

Operating Systems

LAB ASESSMENT – 3

Name: **VIBHU KUMAR SINGH**

Reg. No: **19BCE0215**

Teacher: **Manikandan K.**

**Q18. Implement the program to pass messages using pipes.**

**A18.**

**CODE:**

#include <stdio.h>

#include <unistd.h>

#include<stdlib.h>

#define MSGSIZE 16

char\* msg1 = "Hazard #1";

char\* msg2 = "Chelsea #2";

char\* msg3 = "Pakistan #3";

int main(){

char inbuf[MSGSIZE];

int p[2], i;

if (pipe(p) < 0)

exit(1);

/\* continued \*/

/\* write pipe \*/

write(p[1], msg1, MSGSIZE);

write(p[1], msg2, MSGSIZE);

write(p[1], msg3, MSGSIZE);

for (i = 0; i < 3; i++) {

/\* read pipe \*/

read(p[0], inbuf, MSGSIZE);

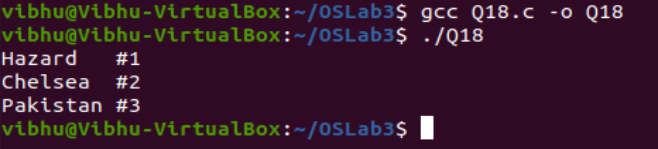
printf("%s\n", inbuf);

}

return 0;

}

**OUTPUT:**



**Q19. Write a program to demonstrate the implementation of Inter Process Communication (IPC) using shared memory.**

**A19.**

1. **CODE:**

#include<stdio.h>

#include<sys/ipc.h>

#include<sys/shm.h>

#include<sys/types.h>

#include<string.h>

#include<errno.h>

#include<stdlib.h>

#include<unistd.h>

#include<string.h>

#define BUF\_SIZE 1024

#define SHM\_KEY 0x1234

struct shmseg {

int cnt;

int complete;

char buf[BUF\_SIZE];

};

int fill\_buffer(char \* bufptr, int size);

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

int shmid, numtimes;

struct shmseg \*shmp;

char \*bufptr;

int spaceavailable;

shmid = shmget(SHM\_KEY, sizeof(struct shmseg), 0644|IPC\_CREAT);

if (shmid == -1) {

perror("Shared memory");

return 1;

}

shmp = shmat(shmid, NULL, 0);

if (shmp == (void \*) -1) {

perror("Shared memory attach");

return 1;

}

bufptr = shmp->buf;

spaceavailable = BUF\_SIZE;

for (numtimes = 0; numtimes < 3; numtimes++) {

shmp->cnt = fill\_buffer(bufptr, spaceavailable);

shmp->complete = 0;

printf("Writing Process: Shared Memory Write: Wrote %d bytes\n", shmp->cnt);

bufptr = shmp->buf;

spaceavailable = BUF\_SIZE;

sleep(3);

}

printf("Writing Process: Wrote %d times\n", numtimes);

shmp->complete = 1;

if (shmdt(shmp) == -1) {

perror("shmdt");

return 1;

}

if (shmctl(shmid, IPC\_RMID, 0) == -1) {

perror("shmctl");

return 1;

}

printf("Writing Process: Complete\n");

return 0;

}

int fill\_buffer(char \* bufptr, int size) {

static char ch = 'A';

int filled\_count;

//printf("size is %d\n", size);

memset(bufptr, ch, size - 1);

bufptr[size-1] = '\0';

if (ch > 122)

ch = 65;

if ( (ch >= 65) && (ch <= 122) ) {

if ( (ch >= 91) && (ch <= 96) ) {

ch = 65;

}

}

filled\_count = strlen(bufptr);

ch++;

return filled\_count;

}

1. **CODE:**

#include<stdio.h>

#include<sys/ipc.h>

#include<sys/shm.h>

#include<sys/types.h>

#include<string.h>

#include<errno.h>

#include<stdlib.h>

#define BUF\_SIZE 1024

#define SHM\_KEY 0x1234

struct shmseg {

int cnt;

int complete;

char buf[BUF\_SIZE];

};

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

int shmid;

struct shmseg \*shmp;

shmid = shmget(SHM\_KEY, sizeof(struct shmseg), 0644|IPC\_CREAT);

if (shmid == -1) {

perror("Shared memory");

return 1;

