**Name: KESHAV KRISHNA**

**Registration Number:11810733**

**Email Address:keshavkrishna2211@gmail.com**

**GitHub Link:github.com/keshavkrishna2211/operating-system**

**Code:**

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <pthread.h>

#include <stdbool.h>

#include <time.h>

int Resource,Proc;

int \*res;

int \*\*allo;

int \*\*maxReq;

int \*\*need;

int \*safeSeq;

int nProcRan = 0;

pthread\_mutex\_t lockRes;

pthread\_cond\_t cond;

bool getSafeSeq();

void\* processCode(void\* );

int main(int argc, char\*\* argv) {

int i,j;

srand(time(NULL));

printf("\nNumber of processes? ");

scanf("%d", &Proc);

printf("\nNumber of resources? ");

scanf("%d", &Resource);

res = (int \*)malloc(Resource \* sizeof(\*res));

printf("\nCurrently Available resources (R1 R2 ...)? ");

for(i=0; i<Resource; i++)

scanf("%d", &res[i]);

allo = (int \*\*)malloc(Proc \* sizeof(\*allo));

for( i=0; i<Proc; i++)

allo[i] = (int \*)malloc(Resource \* sizeof(\*\*allo));

maxReq = (int \*\*)malloc(Proc \* sizeof(\*maxReq));

for(i=0; i<Proc; i++)

maxReq[i] = (int \*)malloc(Resource \* sizeof(\*\*maxReq));

printf("\n");

for(i=0; i<Proc; i++) {

printf("\nResource allocated to process %d (R1 R2 ...)? ", i+1);

for(j=0; j<Resource; j++)

scanf("%d", &allo[i][j]);

}

printf("\n");

for(i=0; i<Proc; i++) {

printf("\nMaximum resource required by process %d (R1 R2 ...)? ", i+1);

for(j=0; j<Resource; j++)

scanf("%d", &maxReq[i][j]);

}

printf("\n");

need = (int \*\*)malloc(Proc \* sizeof(\*need));

for(i=0; i<Proc; i++)

need[i] = (int \*)malloc(Resource \* sizeof(\*\*need));

for(i=0; i<Proc; i++)

for(j=0; j<Resource; j++)

need[i][j] = maxReq[i][j] - allo[i][j];

safeSeq = (int \*)malloc(Proc \* sizeof(\*safeSeq));

for(i=0; i<Proc; 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(i=0; i<Proc; i++) {

printf("%-3d", safeSeq[i]+1);

}

printf("\nProcesses Exexcuting...\n\n");

sleep(1);

pthread\_t processes[Proc];

pthread\_attr\_t attr;

pthread\_attr\_init(&attr);

int processNumber[Proc];

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

for(i=0; i<Proc; i++)

pthread\_create(&processes[i], &attr, processCode, (void \*)(&processNumber[i]));

for(i=0; i<Proc; i++)

pthread\_join(processes[i], NULL);

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

free(res);

for(i=0; i<Proc; i++) {

free(allo[i]);

free(maxReq[i]);

free(need[i]);

}

free(allo);

free(maxReq);

free(need);

free(safeSeq);

}

bool getSafeSeq() {

int i,k,j;

int tempRes[Resource];

for(i=0; i<Resource ;i++) tempRes[i] = res[i];

bool finished[Proc];

for(i=0; i<Proc; i++) finished[i] = false;

int nfinished=0;

while(nfinished < Proc) {

bool safe = false;

for(i=0; i<Proc; i++) {

if(!finished[i]) {

bool possible = true;

for(j=0; j<Resource; j++)

if(need[i][j] > tempRes[j]) {

possible = false;

break;

}

if(possible) {

for(j=0; j<Resource; j++)

tempRes[j] += allo[i][j];

safeSeq[nfinished] = i;

finished[i] = true;

++nfinished;

safe = true;

}

}

}

if(!safe) {

for(k=0; k<Proc; k++) safeSeq[k] = -1;

return false;

}

}

return true;

