

CSE-316: OPERATING SYSTEMS

Programming Assignment

PROBLEM STATEMENT

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.

CODE

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <pthread.h>

#include <stdbool.h>

#include <time.h>

int rs, ps ,yo;

int \*re, \*ss;

int \*\*al, \*\*mr , \*\*ne;

int np = 0;

pthread\_mutex\_t lr;

pthread\_cond\_t co;

bool SafeSequence()

{

int a[rs];

for(int i=0; i<rs; i++)

a[i] = re[i];

bool f[ps];

for(int i=0; i<ps; i++)

f[i] = false;

int fd=0;

while(fd < ps)

{

bool s = false;

for(int i=0; i<ps; i++)

{

if(!f[i])

{

bool bl = true;

for(int j=0; j<rs; j++)

if(ne[i][j] > a[j])

{

bl = false;

break;

}

if(bl)

{

for(int j=0; j<rs; j++)

a[j] += al[i][j];

ss[fd] = i;

f[i] = true;

++fd;

s = true;

}

}

}

if(!s)

{

for(int k=0; k<ps; k++) ss[k] = -1;

return false;

}

}

return true;

}

void\* MainResult(void \*arg)

{

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

pthread\_mutex\_lock(&lr);

while(p != ss[np])

pthread\_cond\_wait(&co, &lr);

printf("\nProcess %d", p+1);

printf("\n\tAllocated : ");

for(int i=0; i<rs; i++)

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

printf("\n\tNeeded : ");

for(int i=0; i<rs; i++)

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

printf("\n\tAvailable : ");

for(int i=0; i<rs; i++)

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

printf("\n");

printf("\tResource Allocated");

for(int i=0; i<rs; i++)

re[i] += al[p][i];

printf("\n\tNow Available : ");

for(int i=0; i<rs; i++)

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

printf("\n\n");

np++;

pthread\_cond\_broadcast(&co);

pthread\_mutex\_unlock(&lr);

pthread\_exit(NULL);

}

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

{

srand(time(NULL));

printf(" WELCOME");

do

{

printf("\nNumber of processes : ");

scanf("%d", &ps);

printf("\nNumber of resources : ");

scanf("%d", &rs);

re = (int \*)malloc(rs \* sizeof(\*re));

printf("\nCurrently Available resources : \n");

for(int i=0; i<rs; i++)

{

printf("%d : ",i+1);

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

}

al = (int \*\*)malloc(ps \* sizeof(\*al));

for(int i=0; i<ps; i++)

al[i] = (int \*)malloc(rs \* sizeof(\*\*al));

mr = (int \*\*)malloc(ps \* sizeof(\*mr));

for(int i=0; i<ps; i++)

mr[i] = (int \*)malloc(rs \* sizeof(\*\*mr));

printf("\n");

for(int i=0; i<ps; i++)

{

printf("\nResource allocated to process %d : \n", i+1);

for(int j=0; j<rs; j++)

{

printf("%d : ", j+1);

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

}

}

printf("\n");

for(int i=0; i<ps; i++)

{

printf("\nMaximum resource required by process %d : \n", i+1);

for(int j=0; j<rs; j++)

{

printf("%d : ", j+1);

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

}

}

printf("\n");

ne = (int \*\*)malloc(ps \* sizeof(\*ne));

for(int i=0; i<ps; i++)

ne[i] = (int \*)malloc(rs \* sizeof(\*\*ne));

for(int i=0; i<ps; i++)

for(int j=0; j<rs; j++)

ne[i][j] = mr[i][j] - al[i][j];

ss = (int \*)malloc(ps \* sizeof(\*ss));

for(int i=0; i<ps; i++) ss[i] = -1;

if(!SafeSequence())

{

printf("\nSystem is in unsafe state.\n\n");

exit(-1);

}

printf("\n\nSafe Sequence Found : ");

for(int i=0; i<ps; i++)

{

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

}

pthread\_t pr[ps];

pthread\_attr\_t at;

pthread\_attr\_init(&at);

int pn[ps];

for(int i=0; i<ps; i++) pn[i] = i;

for(int i=0; i<ps; i++)

pthread\_create(&pr[i], &at, MainResult, (void \*)(&pn[i]));

for(int i=0; i<ps; i++)

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

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

free(re);

for(int i=0; i<ps; i++)

{

free(al[i]);

free(mr[i]);

free(ne[i]);

}

free(al);

free(mr);

free(ne);

free(ss);

printf("You want to continue then you press '0' otherwise press '1'");

printf("\n0 = 'Yes'");

printf("\n1 = 'No'\n");

scanf("%d",&yo);

}while(yo != 1);

}

EXPLAINATION OF THE CODE

For the Banker’s Algorithm we mainly use two Algorithm

1. Safety Alogrithm
2. Resource – Request Algorithm

Here the following Algorithm as follow:

**Safetly Algorithm**

1. Let Work and Finish be the vectors. Initialize

Work = Available and Finish[i] = False

for i = 0,1,2,---,n-1.

1. Find an index i such that both

a.Finish[i] == False

b.Need i<=Work

If no such i exist, goto step 4.

1. Work = Work + Allocation;

Finish[i] = true

Goto step 2.