}

shmp = shmat(shmid, NULL, 0);

if (shmp == (void \*) -1) {

perror("Shared memory attach");

return 1;

}

while (shmp->complete != 1) {

printf("segment contains : \n\"%s\"\n", shmp->buf);

if (shmp->cnt == -1) {

perror("read");

return 1;

}

printf("Reading Process: Shared Memory: Read %d bytes\n", shmp->cnt);

sleep(3);

}

printf("Reading Process: Reading Done, Detaching Shared Memory\n");

if (shmdt(shmp) == -1) {

perror("shmdt");

return 1;

}

printf("Reading Process: Complete\n");

return 0;

}

**OUTPUT:**

****

****

**Q20. Write a program to provide a solution for reader- writer problem / producer consumer using semaphore.**

**A20.**

**CODE:**

#include<pthread.h>

#include <semaphore.h>

#include <stdio.h>

/\*This program provides a possible solution for first readers writers

problem using mutex and semaphore.

I have used 10 readers and 5 producers to demonstrate the solution. You

can always play with these values.\*/

sem\_t wrt;

pthread\_mutex\_t mutex;

int cnt = 1;

int numreader = 0;

void \*writer(void \*wno)

{

sem\_wait(&wrt);

cnt = cnt\*2;

printf("Writer %d modified cnt to %d\n",(\*((int \*)wno)),cnt);

sem\_post(&wrt);

}

void \*reader(void \*rno)

{

// Reader acquire the lock before modifying numreader

pthread\_mutex\_lock(&mutex);

numreader++;

if(numreader == 1)

{

sem\_wait(&wrt); // If this id the first reader, then it will block the writer

}

pthread\_mutex\_unlock(&mutex);

// Reading Section

printf("Reader %d: read cnt as %d\n",\*((int \*)rno),cnt);

// Reader acquire the lock before modifying numreader

pthread\_mutex\_lock(&mutex);

numreader--;

if(numreader == 0)

{

sem\_post(&wrt); // If this is the last reader, it will wake up the writer.

}

pthread\_mutex\_unlock(&mutex);

}

int main()

{

pthread\_t read[10],write[5];

pthread\_mutex\_init(&mutex, NULL);

sem\_init(&wrt,0,1);

int a[10] = {1,2,3,4,5,6,7,8,9,10}; //Just used for numbering the producer and consumer

for(int i = 0; i < 10; i++)

{

pthread\_create(&read[i], NULL, (void \*)reader, (void \*)&a[i]);

}

for(int i = 0; i < 5; i++)

{

pthread\_create(&write[i], NULL, (void \*)writer, (void \*)&a[i]);

}

for(int i = 0; i < 10; i++)

{

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

}

for(int i = 0; i < 5; i++)

{

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

}

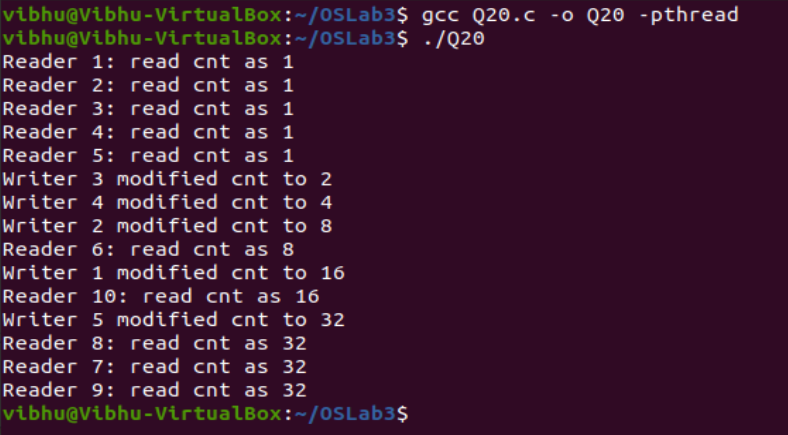
pthread\_mutex\_destroy(&mutex);

sem\_destroy(&wrt);

return 0;

}

**OUTPUT:**



**Q21. Implement a solution for the classical synchronization problem: Dining Philosophers using monitor.**

**A21.**

**CODE:**

#include<iostream>

#define n 5

using namespace std;

int compltedPhilo = 0,i;

struct fork

{

int taken;

}ForkAvil[n];

struct philosp

{

int left;

int right;

