\mathbf{P_i}$ may request at most **‘k’** instances of resource type $\mathbf{R_j}$.

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 $\mathbf{P_i}$ is currently allocated **‘k’** instances of resource type $\mathbf{R_j}$

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 $\mathbf{P_i}$ currently need **‘k’** instances of resource type $\mathbf{R_j}$
- for its execution.
- Need [i, j] = Max [i, j] – Allocation [i, j]

Allocation_i specifies the resources currently allocated to process P_i and Need_i specifies the additional resources that process P_i may still request to complete its task.

Banker’s algorithm consists of Safety algorithm and Resource request algorithm

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
$Allocation_i = Allocation_i + Request_i$
Needi = $Need_i$– $Request_i$

The **time complexity of banker's algorithm** is r* (P*P) where P is the number of active processes and r is the number of resources.

There are 5 processes and 3 resource types, resource A with 10 instances, B with 5 instances and C with 7 instances. Consider following and write a c code to find whether the system is in safe state or not ?

Processes	Allocation			Max need			Available			Remaining Need		
P	A	B	C	A	B	C	A	B	C	A	B	C
P0	0	1	0	7	5	3	3	3	2	7	4	3
P1	2	0	0	3	2	2				1	2	2
P2	3	0	2	9	0	2				6	0	0
P3	2	1	1	2	2	2				0	1	1
P4	0	0	2	4	3	3				4	3	1

Here currently available : (3,3,2)

Can we fulfill the request the of

P0(7,4,3) not less or equal to (3,3,2) so P0 will not be executed .

P1(1,2,2) is less than (3,3,2) so P1 will be executed . **(P1)**

Now the available instances (3,3,2)+ P1(2,0,0) = now total currently available (5,3,2)

Can we fulfill the request the of

P2(6,0,0) not less or equal to (5,3,2) so P0 will not be executed .

P3(0,1,1) is less than (5,3,2) P3 will be executed . **(P3)**

Now the available instances (5,3,2) + P1(2,1,1) = now total currently available(7,4,3)

P4(4,3,1) is less than (7,4,3) P3 will be executed . **(P4)**

Now the available instances (7,4,3) + P4(0,0,2) = now total currently available (7,4,5)

Can we fulfill the request the of

P0(7,4,3) is less than (7,4,5) so P0 will be executed **(P0)**

Now the available instances (7,4,5) + P4(0,1,0) = now total currently available (7,5,5)

P2(6,0,0) is less than (7,5,5) so P0 will be executed **(P2)**

Now the available instances (7,5,5) + P2(3,0,2) = now total currently available (10,5,7)

