LOVELY PROFESSIONAL UNIVERSITY

Transforming Education Transforming India

Report

Course: Opearting System

SUBMITTED BY:

Name: PRABIN TELI

Reg_no: 12101991

SUBMITTED TO:

Name: NEHA

Question:

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. It is important that shared data be safe from concurrent access. To ensure safe access to shared data, you can use mutex locks.

Ensure:

The program should be dynamic such that the threads are created at run time based on the input from the user.

The resources must be displaced after each allocation.

The system state should be visible after each allocation.

---

1. What id Banker's Algorithms?

➡ The banker's algorithm is a resource allocation and deadlock avoidance algorithm that tests for safety by simulating the allocation for the predetermined maximum possible amounts of all resources, then makes a check to test for possible activities, before deciding whether allocation should be allowed to continue.

➡ Multithreading

Multithreading is a Java feature that allows concurrent execution of two or more parts of a program for maximum utilization of CPU. Each part of such program is called a thread. So, threads are light-weight processes within a process.

2. Advantages & Disadvantages

Advantages	Disadvantages
• It contains various resources that meet the requirements of each process. • Each process should provide information to the operating system for upcoming resource requests, the number of resources, and how long the resources will be held. • It helps the operating system manage and control process requests for each type of resource in the computer system	• It requires a fixed number of processes, and no additional processes can be started in the system while executing the process. • The algorithm does no longer allows the processes to exchange its maximum needs while processing its tasks. • Each process has to know and state their maximum resource requirement in advance for the system.

3. **When working with a banker's algorithm, it requests to know about three things:**

1. How much each process can request for each resource in the system. It is denoted by the [**MAX**] request.
2. How much each process is currently holding each resource in a system. It is denoted by the [**ALLOCATED**] resource.
3. It represents the number of each resource currently available in the system. It is denoted by the [**AVAILABLE**] resource.

4. The following Operating System are used to implement the Banker'sAlgorithm:

Let **'n'** be the number of processes in the system and **'m'** be the number of resourcetypes.

Available:

- It is a 1-d array of size **'m'** indicating the number of available resources of each type.
- Available [j] = k means there are **'k'** instances of resource type $\mathbf{R_j}$

Max:

- It is a 2-d array of size '**n*m'** that defines the maximum demand of each process in a system.
- Max [i, j] = k means process $\mathbf{P_i}$ may request at most **'k'** instances of resource type $\mathbf{R_j}$.

Allocation:

- It is a 2-d array of size **'n*m'** that defines the number of resources of each type currently allocated to each process.
- Allocation [i, j] = k means process $\mathbf{P_i}$ is currently allocated **'k'** instances of resource type $\mathbf{R_j}$

Need:

- It is a 2-d array of size **'n*m'** that indicates the remaining resource need of each process.
- Need [i, j] = k means process $\mathbf{P_i}$ currently needs **'k'** instances of resource type $\mathbf{R_j}$
- Need [i, j] = Max [i, j] – Allocation [i, j]

5. Step-by-step guide to creating a multithreaded program that implements the banker's algorithm:

1. Define the necessary OS: You will need to define data structures to represent the available resources, maximum resources, allocation matrix, and need matrix. You should also define a data structure to represent a process, which includes its ID, current allocation, and maximum request.

2. Initialize the OS: You should initialize the data structures with appropriate values. For example, the available resources should be set to the total number of resources in the system, and the allocation matrix and need matrix should be initialized to zero.

3. Create the threads: Create n threads that will request and release resources from the bank. Each thread should have its own ID and a randomly generated maximum request. The threads should execute in an infinite loop, making requests and releases as needed.

4. Define the banker's algorithm: The banker's algorithm should be implemented in a separate function. This function will take the process ID and the number of resources requested as parameters. The algorithm should check if granting the request would leave the system in a safe state. If so, the request is granted and the allocation and need matrices are updated. If not, the request is denied.

5. Implement mutex locks: To ensure safe access to shared data, you should use mutex locks. The data structures that are accessed by multiple threads should be protected by mutex locks to prevent concurrent access.

6. Test the program: Run the program and test it with different scenarios. Verify that the program correctly grants and denies requests, and that the system always remains in a safe state.