}Philostatus[n];

void goForDinner(int philID){

if(Philostatus[philID].left==10 && Philostatus[philID].right==10)

cout<<"Philosopher "<<philID+1<<" completed his dinner\n";

else if(Philostatus[philID].left==1 && Philostatus[philID].right==1){

cout<<"Philosopher "<<philID+1<<" completed his dinner\n";

Philostatus[philID].left = Philostatus[philID].right = 10;

int otherFork = philID-1;if(otherFork== -1)

otherFork=(n-1);

ForkAvil[philID].taken = ForkAvil[otherFork].taken = 0;

cout<<"Philosopher "<<philID+1<<" released fork "<<philID+1<<" and fork"<<otherFork+1<<"\n";

compltedPhilo++;

}

else if(Philostatus[philID].left==1 && Philostatus[philID].right==0){

if(philID==(n-1)){

if(ForkAvil[philID].taken==0){

ForkAvil[philID].taken = Philostatus[philID].right = 1;

cout<<"Fork "<<philID+1<<" taken by philosopher "<<philID+1<<"\n";

}else{

cout<<"Philosopher "<<philID+1<<" is waiting for fork "<<philID+1<<"\n";

}

}else{

int dupphilID = philID;

philID-=1;

if(philID== -1)

philID=(n-1);

if(ForkAvil[philID].taken == 0){

ForkAvil[philID].taken = Philostatus[dupphilID].right = 1;

cout<<"Fork "<<philID+1<<" taken by Philosopher "<<dupphilID+1<<"\n";

}else{

cout<<"Philosopher "<<dupphilID+1<<" is waiting for Fork "<<philID+1<<"\n";

}

}

}

else if(Philostatus[philID].left==0){

if(philID==(n-1)){

if(ForkAvil[philID-1].taken==0){ForkAvil[philID-1].taken =

Philostatus[philID].left = 1;

cout<<"Fork "<<philID<<" taken by philosopher "<<philID+1<<"\n";

}else{

cout<<"Philosopher "<<philID+1<<" is waiting for fork"<<philID<<"\n";

}

}else{

if(ForkAvil[philID].taken == 0){

ForkAvil[philID].taken = Philostatus[philID].left = 1;

cout<<"Fork "<<philID+1<<" taken by Philosopher "<<philID+1<<"\n";

}else{

cout<<"Philosopher "<<philID+1<<" is waiting for Fork"<<philID+1<<"\n";

}

}

}else{}

}

int main(){

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

ForkAvil[i].taken=Philostatus[i].left=Philostatus[i].right=0;

while(compltedPhilo<n){

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

goForDinner(i);

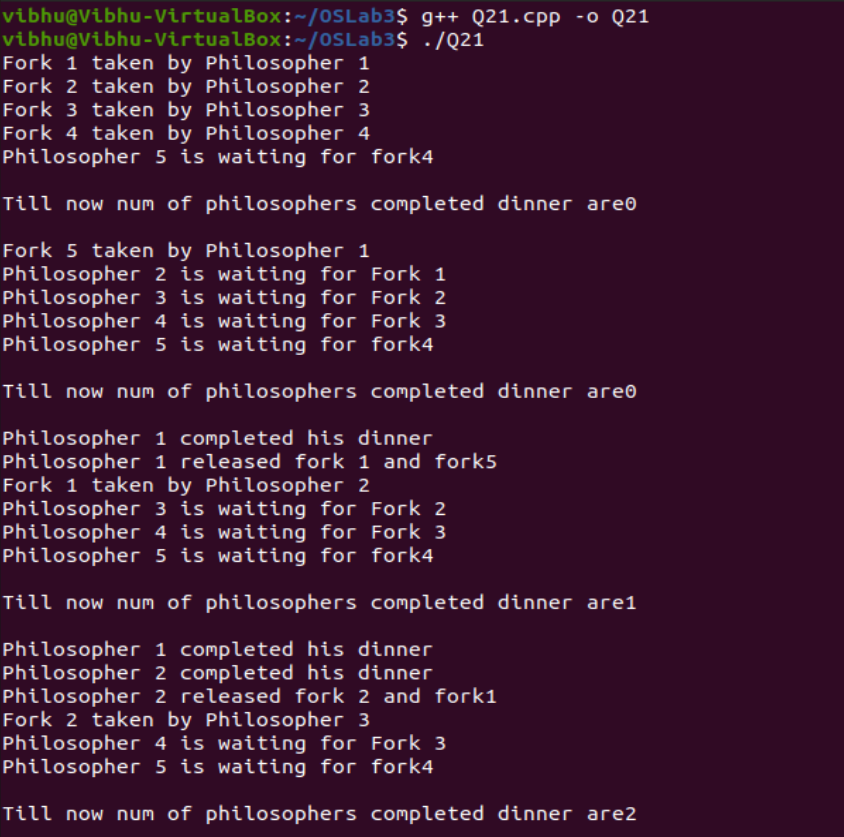
cout<<"\nTill now num of philosophers completed dinner are"<<compltedPhilo<<"\n\n";

