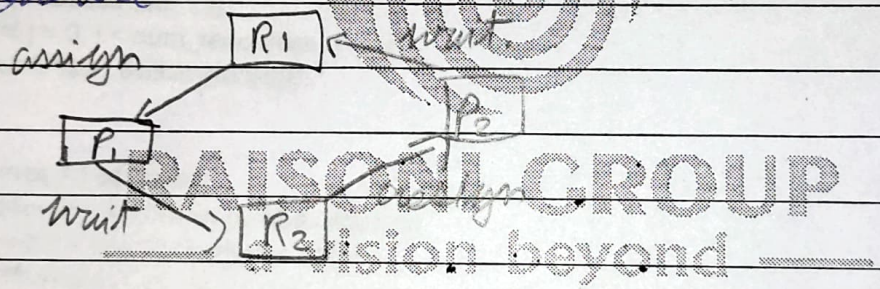
Date
Page No.

Practical No. 6

* Aim:- Implement Deadlock management algorithm (Banker's algo)

* Theory:-

Deadlock is a situation where a set of process are blocked because each process is holding a resource & waiting for other resource.



In diagram, P1 is holding R1 & waiting for R2 which is already held by P2 & P2 is waiting for R1.

• Necessary condition for deadlock.

1) Mutual exclusion:- 2 or more resources are non-shareable (one process can use at a time).

Date
Page No.

2) Hold and wait :- A process is holding at least one resource and waiting for resource

3) No Preemption :- A resource can not be taken from a process unless releases the resource

4) Circular wait :- A set of process waiting for each other in circular form.

Method to handle deadlock :-

1) deadlock Avoidance :- it means the deadlock is occour in the system which we have to avoid & handle the deadlock for handling it we use banker's algo.

When a new process is created in a computer system, It should provide all types of info to OS like upcoming process, request for resources, counting the & delays. Based on this criteria, OS should decide the sequence of process execution. or wait to avoid deadlock.

Date
Page No.

- Things to remember while working with banker's algo.

1) How each process can request for each resource in the system. It is denoted by [max] request.
2) How much the process is currently holding resource. Denoted by [allocated]
3) [available], how much resource are currently available.

* Result:- In this we have successfully implemented deadlock (banker's algo).

P	T	D	K	Total
3M	3M	3M	6M	15M
3	3	3	4	13
Sign With date		8/4/24		

Practical no 6

Program :-

```
#include <stdio.h>
#define MAX_PROCESSES 10
#define MAX_RESOURCES 10
int allocation[MAX_PROCESSES][MAX_RESOURCES];
int max_need[MAX_PROCESSES][MAX_RESOURCES];
int available[MAX_RESOURCES];
int need[MAX_PROCESSES][MAX_RESOURCES];
int finish[MAX_PROCESSES];
int num_processes, num_resources;
// Function prototypes
void calculate_need();
int is_safe();
void request_resources(int process_id, int request[]);
int main() {
   printf("Enter the number of processes: ");
   scanf("%d", &num_processes);
   printf("Enter the number of resources: ");
   scanf("%d", &num_resources);
   // Input allocation matrix
   printf("Enter the allocation matrix:\n");
   for (int i = 0; i < num_processes; ++i) {
      printf("Process %d: ", i);
      for (int j = 0; j < num_resources; ++j) {
         scanf("%d", &allocation[i][j]);
      }
   }
   // Input max_need matrix
   printf("Enter the maximum need matrix:\n");
   for (int i = 0; i < num_processes; ++i) {
      printf("Process %d: ", i);
      for (int j = 0; j < num_resources; ++j) {
         scanf("%d", &max_need[i][j]);
      }
   }
   // Input available resources
   printf("Enter the available resources: ");
   for (int i = 0; i < num_resources; ++i) {
      scanf("%d", &available[i]);
   }
   // Calculate need matrix
   calculate_need();
   // Check if the system is in a safe state
   if (is_safe()) {
      printf("System is in a safe state.\n");
   } else {
      printf("System is in an unsafe state.\n");
```

```
            for (int i = 0; i < num_processes; ++i) {
                printf(" %d", safe_sequence[i]);
            }
            printf("\n");
            return 1; // System is in safe state
        } else {
            return 0; // System is in unsafe state
        }
    }
    // Process requests for resources
    void request_resources(int process_id, int request[]) {
        // Check if request is within need
        for (int i = 0; i < num_resources; ++i) {
            if (request[i] > need[process_id][i]) {
                printf("Error: Request exceeds maximum need.\n");
                return;
            }
            if (request[i] > available[i]) {
                printf("Error: Request exceeds available resources.\n");
                return;
            }
        }
        // Try to allocate resources
        for (int i = 0; i < num_resources; ++i) {
            available[i] -= request[i];
            allocation[process_id][i] += request[i];
            need[process_id][i] -= request[i];
        }
        // Check if system is in a safe state after allocation
        if (is_safe()) {
            printf("Request granted.\n");
        } else {
            printf("Request denied. System would be in an unsafe state.\n");
            // Rollback allocation
            for (int i = 0; i < num_resources; ++i) {
                available[i] += request[i];
                allocation[process_id][i] -= request[i];
                need[process_id][i] += request[i];
            }
        }
    }
```