Name: Ashutosh Soni

Id: 2018ucp1505

**CST 303 Concurrent and Parallel Programming Lab**

**ASSIGNMENT-1**

Q1: Implement all the four attempts of Dekker’s algorithm to solve critical section problem in C++.

Ans:

**First Attempt**

// First attempt

// Pros

// Mutual Exclusion holds

// Free from deadlock

//cons

// If one process dies other gets blocked.

#include <bits/stdc++.h>

#include <pthread.h>

using namespace std;

int turn = 1;

int x=0;

// Critical Section

void\* critical\_section(){

x++;

}

// Process P

void\* p(){

while(1) {

if(x>=50){

return NULL;

}

cout<<"In process p"<<endl;

while(turn!=1){

}

critical\_section();

turn=2;

}

}

// Process Q

void\* q(){

while(1){

if(x>=50){

return NULL;

}

cout<<"In process q"<<endl;

while(turn!=2){

}

critical\_section();

turn=1;

}

}

// Loop forever type section

void\* start\_p(void\* arg){

p();

}

void\* start\_q(void\* arg){

q();

}

int main(){

// creation of two thread

pthread\_t pid,qid;

pthread\_create(&pid,NULL,&start\_p,NULL);

pthread\_create(&qid,NULL,&start\_q, NULL);

// joining of thread

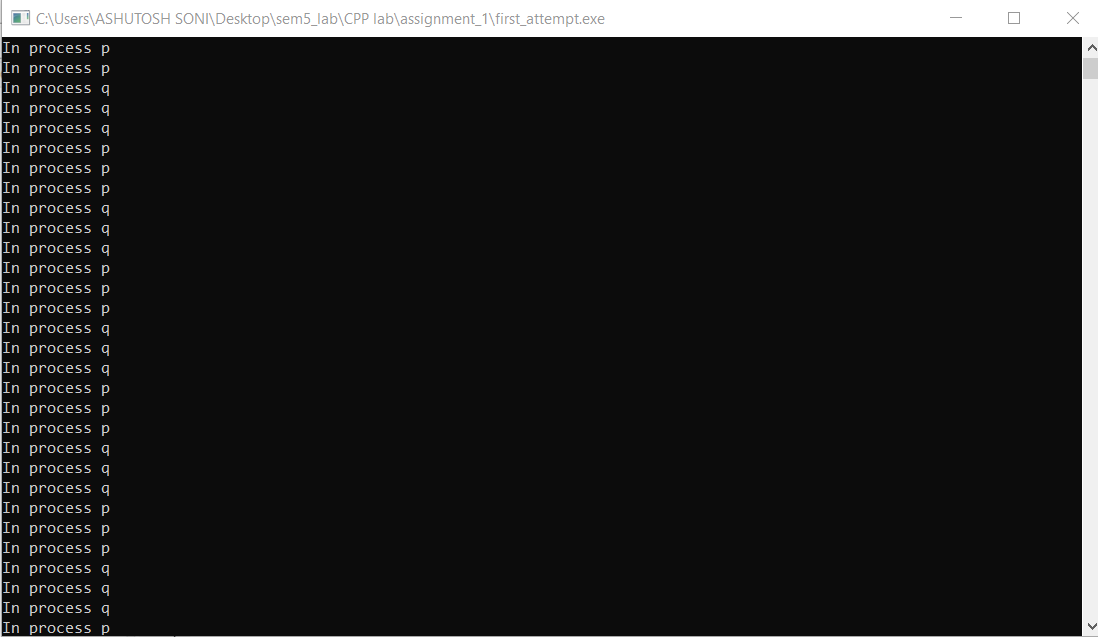
pthread\_join(pid,NULL);

pthread\_join(qid,NULL);

// Exit

pthread\_exit(NULL);

pthread\_exit(NULL);

}

**Output of the**

**Program:**

**Second Attempt**

// Second Attempt

// pros

// No deadlock .

// Free from starvation.

// cons

// Mutual Exclusion principle not holds.