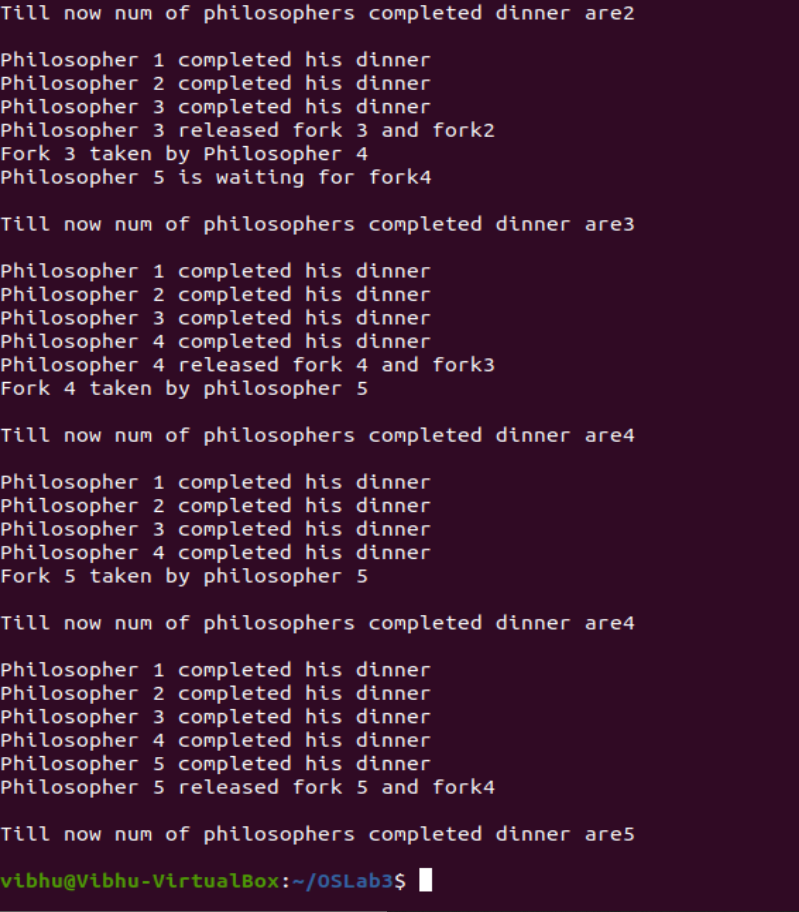
}

return 0;

}

**OUTPUT:**





**Q22. Implement**

1. **Binary Semaphore**

**CODE:**

#include <stdio.h>

#include <pthread.h>

#include <semaphore.h>

int a, b;

sem\_t sem;

void ScanNumbers(void \*ptr){

for (;;){

printf("%s", (char \*)ptr);

scanf("%d %d", &a, &b);

sem\_post(&sem);

usleep(100 \* 1000);

}

}

void SumAndPrint(void \*ptr){

for (;;){

sem\_wait(&sem);

printf("%s %d\n", (char \*)ptr, a + b);

}

}

int main()

{

pthread\_t thread1;

pthread\_t thread2;

char \*Msg1 = "Enter Two Numbers\n";

char \*Msg2 = "sum = ";

sem\_init(&sem, 0, 0);

pthread\_create(&thread1, NULL, (void \*)ScanNumbers, (void \*)Msg1);

pthread\_create(&thread2, NULL, (void \*)SumAndPrint, (void \*)Msg2);

pthread\_join(thread1, NULL);

pthread\_join(thread2, NULL);

