NAME:- Shreeram

GROUP:- D1

REG. NO:- 20214033

ASSIGNMENT - 07

1) **Implement Banker’s Algorithm considering P0 to P4 as shown in table and check if the system is in a safe state? If it is in safe state, determine the safe sequence.**

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

Ans:-

```
#include <stdio.h>
#define NUM_PROCESSES 5
#define NUM_RESOURCES 3
int max_claim[NUM_PROCESSES][NUM_RESOURCES] = {
    {4, 3, 3},
    {3, 2, 2},
    {9, 0, 2},
    {7, 5, 3},
    {1, 1, 2}
};
int allocated_resources[NUM_PROCESSES][NUM_RESOURCES] =
{
    {1, 1, 2},
    {2, 1, 2},
    {4, 0, 1},
    {0, 2, 0},
    {1, 1, 2}
};
int available_resources[NUM_RESOURCES] = {2, 1, 0};
int need[NUM_PROCESSES][NUM_RESOURCES];
int work[NUM_RESOURCES];
int finish[NUM_PROCESSES];
int safe_sequence[NUM_PROCESSES];
void calculate_need() {
    for (int i = 0; i < NUM_PROCESSES; i++) {
        for (int j = 0; j < NUM_RESOURCES; j++) {
            need[i][j] = max_claim[i][j] - allocated_resources[i][j];
        }
    }
}
int find_safe_sequence() {
    for (int i = 0; i < NUM_RESOURCES; i++) {
        work[i] = available_resources[i];
    }
    for (int i = 0; i < NUM_PROCESSES; i++) {
        finish[i] = 0;
    }
    int safe_count = 0;
    while (safe_count < NUM_PROCESSES) {
        int found = 0;
        for (int i = 0; i < NUM_PROCESSES; i++) {
            if (!finish[i]) {
                int j;
                for (j = 0; j < NUM_RESOURCES; j++) {
                    if (need[i][j] > work[j]) {
                        break;
                    }
                }
                if (j == NUM_RESOURCES) {
                    for (j = 0; j < NUM_RESOURCES; j++) {
                        work[j] += allocated_resources[i][j];
                    }
                    safe_sequence[safe_count] = i;
                    finish[i] = 1;
                    safe_count++;
                    found = 1;
                }
            }
        }
        if (!found) {
            return 0;
        }
    }
    return 1;
}

int main() {
    calculate_need();
    if (find_safe_sequence()) {
        printf("System is in a safe state.\n");
        printf("Safe Sequence: ");
        for (int i = 0; i < NUM_PROCESSES; i++) {
            printf("P%d", safe_sequence[i]);
            if (i < NUM_PROCESSES - 1) {
                printf(" -> ");
            }
        }
        printf("\n");
    } else {
        printf("System is not in a safe state. Deadlock detected.\n");
    }
    return 0;
}
```

```
/*OUTPUT
System is in a safe state.
Safe Sequence: P1 -> P4 -> P0 -> P2 -> P3
*/
```

2) Write a multithreaded program that implements the banker's algorithm. Create n threads that request and release resources from the bank. The banker will grant the request only if it leaves the system in a safe state. You may write this program using Pthreads. It is important that shared data be safe from concurrent access. To ensure safe access to shared data, you can use mutex locks, which are available in the Pthreads.

Ans:-

```
#include <stdio.h>
#include <pthread.h>
```

```
#include <stdbool.h>
#include<stdlib.h>
#define NUM_RESOURCES 3
#define NUM_PROCESSES 5
int available[NUM_RESOURCES] = {10, 5, 7};
int max_claim[NUM_PROCESSES][NUM_RESOURCES] = {
    {7, 5, 3},
    {3, 2, 2},
    {9, 0, 2},
    {2, 2, 2},
    {4, 3, 3}
};
int allocated[NUM_PROCESSES][NUM_RESOURCES] = {
    {0, 1, 0},
    {2, 0, 0},
    {3, 0, 2},
    {2, 1, 1},
    {0, 0, 2}
};
pthread_mutex_t mutex;
bool is_safe_state(int process_id, int request[NUM_RESOURCES]) {
    // Simulate resource allocation and check for safety
    int work[NUM_RESOURCES];
    int finish[NUM_PROCESSES];
    for (int i = 0; i < NUM_RESOURCES; i++) {
        work[i] = available[i] - request[i];
    }
    for (int i = 0; i < NUM_PROCESSES; i++) {
        finish[i] = false;
    }
    int count = 0;
    while (count < NUM_PROCESSES) {
        bool found = false;
        for (int i = 0; i < NUM_PROCESSES; i++) {
            if (!finish[i]) {
                bool can_allocate = true;
                for (int j = 0; j < NUM_RESOURCES; j++) {
                    if (max_claim[i][j] - allocated[i][j] > work[j]) {
                        can_allocate = false;
                        break;
                    }
                }
                if (can_allocate) {
                    for (int j = 0; j < NUM_RESOURCES; j++) {
                        work[j] += allocated[i][j];
                    }
                    finish[i] = true;
                    found = true;
                    count++;
                }
            }
        }
        if (!found) {
            return false;
        }
    }
    return true;
}
void request_resources(int process_id, int
request[NUM_RESOURCES]) {
    pthread_mutex_lock(&mutex);
    if (is_safe_state(process_id, request)) {
        // Grant the request and update data structures
        for (int i = 0; i < NUM_RESOURCES; i++) {
            available[i] -= request[i];
            allocated[process_id][i] += request[i];
        }
        printf("Process %d: Request granted\n", process_id);
    } else {
        printf("Process %d: Request denied (unsafe state)\n",
process_id);
    }
    pthread_mutex_unlock(&mutex);
}
void release_resources(int process_id, int
release[NUM_RESOURCES]) {
    pthread_mutex_lock(&mutex);
    for (int i = 0; i < NUM_RESOURCES; i++) {
        available[i] += release[i];
        allocated[process_id][i] -= release[i];
    }
    printf("Process %d: Resources released\n", process_id);
    pthread_mutex_unlock(&mutex);
}
void *process_thread(void *arg) {
    int process_id = *(int *)arg;
    int request[NUM_RESOURCES];
    int release[NUM_RESOURCES];
   for (int i = 0; i < NUM_RESOURCES; i++) {
        request[i] = rand() % (max_claim[process_id][i] + 1);
        release[i] = rand() % (allocated[process_id][i] + 1);
    }
    request_resources(process_id, request);
    release_resources(process_id, release);
    pthread_exit(NULL);
}

int main() {
    pthread_t threads[NUM_PROCESSES];
    int thread_args[NUM_PROCESSES];

    pthread_mutex_init(&mutex, NULL);

     for (int i = 0; i < NUM_PROCESSES; i++) {
        thread_args[i] = i;
        pthread_create(&threads[i], NULL, process_thread,
&thread_args[i]);
```

```
    }

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

/*OUTPUT
Process 3: Request granted
Process 3: Resources released
Process 4: Request granted
Process 4: Resources released
Process 0: Request denied (unsafe state)
Process 0: Resources released
Process 1: Request granted
Process 1: Resources released
Process 2: Request granted
Process 2: Resources released
*/
```