#include<bits/stdc++.h>

#include<pthread.h>

using namespace std;

int x=0;

bool wantp=false,wantq=false;

// Critical Section

void critical\_section(){

x++;

}

// Process p

void\* p(){

while(1){

if(x>=50){

return NULL;

}

while(wantq==true){

}

wantp=true;

cout<<"Critical Section of P starts"<<endl;

critical\_section();

cout<<"Critical Section of P ends"<<endl;

wantp=false;

}

}

// Process q

void\* q(){

while(1){

if(x>=50){

return NULL;

}

while(wantp==true){

}

wantq=true;

cout<<"Critical Section of Q starts"<<endl;

critical\_section();

cout<<"Critical Section of Q ends"<<endl;

wantq=false;

}

}

// Loop forever type section

void\* start\_p(void\* arg){

p();

}

void\* start\_q(void\* arg){

q();

}

int main(){

pthread\_t pid,qid;

// creating two thread

pthread\_create(&pid,NULL,&start\_p,NULL);

pthread\_create(&qid,NULL,&start\_q,NULL);

// joining of thread

pthread\_join(pid,NULL);

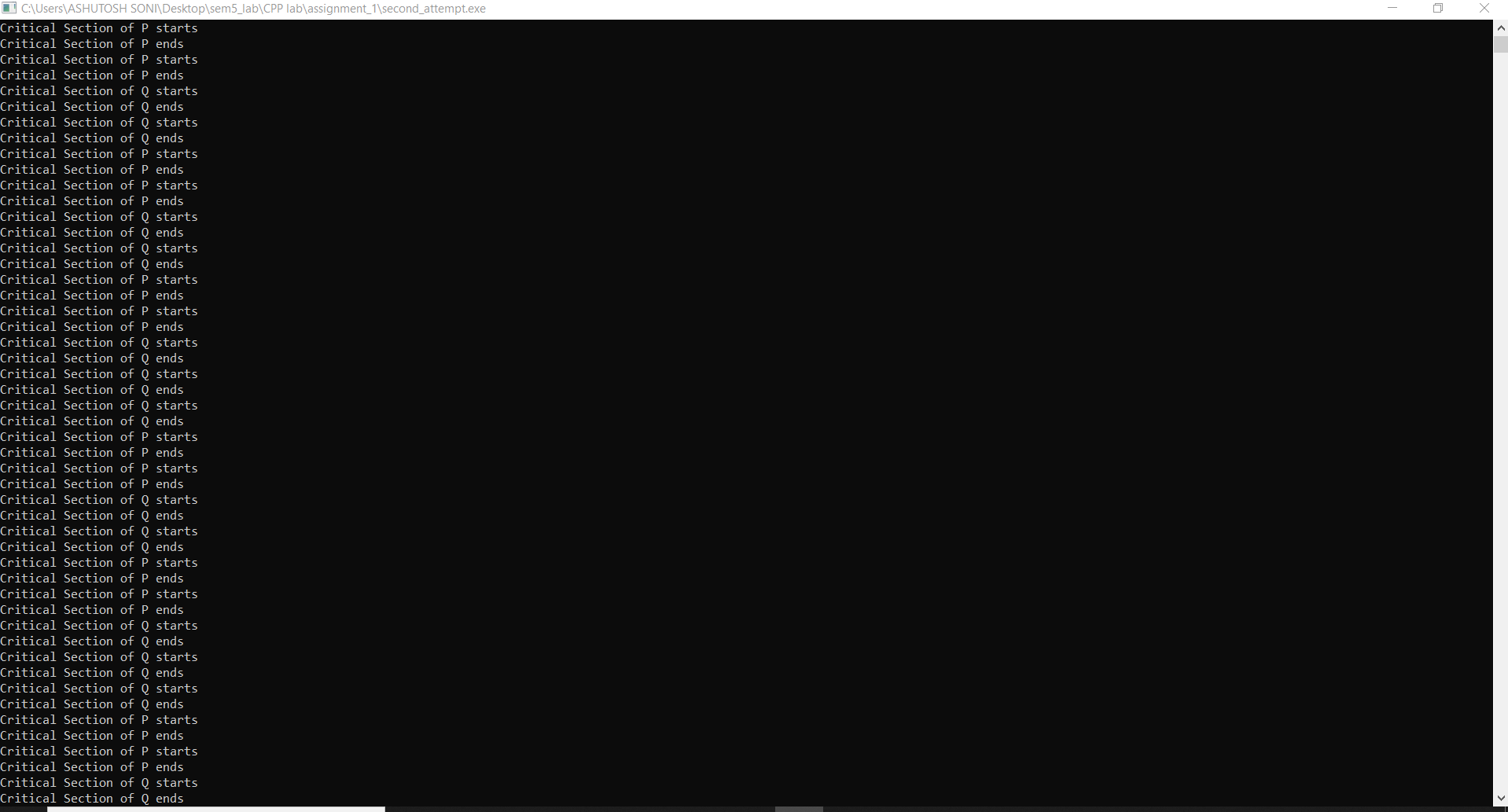
pthread\_join(qid,NULL);

// Exit

pthread\_exit(NULL);

return 0;

}

****

**Output of the**

**Program:**

**Third Attempt**

// Third attempt

// mutual Exclusion satisfied.