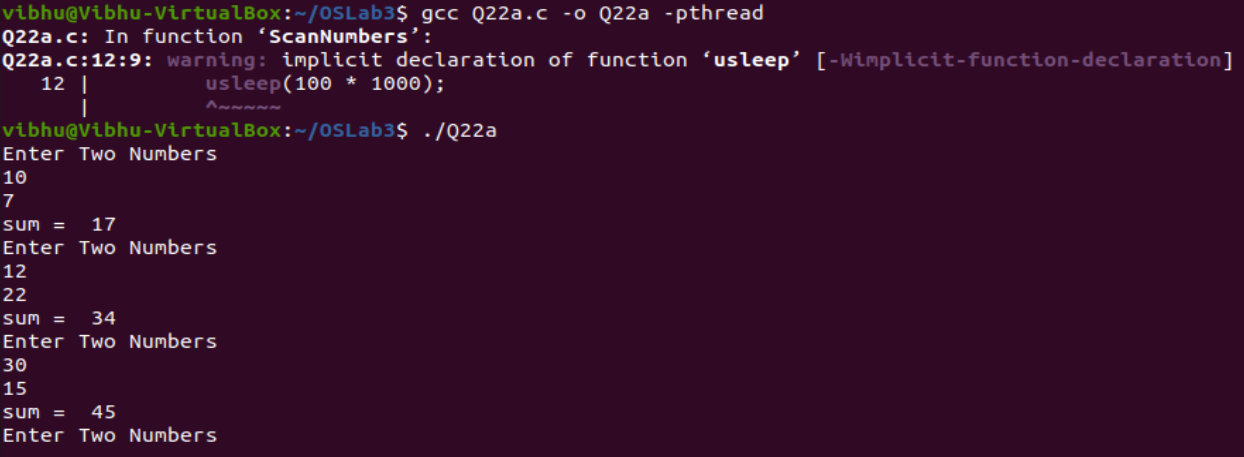
printf("Wait For Both Thread Finished\n");

sem\_destroy(&sem);

return 0;

}

**OUTPUT:**

****

1. **Counting Semaphore.**

**CODE:**

#include<stdio.h>

#include<stdlib.h>

int mutex=1,full=0,empty=3,x=0;

int main()

{

int n;

void producer();

void consumer();

int wait(int);

int signal(int);

printf("\n1.Producer\n2.Consumer\n3.Exit");

while(1)

{

printf("\nEnter your choice:");

scanf("%d",&n);

switch(n)

{

case 1: if((mutex==1)&&(empty!=0))

producer();

else

printf("Buffer is full!!");

break;

case 2: if((mutex==1)&&(full!=0))

consumer();

else

printf("Buffer is empty!!");

break;

case 3:

exit(0);

break;

}

}

return 0;

}

int wait(int s)

{

return (--s);

}

int signal(int s)

{

return(++s);

}

void producer()

{

mutex=wait(mutex);

full=signal(full);

empty=wait(empty);

x++;

printf("\nProducer produces the item %d",x);

mutex=signal(mutex);

}

void consumer()

{

mutex=wait(mutex);

full=wait(full);

empty=signal(empty);

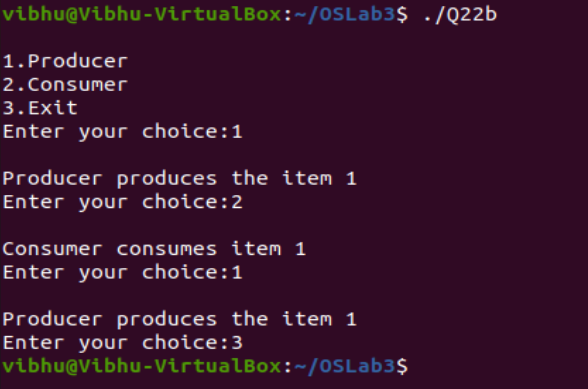
printf("\nConsumer consumes item %d",x);

x--;

mutex=signal(mutex);

}

**OUTPUT:**

****

**Q23. In the Cigarette-Smokers Problem, Consider a system with three smoker processes and one agent process. Each smoker continuously rolls a cigarette and then smokes it. But to roll and smoke a cigarette, the smoker needs three ingredients: tobacco, paper and matches. One of the smoker processes has paper, another has tobacco and the third has matches. The agent has an infinite supply of all three materials. The agent places two of the ingredients on the table. The smoker who has the remaining ingredient then makes and smokes a cigarette, signaling the agent on completion. The agent then puts out another two of the three ingredients and the cycle repeats. Write a program to synchronize the agent and the smokers.**