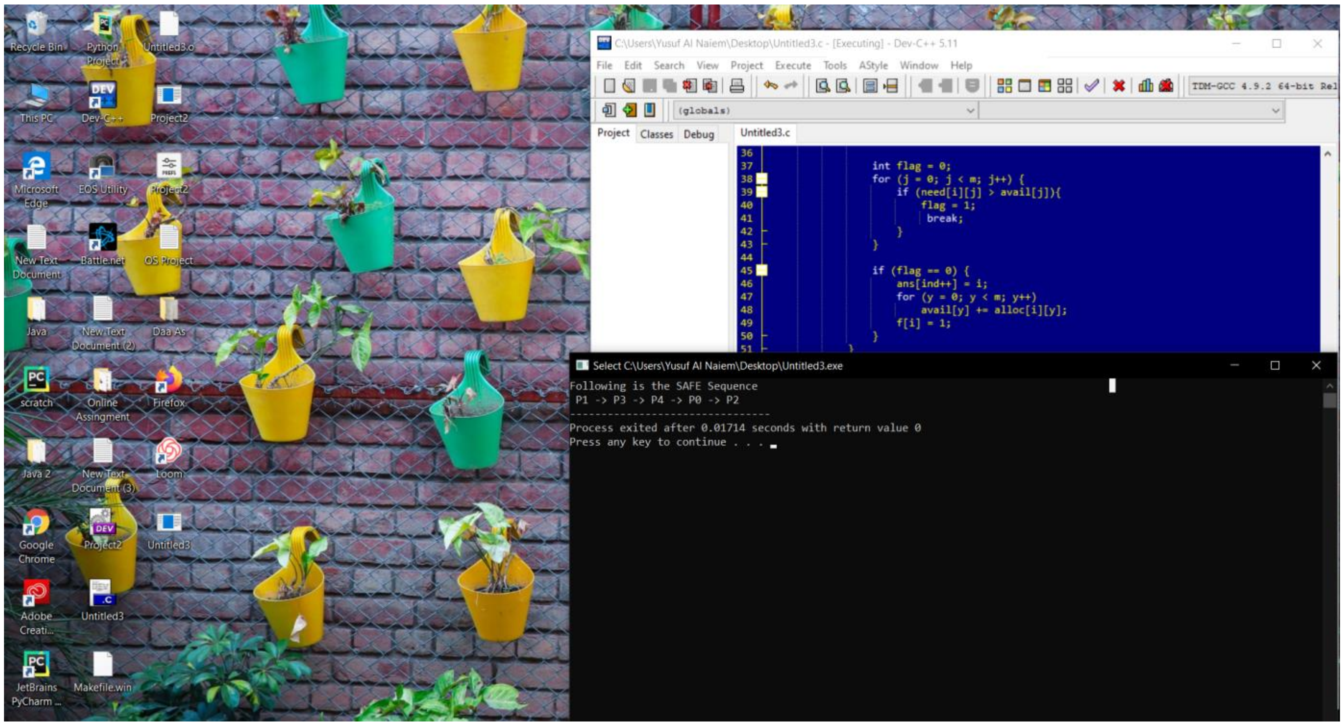
Here now available instances are A=10 ,B=5, C=7 which is same as the given instances .

So the safe sequence is P1 > P3 > P4 > P0> P2

Code of given problem :

```
#include <stdio.h>
int main()
{
    int n, m, i, j, k;
    n = 5; // Number of processes
    m = 3; // Number of resources
    int alloc[5][3] = { { 0, 1, 0 }, // P0    // Allocation Matrix
                        { 2, 0, 0 }, // P1
                        { 3, 0, 2 }, // P2
                        { 2, 1, 1 }, // P3
                        { 0, 0, 2 } }; // P4

    int max[5][3] = { { 7, 5, 3 }, // P0    // MAX Matrix
                      { 3, 2, 2 }, // P1
                      { 9, 0, 2 }, // P2
                      { 2, 2, 2 }, // P3
                      { 4, 3, 3 } }; // P4
    int avail[3] = { 3, 3, 2 }; // Available Resources
    int f[n], ans[n], ind = 0;
    for (k = 0; k < n; k++) {
        f[k] = 0;
    }
    int need[n][m];
    for (i = 0; i < n; i++) {
        for (j = 0; j < m; j++)
            need[i][j] = max[i][j] - alloc[i][j];
    }
    int y = 0;
    for (k = 0; k < 5; k++) {
        for (i = 0; i < n; i++) {
            if (f[i] == 0) {
                int flag = 0;
                for (j = 0; j < m; j++) {
                    if (need[i][j] > avail[j]){
                        flag = 1;
                         break;
                    }
                }
                if (flag == 0) {
                    ans[ind++] = i;
                    for (y = 0; y < m; y++)
                        avail[y] += alloc[i][y];
                    f[i] = 1;
                }
            }
        }
    }
    printf("Following is the SAFE Sequence\n");
    for (i = 0; i < n - 1; i++)
        printf(" P%d ->", ans[i]);
    printf(" P%d", ans[n - 1]);
    return (0);
}
```