// Not free from deadlock.

#include<bits/stdc++.h>

#include<pthread.h>

using namespace std;

bool wantp=false,wantq=false;

int x=0;

// Critical Section

void critical\_section(){

x++;

}

// Process p

void\* p(){

while(1){

if(x>=50){

return NULL;

}

wantp=true;

while(wantq==true){

}

cout<<"Critical Section of P starts here"<<endl;

critical\_section();

cout<<"Critical Section of P ends here"<<endl;

wantp=false;

}

}

// Process Q

void\* q(){

while(1){

if(x>=50){

return NULL;

}

wantq=true;

while(wantp==true){

}

cout<<"Critical Section of Q starts here"<<endl;

critical\_section();

cout<<"Critical Section of Q ends here"<<endl;

wantq=false;

}

}

// starting of the process

void\* start\_p(void\* arg){

p();

}

void\* start\_q(void\* arg){

q();

}

int main(){

pthread\_t pid,qid;

// Creating two thread

pthread\_create(&pid,NULL,\*start\_p,NULL);

pthread\_create(&qid,NULL,\*start\_q,NULL);

// Joining thread

pthread\_join(pid,NULL);

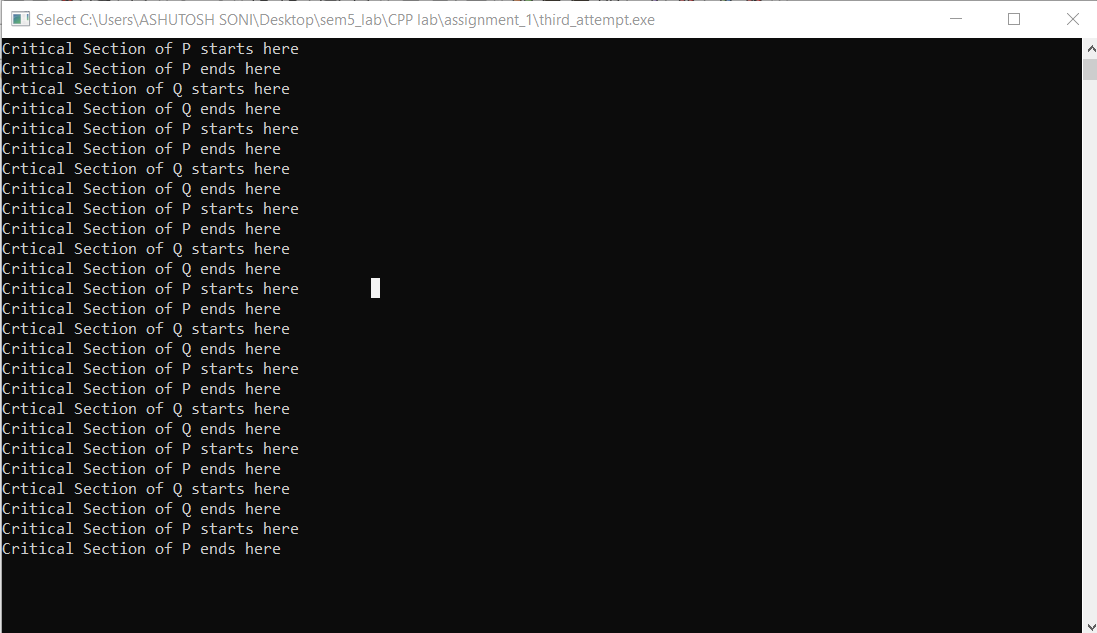
pthread\_join(qid,NULL);

// Exit

pthread\_exit(NULL);

return 0;

}



**Output of the**

**Program:**

**Fourth Attempt**

// Fourth Attempt

// Mutual Exclusion satisfied

// Free from deadlock

// Starvation may happens.