**A23.**

**CODE:**

#include <pthread.h>

#include <semaphore.h>

#include <stdbool.h>

#include <stdio.h>

#include <stdlib.h>

#include <errno.h>

#include <unistd.h>

sem\_t agent\_ready;

sem\_t smoker\_semaphors[3];

char\* smoker\_types[3] = { "matches & tobacco", "matches & paper", "tobacco & paper" }

bool items\_on\_table[3] = { false, false, false };

sem\_t pusher\_semaphores[3];

void\* smoker(void\* arg)

{

int smoker\_id = (int) arg;

int type\_id = smoker\_id % 3;

for (int i = 0; i < 3; ++i)

{

printf("\033[0;37mSmoker %d \033[0;31m>>\033[0m Waiting for %s\n",

smoker\_id, smoker\_types[type\_id]);

sem\_wait(&smoker\_semaphors[type\_id]);

printf("\033[0;37mSmoker %d \033[0;32m<<\033[0m Now making the a cigarette\n", smoker\_id);

usleep(rand() % 50000);

sem\_post(&agent\_ready);

printf("\033[0;37mSmoker %d \033[0;37m--\033[0m Now smoking\n", smoker\_id);

usleep(rand() % 50000);

}

return NULL;

}

sem\_t pusher\_lock;

void\* pusher(void\* arg)

{

int pusher\_id = (int) arg;

for (int i = 0; i < 12; ++i)

{

sem\_wait(&pusher\_semaphores[pusher\_id]);

sem\_wait(&pusher\_lock);

if (items\_on\_table[(pusher\_id + 1) % 3])

{

items\_on\_table[(pusher\_id + 1) % 3] = false;

sem\_post(&smoker\_semaphors[(pusher\_id + 2) % 3]);

}

else if (items\_on\_table[(pusher\_id + 2) % 3])

{

items\_on\_table[(pusher\_id + 2) % 3] = false;

sem\_post(&smoker\_semaphors[(pusher\_id + 1) % 3]);

}

else

{

items\_on\_table[pusher\_id] = true;

}

sem\_post(&pusher\_lock);

}

return NULL;

}

void\* agent(void\* arg)

{

int agent\_id = (int) arg;

for (int i = 0; i < 6; ++i)

{

usleep(rand() % 200000);

sem\_wait(&agent\_ready);

sem\_post(&pusher\_semaphores[agent\_id]);

sem\_post(&pusher\_semaphores[(agent\_id + 1) % 3]);

printf("\033[0;35m==> \033[0;33mAgent %d giving out %s\033[0;0m\n",

agent\_id, smoker\_types[(agent\_id + 2) % 3]);

}

return NULL;

}

int main(int argc, char\* arvg[])

{

srand(time(NULL));

sem\_init(&agent\_ready, 0, 1);

sem\_init(&pusher\_lock, 0, 1);

for (int i = 0; i < 3; ++i)

{

sem\_init(&smoker\_semaphors[i], 0, 0);

sem\_init(&pusher\_semaphores[i], 0, 0);

}

int smoker\_ids[6];

pthread\_t smoker\_threads[6];

for (int i = 0; i < 6; ++i)

{

smoker\_ids[i] = i;

if (pthread\_create(&smoker\_threads[i], NULL, smoker, &smoker\_ids[i]) == EAGAIN)

{

perror("Insufficient resources to create thread");

return 0;

}

}

int pusher\_ids[6];

pthread\_t pusher\_threads[6];

for (int i = 0; i < 3; ++i)

{

pusher\_ids[i] = i;

if (pthread\_create(&pusher\_threads[i], NULL, pusher, &pusher\_ids[i]) == EAGAIN)

{

perror("Insufficient resources to create thread");

return 0;

}

}

int agent\_ids[6];

pthread\_t agent\_threads[6];

for (int i = 0; i < 3; ++i)

{

agent\_ids[i] =i;

if (pthread\_create(&agent\_threads[i], NULL, agent, &agent\_ids[i]) == EAGAIN)

{

perror("Insufficient resources to create thread");

return 0;

}

}

for (int i = 0; i < 6; ++i)

