**Name : Ansari Aatif Akhlaque**

**Roll No : 25**

**Subject : OSL**

**Assignment No. 5:**

Implement the C program for Deadlock Avoidance Algorithm: Bankers Algorithm.

**Aim:**

To implement Bankers Algorithm for deadlock avoidance using

(a) Safety algorithm

(b) resource request algorithm

**Objectives:**

To understand the concept of deadlock.

How to avoid deadlock by implementing safety algorithm.

**Theory:**

In a multiprogramming environment, several processes may compete for a finite set of resources. A process requests resources; if the resources are not available at that time, the process enters a wait state. Waiting processes may never again change state, because the resources they have requested are held by other waiting processes. This situation is called as a **deadlock**. The resources are partitioned into several types, each of which consists of some number of identical instances. A process must request a resource before using it and release it after using it. The number of resources requested must not exceed the number of resource available in the system.

**A Deadlock situation can arise if the following four conditions hold simultaneously in a system.**

**NECESSORY CONDITIONS FOR DEADLOCK EXISTANCE**

1. Mutual Exclusion: At least one resource must be held in a non-shareable mode, that is only one process at a time can use the resource.

2. Hold and Wait: A process must hold one resource and waiting to acquire additional resources that are currently being held by other processes

3. No Preemption : Resources cannot be pre-empted. The resources can be released by the process only when it completes the task.

1. Circular Wait : A set of (P0,P1,P2,….Pn} of waiting processes must exist such that process P0 is waiting for a resource that is held by P1, P1 is waiting for a resource that is held by P2, and so on and lastly Pn is waiting for a resource held by P0.

**METHODS OF HANDLING DEADLOCKS**:

1. We can use a protocol to prevent or avoid deadlocks, ensuring that the system

will never enter a deadlock state.

2. We can allow the system to enter a deadlock state, detect and recover it.

3. We can ignore the problem altogether and pretend that deadlocks never occur in the system. This solution is used by most operating systems, including UNIX

**DEADLOCK PREVENTION:**

It is a set of methods ensuring that at least one of the necessary conditions do not hold. These methods prevent deadlocks by constraining how requests for resources can be made.

**DEADLOCK AVOIDANCE:**

It requires that the Operating system be given in advance additional information concerning which resource a process will request and use during its lifetime. With this additional knowledge we can decide for each request whether a process should wait or not.

**SAFE AND UNSAFE STATE:**

A state is safe if the system can allocate resources to each process (up to its maximum) in some order and still avoid a deadlock. A system is in a safe state only if there exists a safe sequence, represented as: < P1, P2, P3, .... Pn >. A sequence of processes < P1 , P2 , P3 , ..... Pn > is a safe sequence for the current allocation state if, for each process Pi, the resources that Pi can still request can be satisfied by the currently available resources plus the resources held by all Pj, with j < i. If no such sequence exists, the system is in unsafe state, or in a deadlock.

**BANKERS ALGORITHM:**

This is a deadlock avoidance algorithm. It is mainly used in a banking system to ensure that the bank never allocates its available cash such that it can no longer satisfy the needs of all customers, and hence the name. Whenever a new process enters the system, it must declare the maximum number of instances of each resource type that it may need. This number should not exceed the total number of resources in the system. When user requests resources, the system determines whether the allocation of these resources will leave the system in a safe state. If system remains in safe state, request is granted, otherwise it enters a wait state.

**Data Structures:**

Let n be the number of processes and m the number of resource types . Then the following data structures are maintained:

1. Available[]: A vector of length m indicates the number of available resource of each type.

If Available[j] = k, there are k instances of resource type Rj are available.

2. Max [][] : An n\*m matrix defines maximum demand of each process.

If Max[i,j] = k, then process Pi may request at most k instances of

resource Type Rj.

3. Allocation[][] : An n\*m matrix that defines the number of resources of each type currently allocated to each process.

If Allocation[i,j] = k, then process Pi is currently owns k instances of resource type Rj.