#include<bits/stdc++.h>

#include<pthread.h>

using namespace std;

bool wantp=false,wantq=false;

int x=0;

// Critical Section

void critical\_section(){

x++;

}

// process P

void\* p(){

while(1){

if(x>=10){

return NULL;

}

wantp=true;

while(wantq){

wantp=false;

wantp=true;

}

cout<<"Critical Section of P starts here"<<endl;

critical\_section();

cout<<"Critical Section of P ends here"<<endl;

wantp=false;

}

}

// Process Q

void\* q(){

while(1){

if(x>=10){

return NULL;

}

wantq=true;

while(wantp){

wantq=false;

wantp=true;

}

cout<<"Critical Section of Q starts here"<<endl;

critical\_section();

cout<<"Critical Section of Q ends here"<<endl;

wantq=false;

}

}

// start for join process P

void\* start\_p(void\* arg){

p();

}

// start for join process Q

void\* start\_q(void\* arg){

q();

}

int main(){

pthread\_t pid,qid;

// creating two threads

pthread\_create(&pid,NULL,&start\_p,NULL);

pthread\_create(&qid,NULL,&start\_q,NULL);

// Joining threads

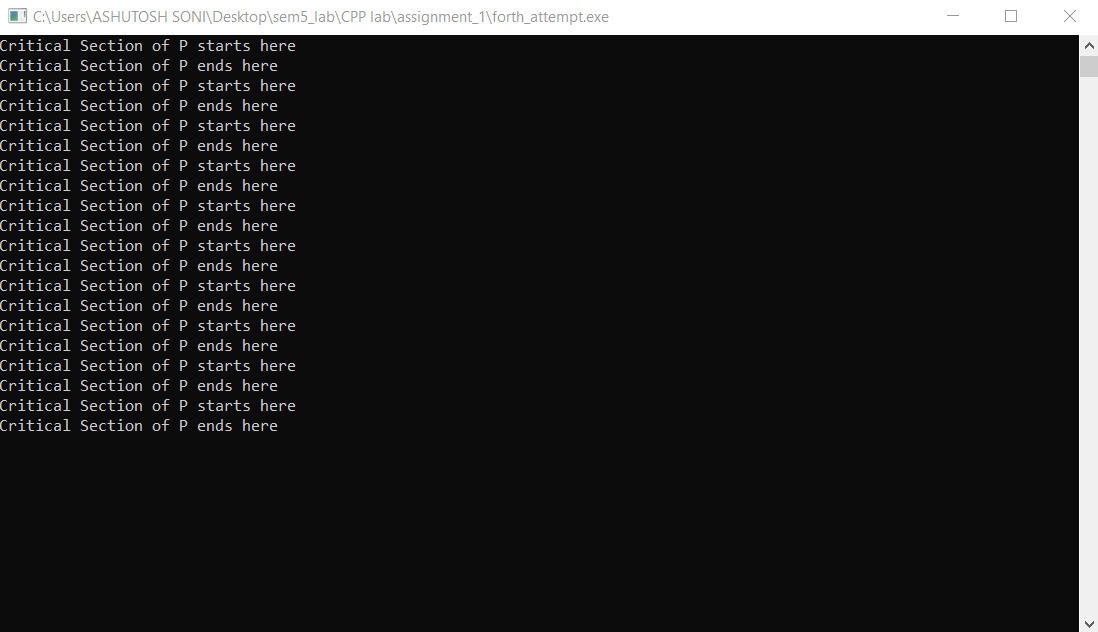
pthread\_join(pid,NULL);

pthread\_join(qid,NULL);

// Exit

pthread\_exit(NULL);

return 0;

}

**Output of the**

**Program:**

Q2: Implement Dekker’s Algorithm for mutual exclusion in C++.

Ans:

**Dekkers Algorithm**

// Dekkers algorithm implementation

// Free from satrvation

// Mutual Exlusion satisfied

// Free from deadlock

#include<bits/stdc++.h>

#include<pthread.h>

using namespace std;

int turn=1;

bool wantp=false,wantq=false;

int x=0;

// Critical Section

void critical\_section(){

x++;

}

// process P

void\* p(){

while(1){

if(x>=50){

return NULL;

}

wantp=true;

while(wantq){

if(turn==2){

wantp=false;

while(turn!=1){

}

wantp=true;

}

}

cout<<"Critical Section of P starts here"<<endl;

critical\_section();

cout<<"Critical Section of P ends here"<<endl;

turn=2;

wantp=false;

}

}

// process Q

void\* q(){

while(1){

if(x>=50){

return NULL;

}

wantq=true;

while(wantp){

if(turn==1){

wantq=false;

while(turn!=2){

}

wantq=true;

}

}

cout<<"Critical Section of Q starts here"<<endl;

critical\_section();

cout<<"Critical Section of Q ends here"<<endl;

turn=1;

wantq=false;

}

}

// start P

void\* start\_p(void\* arg){

p();

}

// start q

void\* start\_q(void\* arg){

q();

