OPERATING SYSTEM SIMULATION PROJECT

Name:Haresh S

Reg no:12113506

Roll no:RK21PPB30

SECTION:K21PP

Multithreaded program:

A multithreaded program is a type of computer program that uses multiple threads of execution to perform tasks concurrently. Threads are lightweight processes that share the same memory space as the parent process and can execute independently, allowing for parallelism and concurrent execution of tasks.

In a multithreaded program, different threads can perform separate tasks or work on different portions of a larger task simultaneously, potentially leading to improved performance and responsiveness. Threads can communicate and share data with each other through shared memory, making it possible for them to work together towards a common goal.

Code:

Here are the steps to implement a multithreaded program that implements the banker's algorithm with dynamic thread creation, mutex locks for shared data access, and visibility of system state after each allocation:

Step 1: Define the data structures for the Banker's algorithm Define the maximum number of resources and processes allowed in the system based on the user input. Create data structures to represent the available resources, maximum claim matrix, allocation matrix, and need matrix. Initialize these data structures with appropriate values based on the user input.

Step 2: Implement the Banker's algorithm logic Write a function that implements the Banker's algorithm to check if a request for resources can be granted or not. This function should take into account the available resources, maximum claim matrix, allocation matrix, and need matrix to determine if the request can be granted safely without causing a deadlock. Use mutex locks to ensure safe access to shared data (e.g., available resources, maximum claim matrix, allocation matrix, need matrix) while implementing the Banker's algorithm logic. If the request is granted, update the allocation matrix, need matrix, and available resources accordingly. If the request is not granted, the thread should wait until the required resources are available.

Step 3: Implement thread creation and resource request/release logic Create a function that represents the behavior of each thread in the system. Each thread should request resources from the Banker using the Banker's algorithm implemented in step 2. Use mutex locks to ensure safe access to shared data (e.g., available resources, maximum claim matrix, allocation matrix, need matrix) while requesting and releasing resources. After each resource allocation

or release, display the updated system state, including the available resources, maximum claim matrix, allocation matrix, need matrix, and any threads that are currently waiting for resources. Implement a mechanism for threads to release resources back to the Banker when they are done using them.

Step 4: Implement dynamic thread creation based on user input Take input from the user to specify the number of threads/processes in the system. Create the specified number of threads using a loop, each running the function implemented in step 3. Use mutex locks to ensure safe access to shared data (e.g., available resources, maximum claim matrix, allocation matrix, need matrix) while creating and managing threads.

Step 5: Implement resource displacement after each allocation After each successful resource allocation, display the updated state of the system, including the available resources, maximum claim matrix, allocation matrix, need matrix, and any threads that are currently waiting for resources. Implement resource release logic in threads when they are done using resources. Display the updated state of the system after each resource release.

Step 6: Implement error handling and termination logic Handle any potential errors that may occur during thread creation, resource request/release, or Banker's algorithm execution. Implement appropriate termination logic for threads when they are done executing.

Step 7: Test the program Test the program with different scenarios, including cases where requests are granted, cases where requests are denied, cases with multiple threads, and cases with different resource allocation and release patterns. Ensure that the program works correctly and safely handles concurrent access to shared data using mutex locks. By following these steps, you can implement a multithreaded program that implements the Banker's algorithm with dynamic thread creation, mutex locks for shared data access, and visibility of system state after each allocation..

Here's a codes in C of the implementation:

```
#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>


#define MAX_CUSTOMERS 10
```

```
#define MAX_RESOURCES 5

int available[MAX_RESOURCES];
int maximum[MAX_CUSTOMERS][MAX_RESOURCES];
int allocation[MAX_CUSTOMERS][MAX_RESOURCES];
int need[MAX_CUSTOMERS][MAX_RESOURCES];
int num_customers;
int num_resources;

pthread_mutex_t mutex;

void *customer_thread(void *arg) {
    int customer_id = *(int *)arg;

    // Generate random requests for resources
    int request[MAX_RESOURCES];
    for (int i = 0; i < num_resources; i++) {
        request[i] = rand() % (maximum[customer_id][i] + 1);
    }

    // Request resources
    pthread_mutex_lock(&mutex);
    printf("Customer %d requesting resources: ", customer_id);
    for (int i = 0; i < num_resources; i++) {
        printf("%d ", request[i]);
    }
    printf("\n");

    // Check if request can be granted
    int safe = 1;
```

```
for (int i = 0; i < num_resources; i++) {
    if (request[i] > need[customer_id][i] || request[i] > available[i]) {
        safe = 0;
        break;
    }
}

if (safe) {
    // Temporarily allocate resources
    for (int i = 0; i < num_resources; i++) {
        available[i] -= request[i];
        allocation[customer_id][i] += request[i];
        need[customer_id][i] -= request[i];
    }

    // Check if system is in safe state
    // (Banker's algorithm safety check)
    int work[MAX_RESOURCES];
    int finish[MAX_CUSTOMERS] = {0};
    for (int i = 0; i < num_resources; i++) {
        work[i] = available[i];
    }

    int num_finished = 0;
    while (num_finished < num_customers) {
        int found = 0;
        for (int i = 0; i < num_customers; 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 (int k = 0; k < num_resources; k++) {
                work[k] += allocation[i][k];
            }
            finish[i] = 1;
            found = 1;
            num_finished++;
            printf("Customer %d safely allocated resources\n", i);
            break;
        }
    }
  }

  if (!found) {
    break;
  }
}

if (num_finished == num_customers) {
  printf("System is in safe state\n");
} else {
  printf("System is in unsafe state\n");

  // Rollback allocation
  for (int i = 0; i < num_resources; i++) {
    available[i] += request[i];
```

