A black sign with white text

Description automatically generated

CSE-316: OPERATING SYSTEMS

Programming Assignment

Submitted to:

Dr. Baljit Singh Saini

Problem Statement

Q3. 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.

**Approach to solution**

Approach was to make a program which can adhere multiple processors and resources and to learn the basic about library so that things can be done smoothly.

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);

}

Explanation for above code

C does not contain any built-in support for multithreaded applications. Instead, it relies entirely upon the operating system to provide this feature.

This tutorial assumes that you are working on Linux OS and we are going to write multi-threaded C program using POSIX. POSIX Threads, or Pthreads provides API which are available on many Unix-like POSIX systems

**pthread\_create** creates a new thread and makes it executable. This routine can be called any number of times from anywhere within your code. Here is the description of The maximum number of threads that may be created by a process is implementation dependent. Once created, threads are peers, and may create other threads. There is no implied hierarchy or dependency between threads.

**Safe State:** No deadlock

**Unsafe State:** May or may not be deadlock

In multiple instances banker’s algorithm is used

This algorithm is also used for deadlock detection

Banker’s algorithm is also called deadlock avoidance algorithm

To avoid deadlock current availability should be greater than equal to allocated processors.

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

**Why Banker’s algorithm is named so?**  
Banker’s algorithm is named so because it is used in banking system to check whether loan can be sanctioned to a person or not. Suppose there are n number of account holders in a bank and the total sum of their money is S. If a person applies for a loan, then the bank first subtracts the loan amount from the total money that bank has and if the remaining amount is greater than S then only the loan is sanctioned. It is done because if all the account holders come to withdraw their money then the bank can easily do it.

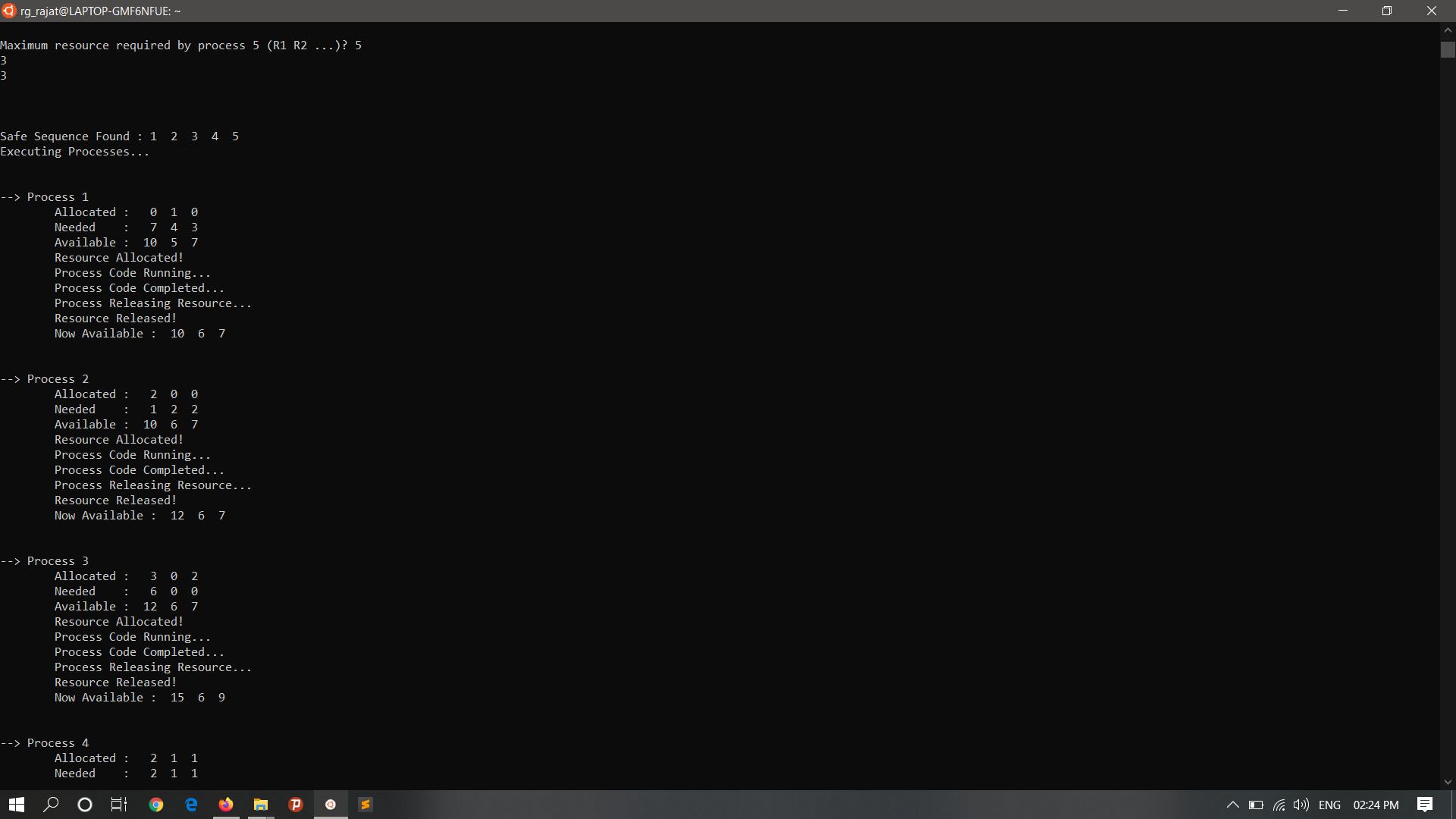
In other words, the bank would never allocate its money in such a way that it can no longer satisfy the needs of all its customers. The bank would try to be in safe state always.

Snapshots and Explanations

A screenshot of a computer

Description automatically generated

Entering the number of process and resources and then allocating resources to processes.



Remaining need= maximum need- allocated resource

A screenshot of a computer screen

Description automatically generated

Therefore, It is a safe state as all resources are allocated and hence there is no deadlock.