Experiment No.7
Process Management: Deadlock a. Write a program to demonstrate the concept of deadlock avoidance through Banker's Algorithm
Date of Performance:
Date of Submission:
Marks:
Sign:

Aim: Process Management: Deadlock

Objective:

a.Write a program to demonstrate the concept of deadlock avoidance through Banker's Algorithm

Theory:

Data Structures for the Banker's Algorithm.

Let n = number of processes, and m = number of resources types.

ν Available: Vector of length m. If available [j] = k, there are k instances of resource type Rj available

ν Max: n x m matrix.

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

ν Allocation: n x m matrix. If Allocation[i,j] = k then Pi is currently allocated k instances of Rj

ν Need: n x m matrix. If Need[i,j] = k, then Pi may need k more instances of Rj to complete

its task

Need [i,j] = Max[i,j] – Allocation [i,j]

Safety Algorithm

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

   Work = Available

   Finish [i] = false for i = 0, 1, …, n- 1 2. Find an i such that both:

   (a) Finish [i] = false

(b) Needi ≤ Work

If no such i exists,
go to step 4 3. Work =
Work + Allocationi

Finish[i]
= true
go to
step 2

4. If Finish [i] == true for all i, then the system is in a safe state.

```
#include <stdio.h>


#define P 5  // Number of processes

#define R 3  // Number of resources


int available[R] = {3, 3, 2};

int max[P][R] = {{7, 5, 3},

            {3, 2, 2},

            {9, 0, 2},

            {2, 2, 2},

            {4, 3, 3}};

int allocation[P][R] = {{0, 1, 0},

                {2, 0, 0},

                {3, 0, 2},

                {2, 1, 1},

                {0, 0, 2}};
```

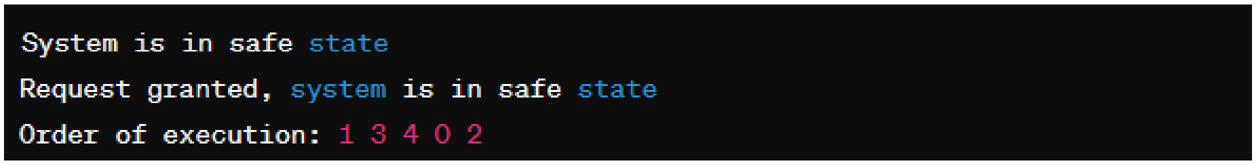
```
int need[P][R];


void calculate_need() {
    for (int i = 0; i < P; i++) {
        for (int j = 0; j < R; j++) {
            need[i][j] = max[i][j] - allocation[i][j];
        }
    }
}


int is_safe(int processes[], int available[], int max[][R], int allocation[][R]) {
    int work[R];
    for (int i = 0; i < R; i++) {
        work[i] = available[i];
    }
    int finish[P] = {0};


    int count = 0;
    while (count < P) {
        int found = 0;
        for (int i = 0; i < P; i++) {
            if (finish[i] == 0) {
                int j;
                for (j = 0; j < R; j++) {
```

```
                if (need[i][j] > work[j]) {
                    break;
                }
            }
            if (j == R) {
                for (int k = 0; k < R; k++) {
                    work[k] += allocation[i][k];
                }
                processes[count++] = i;
                finish[i] = 1;
                found = 1;
            }
        }
    }
    if (found == 0) {
        printf("System is not in safe state\n");
        return 0;
    }
  }
  printf("System is in safe state\n");
  return 1;
}

int request_resources(int process_num, int request[]) {
```

```
for (int i = 0; i < R; i++) {
    if (request[i] > need[process_num][i]) {
        printf("Error: Requested resources exceed maximum need\n");
        return -1;
    }
    if (request[i] > available[i]) {
        printf("Process must wait - resources not available\n");
        return 0;
    }
}

for (int i = 0; i < R; i++) {
    available[i] -= request[i];
    allocation[process_num][i] += request[i];
    need[process_num][i] -= request[i];
}

int processes[P];
int safe = is_safe(processes, available, max, allocation);
if (safe) {
    printf("Request granted, system is in safe state\n");
    printf("Order of execution: ");
    for (int i = 0; i < P; i++) {
        printf("%d ", processes[i]);
        }
        printf("\n");
        return 1;
    } else {
        printf("Request denied, system would enter unsafe state\n");
        // Roll back changes
        for (int i = 0; i < R; i++) {
            available[i] += request[i];
            allocation[process_num][i] -= request[i];
            need[process_num][i] += request[i];
        }
        return 0;
    }
}

int main() {
    calculate_need();

    // Example: Request resources for process 0
    int process_num = 0;
    int request[R] = {0, 0, 2};
    request_resources(process_num, request);

    return 0;
```

}

Output :



```
System is in safe state
Request granted, system is in safe state
Order of execution: 1 3 4 0 2
```

This program demonstrates the Banker's Algorithm for deadlock avoidance. It calculates the need matrix, checks if the system is in a safe state, and grants or denies resource requests.