}

void\* processCode(void \*arg) {

int i;

int p = \*((int \*) arg);

pthread\_mutex\_lock(&lockRes);

while(p != safeSeq[nProcRan])

pthread\_cond\_wait(&cond, &lockRes);

printf("\n--> Process %d", p+1);

printf("\n\tAllocated : ");

for(i=0; i<Resource; i++)

printf("%3d", allo[p][i]);

printf("\n\tNeeded : ");

for(i=0; i<Resource; i++)

printf("%3d", need[p][i]);

printf("\n\tAvailable : ");

for(i=0; i<Resource; i++)

printf("%3d", res[i]);

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

printf("\tAllocated resources!");

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

printf("\tProcess Code Running...");

printf("\n"); sleep(rand()%3 + 2);

printf("\tProcess Code Completed...");

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

printf("\tProcess Releasing Resource...");

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

printf("\tResource Released!");

for(i=0; i<Resource; i++)

res[i] += allo[p][i];

printf("\n\tAvailable Resources: ");

for(i=0; i<Resource; i++)

printf("%3d", res[i]);

printf("\n\n");

sleep(1);

nProcRan++;

pthread\_cond\_broadcast(&cond);

pthread\_mutex\_unlock(&lockRes);

pthread\_exit(NULL);

}

**Problem:**

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.

**Description:**

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.

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 comes to withdraw their money then the bank can easily do it

**Algorithm:**

**Banker’s Algorithm**

**Safety Algorithm(n):**

1. Set i:=0
2. Repeat loop until i!=n:

Initialize: Work = Available

Finish[i] = false

1. Repeat loop until i!=n:

if Finish[i] != false and Needi <= Work

goto step 4

1. Work = Work + Allocation[i]

Finish[i] = true

goto step (2)

1. if Finish [i] = true for all i

then the system is in a **safe state**

Complexity for safe state:O(n\*n\*m) where n is the number of active process and m is the number of resource.

**Resource-Request Algorithm**

1) If Request[i] <= Need[i]

Goto step (2)

otherwise, raise an error condition

2) If Request[i] <= Available

Goto step (3)

otherwise, P[i] must wait

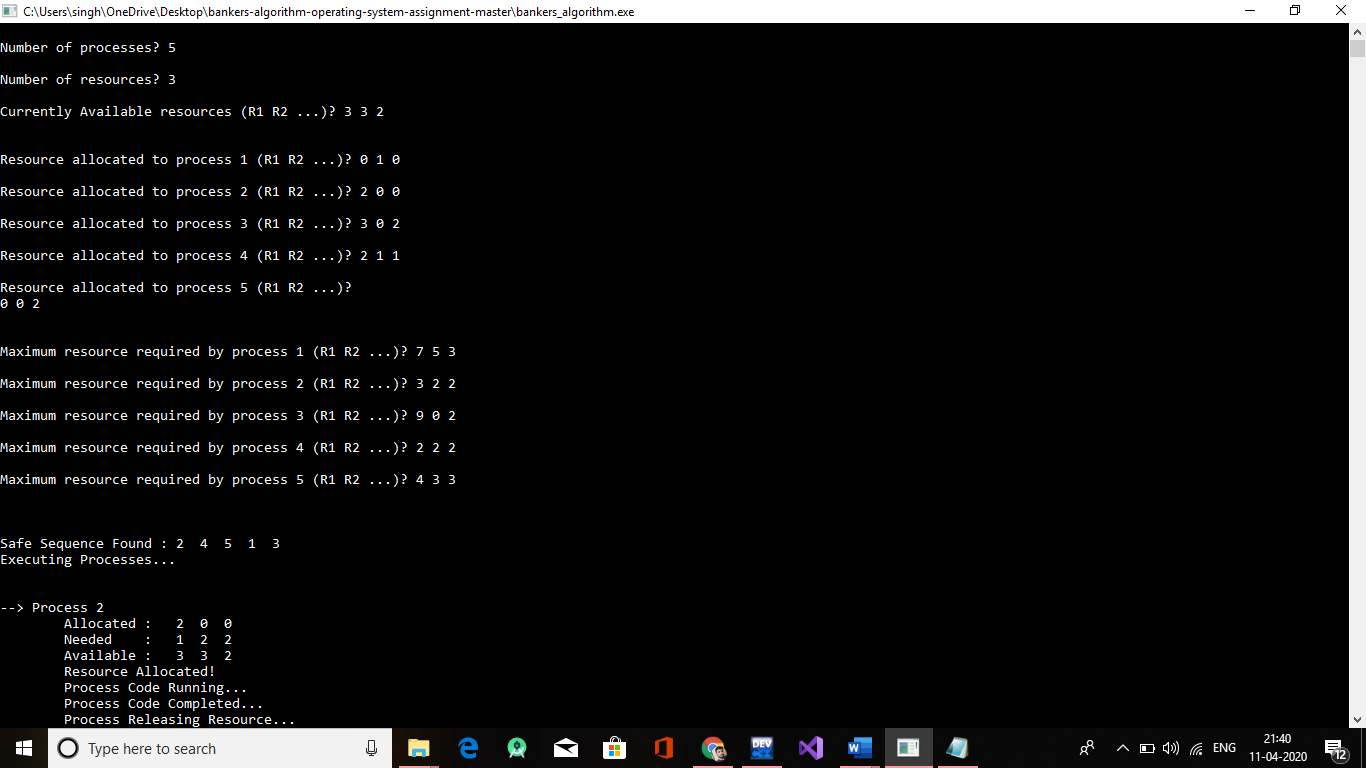
3) Available = Available – Request[i]