CODE: -

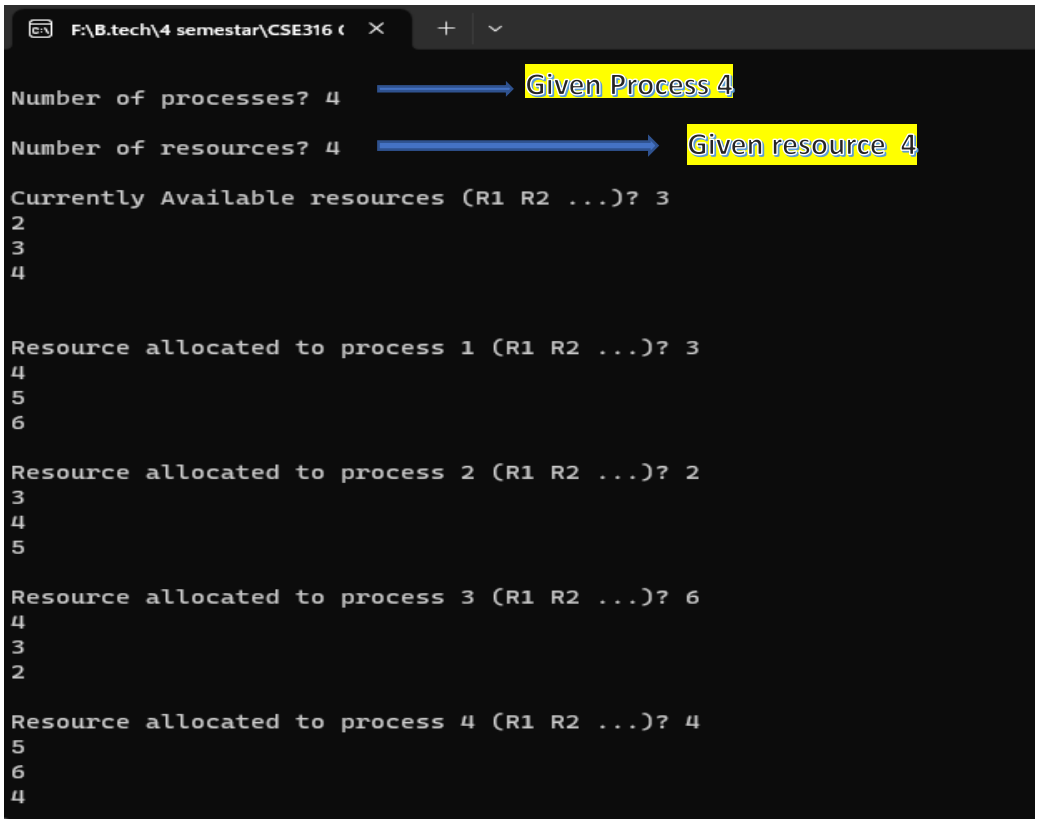
// A Multithreaded Program that implements the banker's algorithm.

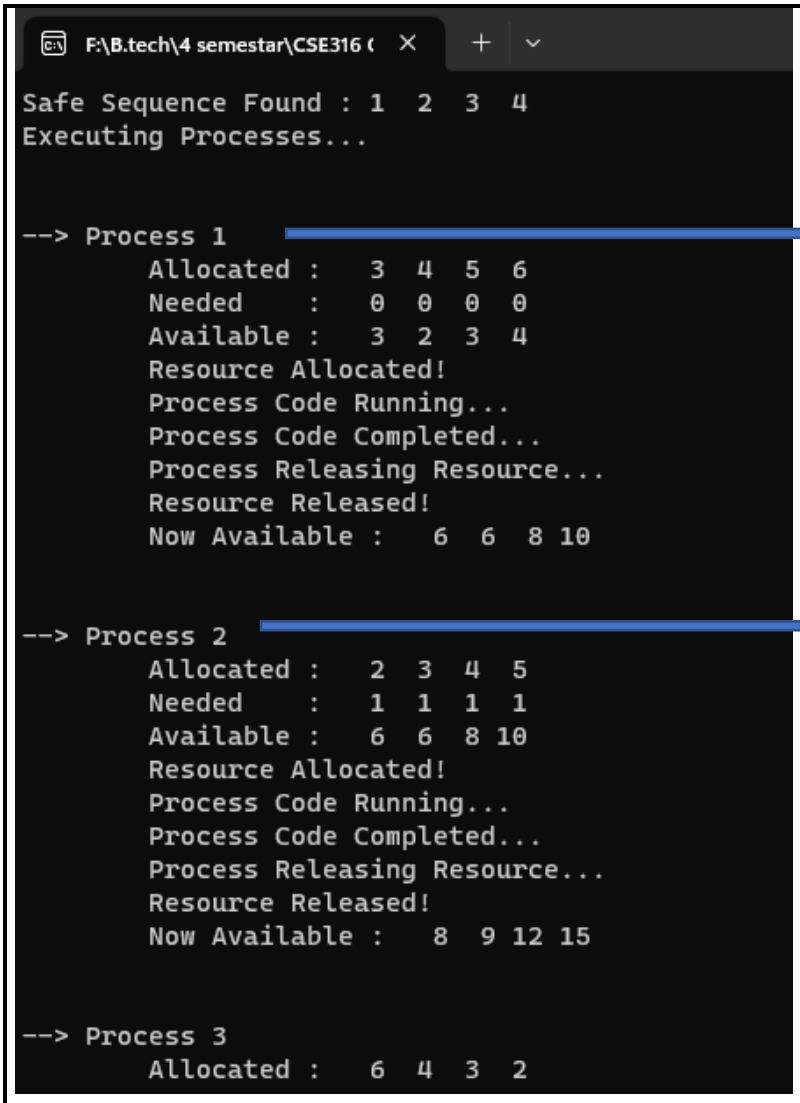
```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>
#include <stdbool.h>
#include <time.h>

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);
}
```

Output:





Output Process 1

Output Process 2

Output Process 3

Output Process 4