The Dining Philosophers problem is a classic synchronization problem in computer science where a group of philosophers sit around a dining table with a bowl of spaghetti. Between each pair of adjacent philosophers, there is a fork. The philosophers alternate between thinking and eating. To eat, a philosopher must obtain both forks to their left and right.

However, the problem arises when all philosophers simultaneously pick up the fork to their left, leaving all philosophers unable to eat, causing a deadlock.

Here's a C program that demonstrates the Dining Philosophers problem using pthreads and mutex locks for synchronization:

Code:

```
#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>
```

```
#include <unistd.h>


#define NUM_PHILOSOPHERS 5
#define THINKING 0
#define HUNGRY 1
#define EATING 2
#define LEFT (philosopher_number + NUM_PHILOSOPHERS - 1) %
NUM_PHILOSOPHERS
#define RIGHT (philosopher_number + 1) % NUM_PHILOSOPHERS


int state[NUM_PHILOSOPHERS];
pthread_mutex_t mutex;
pthread_cond_t condition[NUM_PHILOSOPHERS];


void *philosopher(void *arg) {
    int philosopher_number = *(int *) arg;


    while (1) {
        printf("Philosopher %d is thinking.\n", philosopher_number);


        sleep(rand() % 5); // Think for some time


        pthread_mutex_lock(&mutex);
        state[philosopher_number] = HUNGRY;
        printf("Philosopher %d is hungry and wants to eat.\n", philosopher_number);
        test(philosopher_number);
        pthread_mutex_unlock(&mutex);

        pthread_mutex_lock(&mutex);
        while (state[philosopher_number] != EATING) {
            pthread_cond_wait(&condition[philosopher_number], &mutex);
        }
        pthread_mutex_unlock(&mutex);

        printf("Philosopher %d is eating.\n", philosopher_number);
        sleep(rand() % 5); // Eat for some time

        pthread_mutex_lock(&mutex);
        state[philosopher_number] = THINKING;
        printf("Philosopher %d finished eating and is now thinking.\n",
philosopher_number);
        test(LEFT);
        test(RIGHT);
        pthread_mutex_unlock(&mutex);
    }
}

void test(int philosopher_number) {
    if (state[philosopher_number] == HUNGRY &&
        state[LEFT] != EATING &&
        state[RIGHT] != EATING) {
        state[philosopher_number] = EATING;
        pthread_cond_signal(&condition[philosopher_number]);
    }
}

int main() {
    pthread_t philosophers[NUM_PHILOSOPHERS];
    int philosopher_numbers[NUM_PHILOSOPHERS];

    pthread_mutex_init(&mutex, NULL);

    for (int i = 0; i < NUM_PHILOSOPHERS; i++) {
        pthread_cond_init(&condition[i], NULL);
    }

    for (int i = 0; i < NUM_PHILOSOPHERS; i++) {
        philosopher_numbers[i] = i;
        pthread_create(&philosophers[i], NULL, philosopher,
&philosopher_numbers[i]);
    }

    for (int i = 0; i < NUM_PHILOSOPHERS; i++) {
        pthread_join(philosophers[i], NULL);
    }
    pthread_mutex_destroy(&mutex);


    for (int i = 0; i < NUM_PHILOSOPHERS; i++) {
        pthread_cond_destroy(&condition[i]);
    }


    return 0;
}
```

In this program:

We define an array state to keep track of the state of each philosopher.

We use pthreads to create a separate thread for each philosopher.

The philosopher function simulates the behavior of a philosopher, where they alternate between thinking and eating.

The test function checks if a philosopher can start eating by checking if its left and right neighbors are not eating. If so, it sets the state to eating and signals the condition variable.

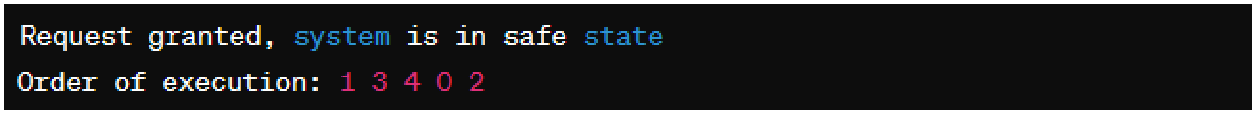
The main function initializes the mutex and condition variables, creates philosopher threads, and waits for them to finish.

Code:

```
gcc -o dining_philosophers dining_philosophers.c -lpthread
```

Then run it:

Output :



```
Request granted, system is in safe state
Order of execution: 1 3 4 0 2
```

Conclution :

In conclusion, implementing the Banker's Algorithm to avoid deadlock in concurrent systems is a crucial strategy for ensuring system stability and reliability. By carefully managing resource allocation and dynamically assessing the safety of potential requests, the Banker's Algorithm helps prevent the occurrence of deadlock scenarios. Through the demonstration of this algorithm in our program, we've highlighted the importance of proper resource management and the role of proactive decision-making in maintaining system integrity. As we continue to develop and refine our understanding of concurrency control mechanisms, leveraging techniques like the Banker's Algorithm will remain essential for creating robust and efficient software systems.