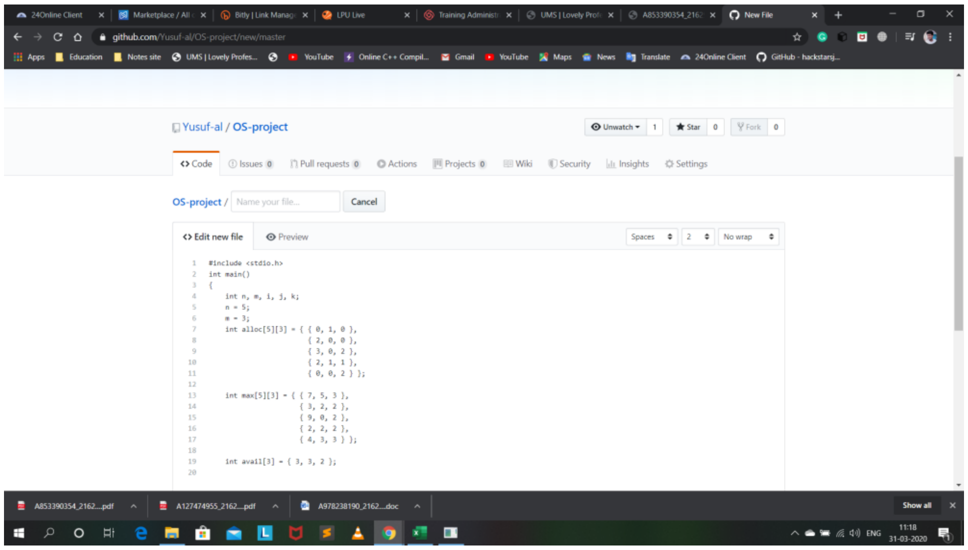


```
            int flag = 0;
            for (j = 0; j < m; j++) {
                if (need[i][j] > avail[j]){
                    flag = 1;
                     break;
                }
            }

            if (flag == 0) {
                ans[ind++] = i;
                for (y = 0; y < m; y++)
                    avail[y] += alloc[i][y];
                f[i] = 1;
            }
```

```
Following is the SAFE Sequence
 P1 -> P3 -> P4 -> P0 -> P2
--------------------------------
Process exited after 0.01714 seconds with return value 0
Press any key to continue . . .
```

Github Submission :



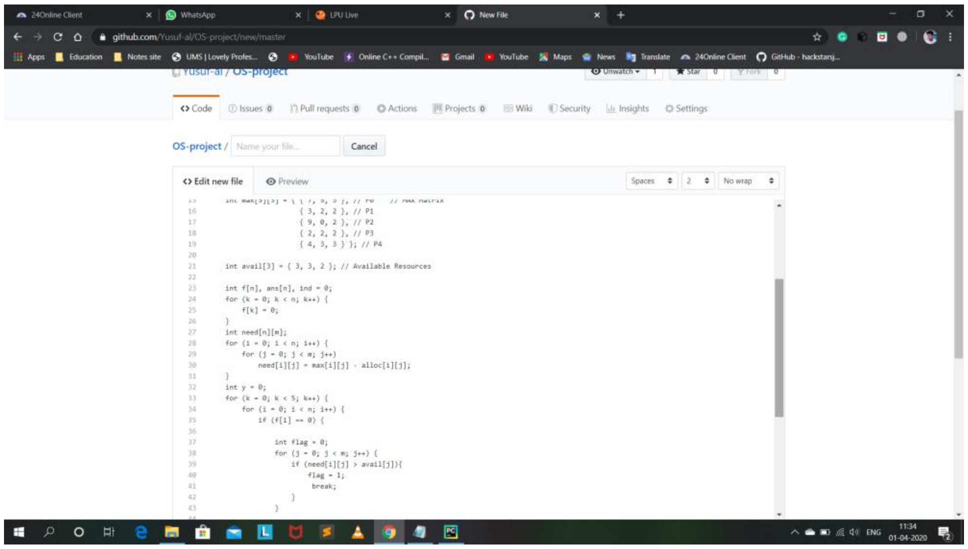
```
#include <stdio.h>
int main()
{
    int n, m, i, j, k;
    n = 5;
    m = 3;
    int alloc[5][3] = { { 0, 1, 0 },
                        { 2, 0, 0 },
                        { 3, 0, 2 },
                        { 2, 1, 1 },
                        { 0, 0, 2 } };

    int max[5][3] = { { 7, 5, 3 },
                      { 3, 2, 2 },
                      { 9, 0, 2 },
                      { 2, 2, 2 },
                      { 4, 3, 3 } };

    int avail[3] = { 3, 3, 2 };

