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 <unistd.h>
            #include <pthread.h>
            #include <stdbool.h>
            #include <time.h>

            int nResources,
                nProcesses;
            int *resources;
            int **allocated;
```

```
int **maxRequired;
int **need;
int *safeSeq;
int nProcessRan = 0;

pthread_mutex_t lockResources;
pthread_cond_t condition;

// get safe sequence is there is one else return false
bool getSafeSeq();
// process function
void* processCode(void* );

int main(int argc, char** argv) {
       srand(time(NULL));

        printf("\nNumber of processes? ");
        scanf("%d", &nProcesses);

        printf("\nNumber of resources? ");
        scanf("%d", &nResources);

        resources = (int *)malloc(nResources * sizeof(*resources));
        printf("\nCurrently Available resources (R1 R2 ...)? ");
        for(int i=0; i<nResources; i++)
                scanf("%d", &resources[i]);

        allocated = (int **)malloc(nProcesses * sizeof(*allocated));
        for(int i=0; i<nProcesses; i++)
                allocated[i] = (int *)malloc(nResources *
sizeof(**allocated));

        maxRequired = (int **)malloc(nProcesses * sizeof(*maxRequired));
        for(int i=0; i<nProcesses; i++)
                maxRequired[i] = (int *)malloc(nResources *
sizeof(**maxRequired));

        // allocated
        printf("\n");
        for(int i=0; i<nProcesses; i++) {
                printf("\nResource allocated to process %d (R1 R2 ...)? ",
i+1);
                for(int j=0; j<nResources; j++)
                        scanf("%d", &allocated[i][j]);
        }
        printf("\n");

       // maximum required resources
        for(int i=0; i<nProcesses; i++) {
```

```
                printf("\nMaximum resource required by process %d (R1 R2
...)? ", i+1);
                for(int j=0; j<nResources; j++)
                        scanf("%d", &maxRequired[i][j]);
        }
        printf("\n");

       // calculate need matrix
        need = (int **)malloc(nProcesses * sizeof(*need));
        for(int i=0; i<nProcesses; i++)
                need[i] = (int *)malloc(nResources * sizeof(**need));

        for(int i=0; i<nProcesses; i++)
                for(int j=0; j<nResources; j++)
                        need[i][j] = maxRequired[i][j] - allocated[i][j];

       // get safe sequence
       safeSeq = (int *)malloc(nProcesses * sizeof(*safeSeq));
        for(int i=0; i<nProcesses; i++) safeSeq[i] = -1;

        if(!getSafeSeq()) {
                printf("\nUnsafe State! The processes leads the system to a
unsafe state.\n\n");
                exit(-1);
        }

        printf("\n\nSafe Sequence Found : ");
        for(int i=0; i<nProcesses; i++) {
                printf("%-3d", safeSeq[i]+1);
        }

        printf("\nExecuting Processes...\n\n");
        sleep(1);

       // run threads
       pthread_t processes[nProcesses];
        pthread_attr_t attr;
        pthread_attr_init(&attr);

       int processNumber[nProcesses];
       for(int i=0; i<nProcesses; i++) processNumber[i] = i;

        for(int i=0; i<nProcesses; i++)
                pthread_create(&processes[i], &attr, processCode, (void
*)(&processNumber[i]));

        for(int i=0; i<nProcesses; i++)
                pthread_join(processes[i], NULL);

        printf("\nAll Processes Finished\n");
```

```
        // free resources
         free(resources);
         for(int i=0; i<nProcesses; i++) {
                 free(allocated[i]);
                 free(maxRequired[i]);
                free(need[i]);
         }
         free(allocated);
         free(maxRequired);
        free(need);
         free(safeSeq);
}


bool getSafeSeq() {
        // get safe sequence
         int tempRes[nResources];
         for(int i=0; i<nResources; i++) tempRes[i] = resources[i];

         bool finished[nProcesses];
         for(int i=0; i<nProcesses; i++) finished[i] = false;
         int nfinished=0;
         while(nfinished < nProcesses) {
                 bool safe = false;

                 for(int i=0; i<nProcesses; i++) {
                         if(!finished[i]) {
                                 bool possible = true;

                                 for(int j=0; j<nResources; j++)
                                         if(need[i][j] > tempRes[j]) {
                                                 possible = false;
                                                 break;
                                         }

                                 if(possible) {
                                         for(int j=0; j<nResources; j++)
                                                 tempRes[j] +=
allocated[i][j];
                                         safeSeq[nfinished] = i;
                                         finished[i] = true;
                                         ++nfinished;
                                         safe = true;
                                 }
                         }
                 }

                 if(!safe) {
                         for(int k=0; k<nProcesses; k++) safeSeq[k] = -1;
```

```
                        return false; // no safe sequence found
                }
        }
        return true; // safe sequence found
}

// process code
void* processCode(void *arg) {
        int p = *((int *) arg);

        // lock resources
        pthread_mutex_lock(&lockResources);

        // condition check
        while(p != safeSeq[nProcessRan])
                pthread_cond_wait(&condition, &lockResources);