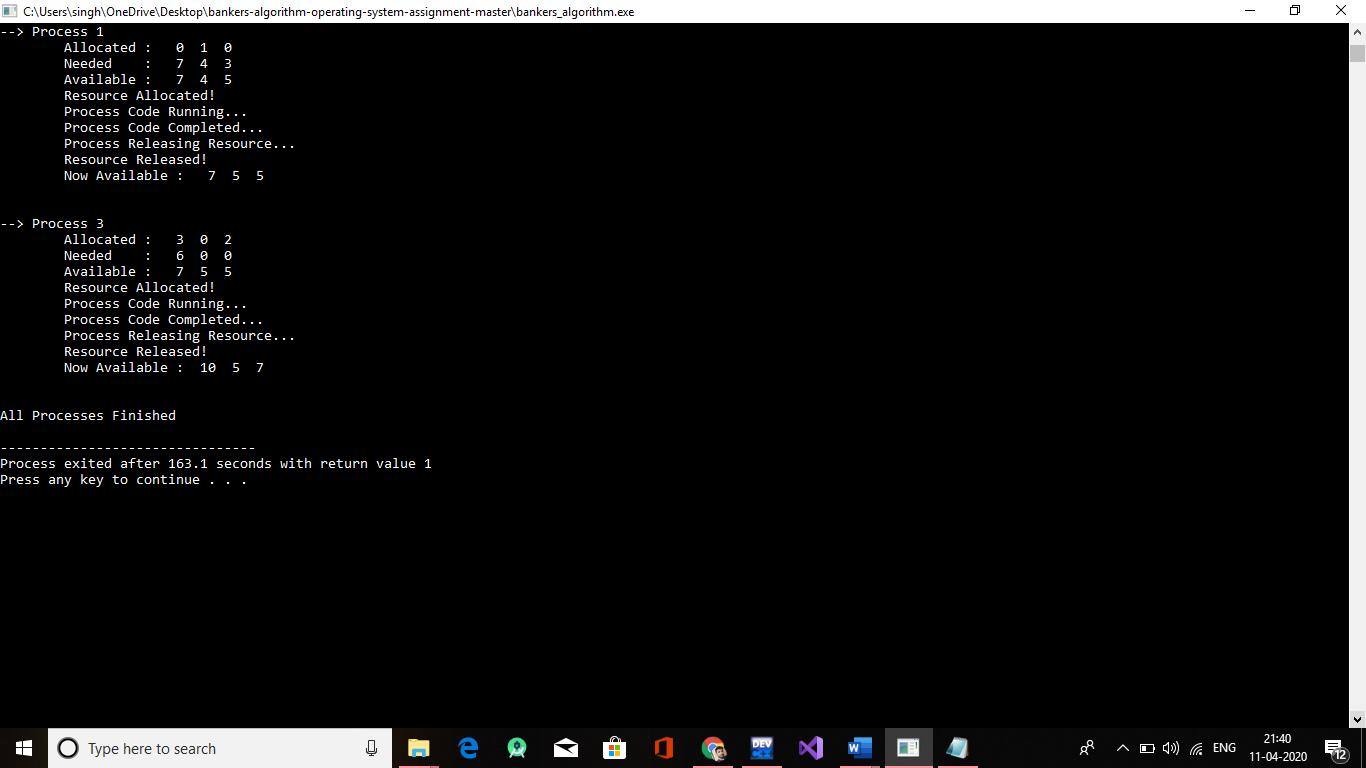
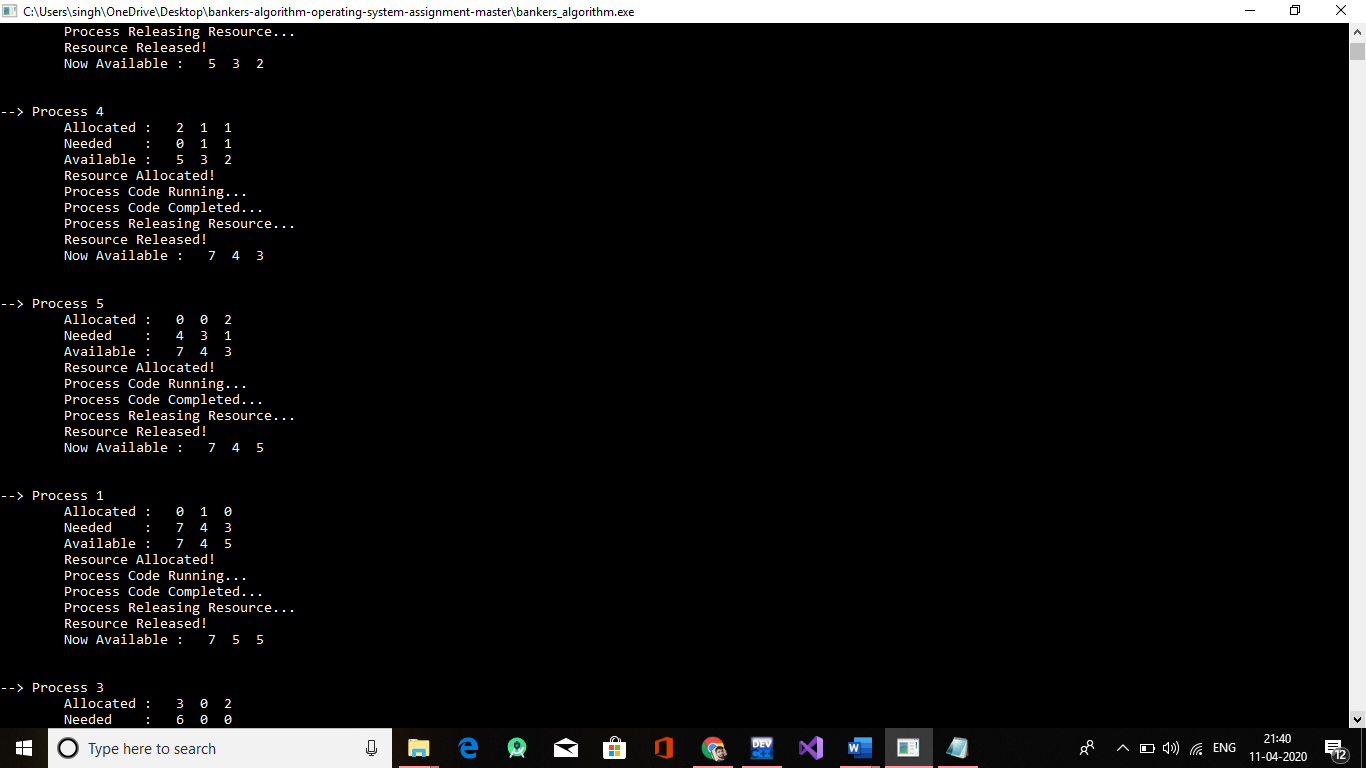
Allocation[i] = Allocation[i] + Request[i]

Need[i] = Need[i]– Request[i]

**Output:**

**Safe State:**





**Unsafe state:**