{

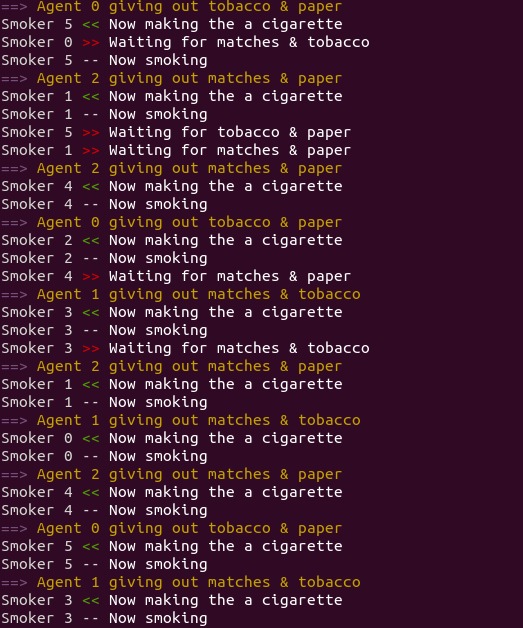
pthread\_join(smoker\_threads[i], NULL);

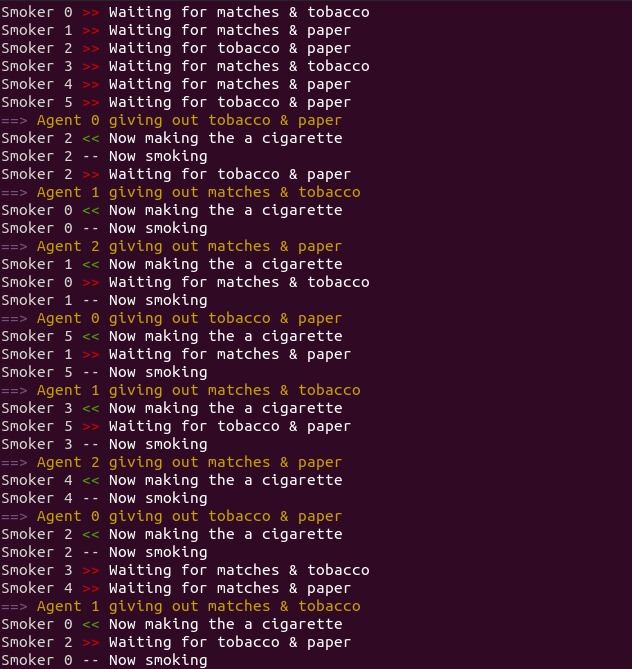
}

return 0;

}

**OUTPUT:**

****

****

**Q24. Write a program to avoid deadlock using Banker’s algorithm. (Safety algorithm).**

**A24.**

**CODE:**

#include<iostream>

using namespace std;

// Number of processes

const int P = 5;

// Number of resources

const int R = 3;

// Function to find the need of each process

void calculateNeed(int need[P][R], int maxm[P][R],

int allot[P][R])

{

// Calculating Need of each P

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

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

// Need of instance = maxm instance -

// allocated instance

need[i][j] = maxm[i][j] - allot[i][j];

}

// Function to find the system is in safe state or not

bool isSafe(int processes[], int avail[], int maxm[][R],

int allot[][R])

{

int need[P][R];

// Function to calculate need matrix

calculateNeed(need, maxm, allot);

// Mark all processes as infinish

bool finish[P] = {0};

// To store safe sequence

int safeSeq[P];

// Make a copy of available resources

int work[R];

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

work[i] = avail[i];

// While all processes are not finished

// or system is not in safe state.

int count = 0;

while (count < P)

{

// Find a process which is not finish and

// whose needs can be satisfied with current

// work[] resources.

bool found = false;

for (int p = 0; p < P; p++)

{

// First check if a process is finished,

// if no, go for next condition

if (finish[p] == 0)

{

// Check if for all resources of

// current P need is less

// than work

int j;

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

if (need[p][j] > work[j])

break;

// If all needs of p were satisfied.

if (j == R)

{

// Add the allocated resources of

// current P to the available/work

// resources i.e.free the resources

for (int k = 0 ; k < R ; k++)

work[k] += allot[p][k];

// Add this process to safe sequence.

safeSeq[count++] = p;

// Mark this p as finished

finish[p] = 1;

found = true;

}

}

}

// If we could not find a next process in safe

// sequence.

if (found == false)

{

cout << "System is not in safe state";

return false;

}

}

// If system is in safe state then

// safe sequence will be as below

cout << "System is in safe state.\nSafe"