        // process
        printf("\n--> Process %d", p+1);
        printf("\n\tAllocated : ");
        for(int i=0; i<nResources; i++)
                printf("%3d", allocated[p][i]);

        printf("\n\tNeeded    : ");
        for(int i=0; i<nResources; i++)
                printf("%3d", need[p][i]);

        printf("\n\tAvailable : ");
        for(int i=0; i<nResources; i++)
                printf("%3d", resources[i]);

        printf("\n"); sleep(1);

        printf("\tResource Allocated!");
        printf("\n"); sleep(1);
        printf("\tProcess Code Running...");
        printf("\n"); sleep(rand()%3 + 2); // process code
        printf("\tProcess Code Completed...");
        printf("\n"); sleep(1);
        printf("\tProcess Releasing Resource...");
        printf("\n"); sleep(1);
        printf("\tResource Released!");

        for(int i=0; i<nResources; i++)
                resources[i] += allocated[p][i];

        printf("\n\tNow Available : ");
        for(int i=0; i<nResources; i++)
                printf("%3d", resources[i]);
        printf("\n\n");
```

```
            sleep(1);

            // condition broadcast
            nProcessRan++;
            pthread_cond_broadcast(&condition);
            pthread_mutex_unlock(&lockResources);
            pthread_exit(NULL);
        }
```

Skeleton code:

// Global variables

// Function to get safe sequence or return false

// Process function

// Initialization

// Input number of processes and resources

// Allocate memory for resources, allocated, maxRequired, and need arrays

// Input available resources and allocation matrix

// Calculate need matrix

// Get safe sequence

// If no safe sequence found, exit with error code -1

// Output safe sequence

// Create threads for each process

// Join threads

// Free allocated memory

// Get safe sequence

// Initialize temporary resources array and finished array

// While not all processes finished

// For each process

// If process not finished

// Check if possible to execute process

// If possible, update temporary resources array, mark process as finished,

// and add process to safe sequence

// If no process could be executed, return false

// If all processes finished, return true

// Process code

// Lock resources mutex

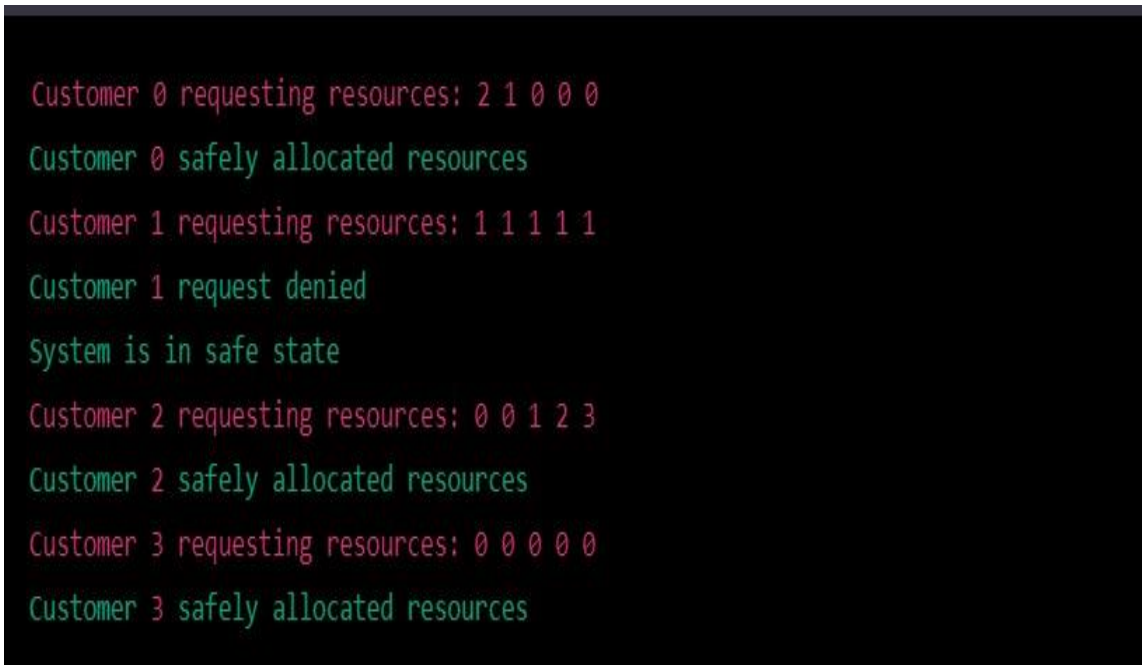
// Wait until it's the process's turn to execute

// Execute process by updating allocated resources and printing output

// Unlock resources mutex

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.

EXPLANATION

The given code is an implementation of the Banker's algorithm, which is used to prevent deadlock in a multi-process, multi-resource environment. The code takes user input for the number of processes, number of resources, resource allocation, and maximum resource requirements for each process. It then calculates the need matrix and finds a safe sequence of processes to execute in order to avoid deadlock.

The code uses multi-threading to simulate the concurrent execution of processes. It creates threads for each process and executes them concurrently. Each thread represents a process and runs the processCode function, which simulates the execution of a process. The processCode function first acquires a lock on the lockResources mutex to prevent multiple threads from accessing the resources simultaneously. It then checks if the process can request resources based on the available resources and the need matrix. If the process can request resources, it updates the allocated resources and releases the lock on lockResources. If the process cannot request resources, it waits on the condition variable until it can. Once the process finishes executing, it releases the allocated resources and signals the condition variable to wake up other threads waiting on it.

The getSafeSeq function is used to find a safe sequence of processes to execute. It uses the Banker's algorithm to check if there is a safe sequence of processes that can execute without causing deadlock. If a safe sequence is found, it returns true; otherwise, it returns false.

Note: It's important to ensure that the lockResources mutex and condition variable are properly initialized before using them in the code. Also, proper error handling and memory deallocation should be implemented to avoid potential issues.

THANK YOU