```

Date : 31-03-2020



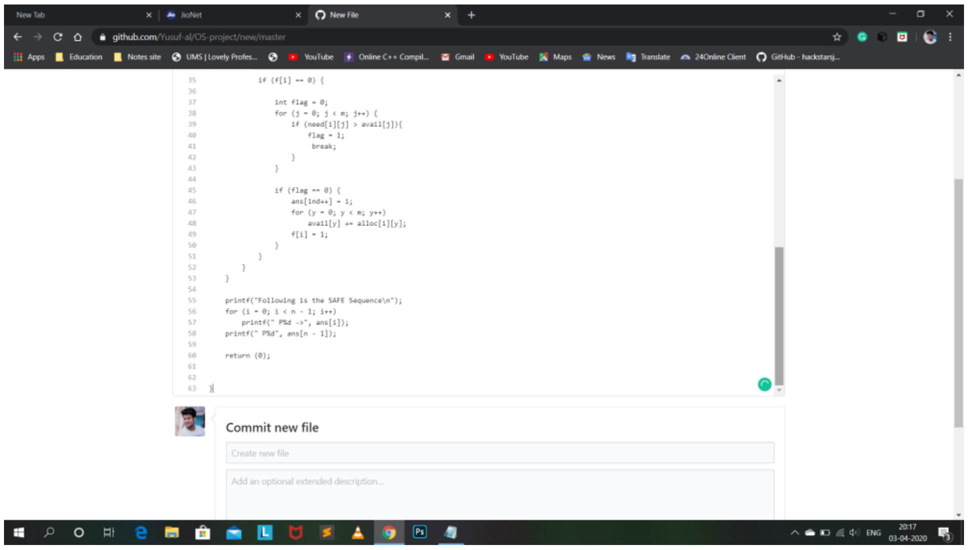
```
int max[5][3] = { { 7, 5, 3 }, // P0    // MAX Matrix
                  { 3, 2, 2 }, // P1
                  { 9, 0, 2 }, // P2
                  { 2, 2, 2 }, // P3
                  { 4, 3, 3 } }; // P4

int avail[3] = { 3, 3, 2 }; // Available Resources

int f[n], ans[n], ind = 0;
for (k = 0; k < n; k++) {
    f[k] = 0;
}
int need[n][m];
for (i = 0; i < n; i++) {
    for (j = 0; j < m; j++)
        need[i][j] = max[i][j] - alloc[i][j];
}
int y = 0;
for (k = 0; k < 5; k++) {
    for (i = 0; i < n; i++) {
        if (f[i] == 0) {

            int flag = 0;
            for (j = 0; j < m; j++) {
                if (need[i][j] > avail[j]){
                    flag = 1;
                     break;
                }
            }
```

Date : 01-04-2020



```
        if (f[i] == 0) {

            int flag = 0;
            for (j = 0; j < m; j++) {
                if (need[i][j] > avail[j]){
                    flag = 1;
                     break;
                }
            }

            if (flag == 0) {
                ans[ind++] = i;
                for (y = 0; y < m; y++)
                    avail[y] += alloc[i][y];
                f[i] = 1;
            }
        }
    }
}

printf("Following is the SAFE Sequence\n");
for (i = 0; i < n - 1; i++)
    printf(" P%d ->", ans[i]);
printf(" P%d", ans[n - 1]);

return (0);


}
```

Date : 03-04-2020

Github Link : https://github.com/Yusuf-al/OS-project