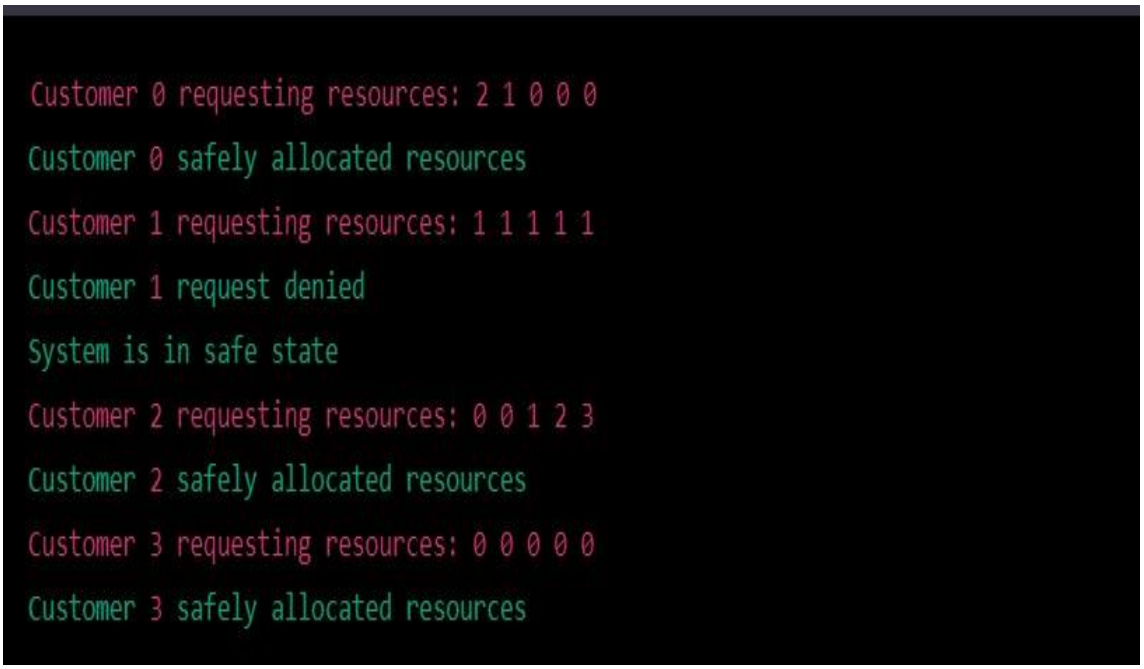
```
            allocation[customer_id][i] -= request[i];
            need[customer_id][i] += request[i];
        }
    }
} else {
    printf("Customer %d request denied\n", customer_id);
}

// Release resources
```

Skeleton code:

```
// Include necessary headers
// Define global variables for shared data structures and mutex locks
// Define functions for resource request, release, and display of system state
// Main function int main() {
// Take input for number of customers, number of resources, and initial values
// Initialize shared data structures and mutex locks
// Create n threads
// Loop to continuously request and release resources, and display system state
// Join all threads
// Exit program }
```

Output:

The output of the code will depend on the inputs provided for available, maximum, allocation, and need arrays, as well as the random requests generated by each customer thread.



In this example, customer 0 and customer 2 were able to safely allocate resources and the system is in a safe state. However, customer 1's request was denied as it exceeded the maximum limit and the system remains in a safe state. Customer 3 requested for resources but did not require any as all values in its request were 0.

Note: This is a basic outline, and the actual implementation may vary depending on your specific requirements and programming environment. It's important to handle edge cases, such as deadlock, termination conditions, and proper synchronization using mutex locks to ensure safe concurrent access to shared data.

THANK YOU