1. If Finish[i] == true for all i, then the system is in a safe state

**Resource – Request Algorithm**

1. If Request i<=Need i, goto step 2
2. If Request i<=Available, goto step 3 otherwise Pi must, since the resources are not available.
3. Have the system pretend to have allocate the requested resource to process Pi by modifying the state as follows

Available = Available – Request ;

Allocation i = Allocation i + Request i;

Need i = Need i – Request i;

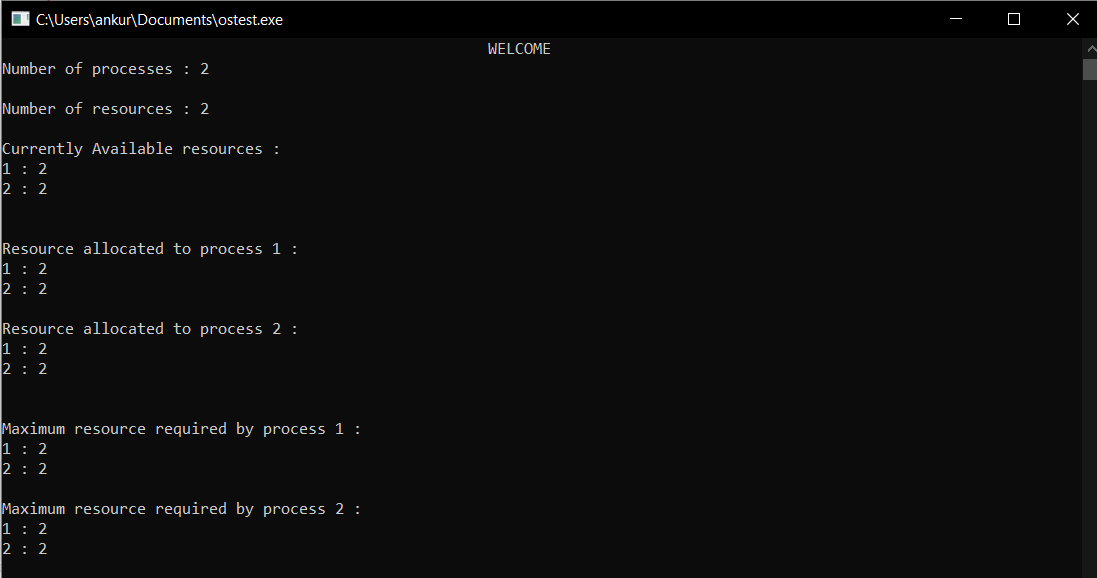
**pthread\_create** creates a new thread and makes it executable. This routine can be called any number of times from anywhere within your code. Here is the description of The maximum number of threads that may be created by a process is implementation dependent. Once created, threads are peers, and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

**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.

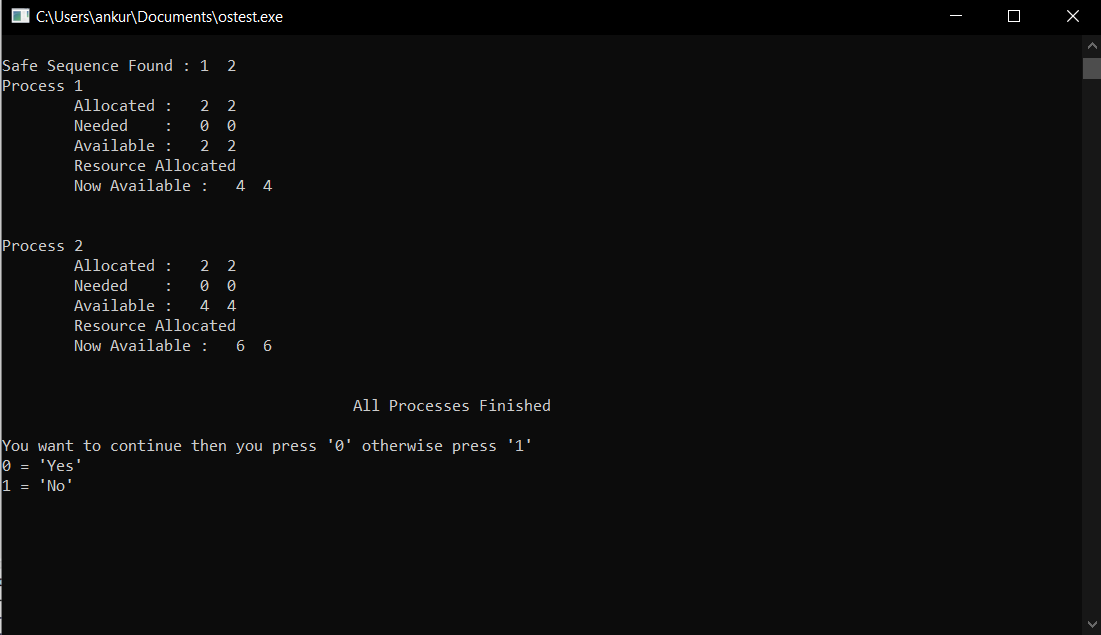
In other words, the bank would never allocate its money in such a way that it can no longer satisfy the needs of all its customers. The bank would try to be in safe state always.

**SCREENSHOT**

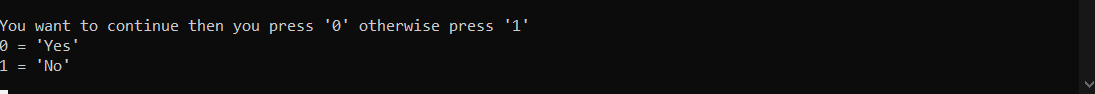
Entering the number of process and resources and then allocating resources to processes.



Remaining need= maximum need- allocated resource



If you want to check it again



If Deadlock occur then