. 4. Need [][] : An n\*m matrix indicating the remaining resource need of each process.

If Need[i,j] = k, then process Pi may need k more instances of resource type Rj.

**(A)** **SAFETY ALGORITHM :**

The algorithm for finding whether the system is in a safe state or not.

Step.1. : Let Work and Finish be vectors of length m and n respectively .

Initialize Work = Available and Finish[i] = false for i=0 ,1, 2, 3 , ...... , n-1 .

Step.2. : Find an 'i' such that both

(a) Finish[i] = false

(b) Need <= work for ith process for every resource type.

If no such 'i' exists go to step.4. .

step.3. : Do for the 'i'th process ;

Work = Work + Allocation, for all resource types

Finish[i] = true.

Goto step.2. .

step.4. : If Finish[i]==true for all 'i' , system is in safe state .

else system is in unsafe state .

Time complexity of this algorithm is m\* n \* n.

**(B)RESOURCE-REQUEST ALLOCATION ALGORITHM :**

This algorithm is used to grant a request. Let Request be the request vector for process Pi. If Request[j] = k, for ith process, then process Pi wants k instances of resource type Rj. When a request for resources is made by the process Pi, following actions are taken:

step.1. : If Request <= Need , for ith process for all resource types, goto step

step.2, else raise an error condition, since process has exceeded its

maximum claim .

step.2. : If Request <= Available, for ith process for all resource types goto

step.3. else process must enter a wait state since resources are

unavailable .

step.3. : Have the system pretend to have allocated the requested resources to

process Pi by modifying the state as follows:

**Available = Available – Request,** for ith process for all resource types

**Allocation = Allocation + Request,** for ith process for all resource types

**Need = Need- Request,** for ith process for all resource types

step.4. : If the resulting resource allocation state is a safe state , the transaction

is completed and the process Pi is allocated its resources. However, if the

new state is unsafe , then process Pi must wait for request and the old

resource allocation state is restored .

**Platform :-** Linux

**Language**:- C/C++-Programming Language

**Conclusion** :-

The Bankers algorithm was studied and successfully implemented in c and all concepts understood.

**Code :**

#include<stdio.h>

int main() {

/\* array will store at most 5 process with 3 resoures if your process or

resources is greater than 5 and 3 then increase the size of array \*/

int p, c, count = 0, i, j, alc[5][3], max[5][3], need[5][3], safe[5], available[3], done[5], terminate = 0;

printf("Enter the number of process and resources");

scanf("%d %d", & p, & c);

// p is process and c is diffrent resources

printf("enter allocation of resource of all process %dx%d matrix", p, c);

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

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

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

}

}

printf("enter the max resource process required %dx%d matrix", p, c);

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

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

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

}

}

printf("enter the available resource");

for (i = 0; i < c; i++)

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

printf("\n need resources matrix are\n");

for (i = 0; i < p; i++)

{

for (j = 0; j < c; j++)

{

need[i][j] = max[i][j] - alc[i][j];

printf("%d\t", need[i][j]);

}

printf("\n");

}

/\* once process execute variable done will stop them for again execution \*/

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

done[i] = 0;

}

while (count < p) {

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

if (done[i] == 0) {

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

if (need[i][j] > available[j])

break;

}

//when need matrix is not greater then available matrix then if j==c will true

if (j == c) {

safe[count] = i;

done[i] = 1;

/\* now process get execute release the resources and add them in available resources \*/

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

available[j] += alc[i][j];

}

count++;

terminate = 0;

} else {

terminate++;

}

}

}

if (terminate == (p - 1)) {

printf("safe sequence does not exist");

break;

}

}

if (terminate != (p - 1)) {

printf("\n available resource after completion\n");

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

printf("%d\t", available[i]);

}

printf("\n safe sequence are\n");

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

printf("p%d\t", safe[i]);

}

}

return 0;

}

**OUTPUT :**

