Name: Pratyush Kumar
Reg. No.: 19BCE0506
Slot: L41+L42
Course: CSE2005, Operating Systems

DIGITAL ASSIGNMENT – 2

1. Single Instance Deadlock Avoidance Algorithm

Aim: To write C/C++ code for a single instance deadlock avoidance algorithm and print out the safe sequence or state whether deadlock has occured or not.

Algorithm:

1. Accept the number of processes and the amount of available resources in OS
2. Accept the pid, Max, Alloc from the user
3. Need = Max – Alloc
(Sort the processes according to the need)
4. Available = Total – Sum of Alloc
5. if need <= Available
6. Available = Need - Available
7. Available = Available + Max (after the complettion of process)
8. Print the Safe sequence
9. If need <= available is not satisfied then print “Deadlock”

Code:

```
#include <iostream>
#include <algorithm>
#include <numeric>
#include <vector>
using namespace std;

struct Process {
    string pid;
    int max_req;
    int current_alloc;
    int need;
};
Process process[10];

bool compare(Process a,Process b) {
    return a.need < b.need;
```

```
}

int main() {
    vector<string> safe_seq;
    int i, n, sum_current = 0, available;
    cout << "Enter the number of processes";
    cin >> n;
    cout << "Enter amount of available resources in OS";
    cin >> available;
    for (i = 0; i < n; i++) {
        cout << "Enter the process id, max_requirement and current allocation";
        cin >> process[i].pid >> process[i].max_req >> process[i].current_alloc;
    }

    for (i = 0; i < n; i++) {
        process[i].need = process[i].max_req - process[i].current_alloc;
    }

    sort(process,process+n,compare);

    for ( i = 0; i < n; i++) {
        sum_current += process[i].current_alloc;
    }

    available -= sum_current;

    for (i = 0; i < n; i++) {
        if (process[i].need > 0 && process[i].need <= available) {
            process[i].current_alloc = process[i].current_alloc - process[i].need;
            available -= process[i].need;
            process[i].need = -1;
            available += process[i].max_req;
            safe_seq.push_back(process[i].pid);
        }
    }

    if (safe_seq.size() != n) {
        cout << "Deadlock has occured\n";
        exit(0);
    }
    else {
        cout << "The safe sequence is: " << "\n";
        for (auto i = safe_seq.begin(); i != safe_seq.end(); i++) {
            cout << *i << " ";
        }
```

```
        cout << "\n";
    }

    return 0;
}
```

Input & Output:

1.

```
pratyush@pratyush-Inspiron-5570:~/Desktop/Pratyush$ ./deadlock_avoidance
Enter the number of processes3
Enter amount of available resources in OS12
Enter the process id, max_requirement and current allocationp0 9 5
Enter the process id, max_requirement and current allocationp1 5 3
Enter the process id, max_requirement and current allocationp2 4 2
The safe sequence is:
p1 p2 p0
```

Yes the system is safe

2. At time T1, if system assigns 1 drive to p2, is the system still in safe state?

```
pratyush@pratyush-Inspiron-5570:~/Desktop/Pratyush$ ./deadlock_avoidance
Enter the number of processes3
Enter amount of available resources in OS12
Enter the process id, max_requirement and current allocationp0 10 5
Enter the process id, max_requirement and current allocationp1 4 2
Enter the process id, max_requirement and current allocationp2 9 3
Deadlock has occured
```

2. Banker’s Algotihm with reguest grant algorithm

Aim: To implement Banker’s algorithm to find safe sequence and check whether the system is SAFE. Also incorporate request grant algorithm for a particular process.