" sequence is: ";

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

cout << safeSeq[i] << " ";

return true;

}

// Driver code

int main()

{

int processes[] = {0, 1, 2, 3, 4};

// Available instances of resources

int avail[] = {3, 3, 2};

// Maximum R that can be allocated

// to processes

int maxm[][R] = {{7, 5, 3},

{3, 2, 2},

{9, 0, 2},

{2, 2, 2},

{4, 3, 3}};

// Resources allocated to processes

int allot[][R] = {{0, 1, 0},

{2, 0, 0},

{3, 0, 2},

{2, 1, 1},

{0, 0, 2}};

// Check system is in safe state or not

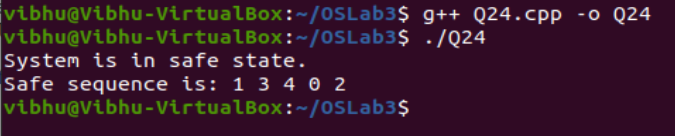
isSafe(processes, avail, maxm, allot);

cout<<endl;

return 0;

}

**OUTPUT:**



**Q25. Simulate with a program to provide deadlock avoidance of Banker’s Algorithm including Safe state and additional resource request.**

**A25.**

**CODE:**

#include <stdio.h>

int main()

{

int count = 0, m, n, process, temp, resource;

int allocation\_table[5] = {0, 0, 0, 0, 0};

int available[5], current[5][5], maximum\_claim[5][5];

int maximum\_resources[5], running[5], safe\_state = 0;

printf("\nEnter The Total Number Of Processes:\t");

scanf("%d", &process);

for(m = 0; m < process; m++)

{

running[m] = 1;

count++;

}

printf("\nEnter The Total Number Of Resources To Allocate:\t");

scanf("%d", &resource);

printf("\nEnter The Claim Vector:\t");

for(m = 0; m < resource; m++)

{

scanf("%d", &maximum\_resources[m]);

}

printf("\nEnter Allocated Resource Table:\n");

for(m = 0; m < process; m++)

{

for(n = 0; n < resource; n++)

{

scanf("%d", &current[m][n]);

}

}

printf("\nEnter The Maximum Claim Table:\n");

for(m = 0; m < process; m++)

{

for(n = 0; n < resource; n++)

{

scanf("%d", &maximum\_claim[m][n]);

}

}

printf("\nThe Claim Vector \n");

for(m = 0; m < resource; m++)

{

printf("\t%d ", maximum\_resources[m]);

}

printf("\n The Allocated Resource Table\n");

for(m = 0; m < process; m++)

{

for(n = 0; n < resource; n++)

{

printf("\t%d", current[m][n]);

}

printf("\n");

}

printf("\nThe Maximum Claim Table \n");

for(m = 0; m < process; m++)

{

for(n = 0; n < resource; n++)

{

printf("\t%d", maximum\_claim[m][n]);

}

printf("\n");

}

for(m = 0; m < process; m++)

{

for(n = 0; n < resource; n++)

{

allocation\_table[n] = allocation\_table[n] + current[m][n];

}

}

printf("\nAllocated Resources \n");

for(m = 0; m < resource; m++)

{

printf("\t%d", allocation\_table[m]);

}

for(m = 0; m < resource; m++)

{

available[m] = maximum\_resources[m] - allocation\_table[m];

}

printf("\nAvailable Resources:");

for(m = 0; m < resource; m++)

{

printf("\t%d", available[m]);

}

printf("\n");

while(count != 0)

{

safe\_state = 0;

for(m = 0; m < process; m++)

{

if(running[m])

{

temp = 1;

for(n = 0; n < resource; n++)

{

if(maximum\_claim[m][n] - current[m][n] > available[n])

{

temp = 0;

break;

}

}

if(temp)

{

printf("\nProcess %d Is In Execution \n", m + 1);

running[m] = 0;

count--;

safe\_state = 1;

for(n = 0; n < resource; n++)

{

available[n] = available[n] + current[m][n];

}

break;

}

}

}

if(!safe\_state)

{

printf("\nThe Processes Are In An Unsafe State \n");

break;

}

else

{

printf("\nThe Process Is In A Safe State \n");

printf("\nAvailable Vector\n");

for(m = 0; m < resource; m++)

{

printf("\t%d", available[m]);

}

printf("\n");

}

}

return 0;

}

**OUTPUT:**