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) Needi <= 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 Pi. $Request_i$ [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 P_i by modifying the state as follows:

Available = Available – $Request_i$

$Allocation_i$ = $Allocation_i$ + $Request_i$

$Need_i$ = $Need_i$– $Request_i$

Code:

```c
#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>


int main()

{

    int n, r, i, j;


    printf("Enter the number of processes: ");

    scanf("%d", &n);


    printf("\n");


    printf("Enter the number of resources: ");

    scanf("%d", &r);


    printf("\n");


    int max[n][r], alloc[n][r], avail[r], need[n][r], req[r];


    for (i = 0; i < n; i++) {

        printf("Enter the maximum requirement matrix for process P%d: ", i);

        for (j = 0; j < r; j++) {

            scanf("%d", &max[i][j]);

        }
```

```
}

printf("\n\n");

for (i = 0; i < n; i++) {
    printf("Enter the current allocation matrix for process P%d: ", i);
    for (j = 0; j < r; j++) {
        scanf("%d", &alloc[i][j]);
    }
}

printf("\n\n");

printf("Enter the initial available matrix");
for (i = 0; i < r; i++)
{
    scanf("%d", &avail[i]);
}

printf("\n\n");

printf("Content of NEED matrix is: \n");
for (i = 0; i < n; i++) {
    for (j = 0; j < r; j++) {
        need[i][j] = max[i][j] - alloc[i][j];
        printf("%d ", need[i][j]);
```

```
    }
    printf("\n");
}

printf("\n");

        int f[n], ans[n], ind = 0;
        for (i = 0; i < n; i++) {
                f[i] = 0;
        }

        int count = 0, y;
        do
{
    for (i = 0; i < n; i++) {
      if (f[i] == 0) {

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

                        if (flag == 0) {
```

```
                        ans[ind++] = i;
                        for (y = 0; y < r; y++)
                            avail[y] += alloc[i][y];
                        f[i] = 1;
            count++;
                    }
                }
    }
} while (count != n);

if (count != n) {
    printf("DEADLOCK HAS OCCURRED!!");
    exit(0);
} else {
    printf("SAFE Sequence is: \n");
    for (i = 0; i < n; i++)
        printf("P%d ", ans[i]);
    printf("\n");
}

printf("\n");

printf("Need matrix: \n");
for (i = 0; i < n; i++) {
    for (j = 0; j < r; j++) {
        printf("%d ", need[i][j]);
```

```
    }
    printf("\n");
}

printf("Availability matrix: \n");
for (i = 0; i < r; i++) {
    printf("%d ", avail[i]);
}

printf("\n");

printf("Enter the process number requesting");
scanf("%d", &y);

printf("Enter the instance of each resource type");
for (i = 0; i < r; i++) {
    scanf("%d", &req[i]);
}

int flag = 0;
for (i = 0; i < r; i++) {
    if (need[y][i] < req[i] && avail[i] < req[i]) {
        flag = 1;
        break;
    }
```

```
        if (flag == 0) {
            avail[i] = avail[i] - req[i];
            alloc[y][i] = alloc[y][i] + req[i];
            need[y][i] = need[y][i] - req[i];
        }
    }
    if (flag == 1) {
        printf("Request not granted\n");
    } else {
        printf("Request granted\n");
    }


      return 0;
}
```

Output:

```
pratyush@pratyush-Inspiron-5570:~/Desktop/Pratyush$ ./bankers_algorithm
Enter the number of processes: 5

Enter the number of resources: 4

Enter the maximum requirement matrix for process P0: 0 0 1 2
Enter the maximum requirement matrix for process P1: 1 7 5 0
Enter the maximum requirement matrix for process P2: 2 3 5 6
Enter the maximum requirement matrix for process P3: 0 6 5 2
Enter the maximum requirement matrix for process P4: 0 6 5 6


Enter the current allocation matrix for process P0: 0 0 1 2
Enter the current allocation matrix for process P1: 1 0 0 0
Enter the current allocation matrix for process P2: 1 3 5 4
Enter the current allocation matrix for process P3: 0 6 3 2
Enter the current allocation matrix for process P4: 0 0 1 4
```

```
Enter the initial available matrix1 5 2 0


Content of NEED matrix is:
0 0 0 0
0 7 5 0
1 0 0 2
0 0 2 0
0 6 4 2

SAFE Sequence is:
P0 P2 P3 P4 P1

Need matrix:
0 0 0 0
0 7 5 0
1 0 0 2
0 0 2 0
0 6 4 2
Availability matrix:
3 14 12 12
Enter the process number requesting1
Enter the instance of each resource type0 4 2 0
Request granted
```