}

int main(){

pthread\_t pid,qid;

// creating two thread

pthread\_create(&pid,NULL,&start\_p,NULL);

pthread\_create(&qid,NULL,\*start\_q,NULL);

// Joining threads

pthread\_join(pid,NULL);

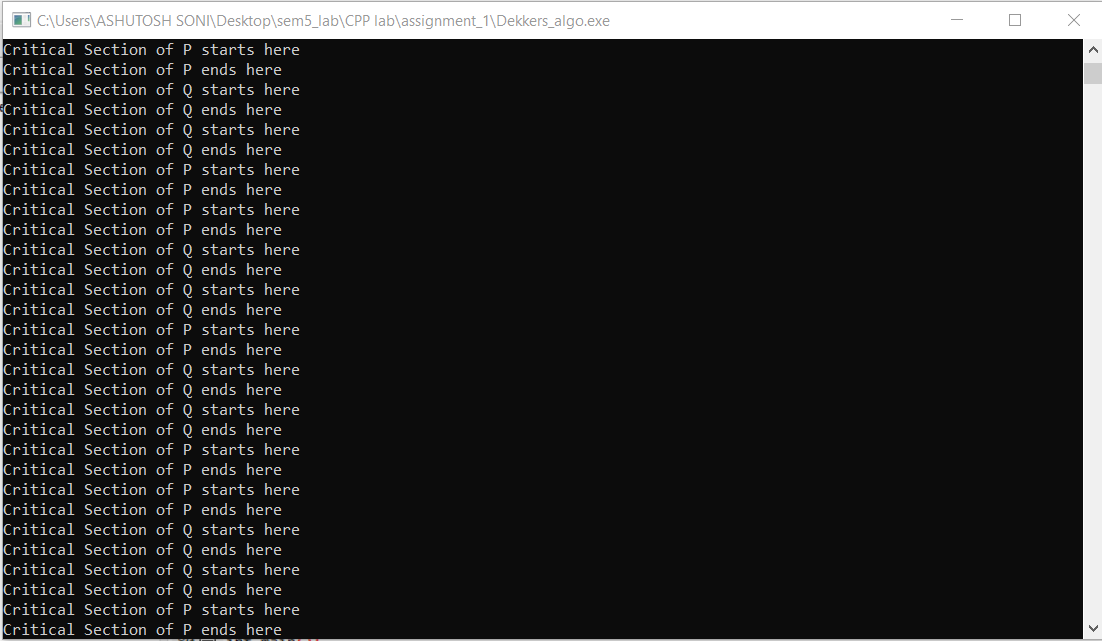
pthread\_join(qid,NULL);

// Exit

pthread\_exit(NULL);

return 0;

}



**Output of the**

**Program:**

Name: Ashutosh Soni

Id: 2018ucp1505

**ASSIGNMENT-2**

Q1: Implement solution of Critical Section problem with Semaphores (two processes).

Ans:

**Critical Section problem with Semaphores (two processes)**

#include<bits/stdc++.h>

#include<pthread.h>

#include<semaphore.h>

using namespace std;

// Declaration

pthread\_t p1,p2;

sem\_t semaphore;

int a=0,x=0;

// Critical Section

void critical\_section(){

// Here -1 because lower thread is 2......so to show readability

cout<<"Critical section of "<<pthread\_self()-1<<" thread"<<endl;

x++;

}

// Process p

void\* p1\_start(void \*arg){

while(x<30){

// Non critical section

a=(a+1)%2;

sem\_wait(&semaphore);

critical\_section();

sem\_post(&semaphore);

}

}

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

// Declaration ......

pthread\_attr\_t attr;

// Initialization of semaphore

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

// pthread\_attr\_t initialization

pthread\_attr\_init(&attr);

pthread\_attr\_setdetachstate(&attr,PTHREAD\_CREATE\_JOINABLE);

// creation of process

int r1=pthread\_create(&p1,&attr,p1\_start,NULL);

if(r1){

cout<<"Error in creating thread"<<endl;

exit(-1);

}

r1=pthread\_create(&p2,&attr,p1\_start,NULL);

if(r1){

cout<<"Error in creating thread"<<endl;

exit(-1);

}

// destroying the pthread\_attr

pthread\_attr\_destroy(&attr);

// Joining the process

r1=pthread\_join(p1,NULL);

if(r1){

cout<<"Error in joining thread"<<endl;

exit(-1);

}

r1=pthread\_join(p2,NULL);

if(r1){

cout<<"Error in joining thread"<<endl;

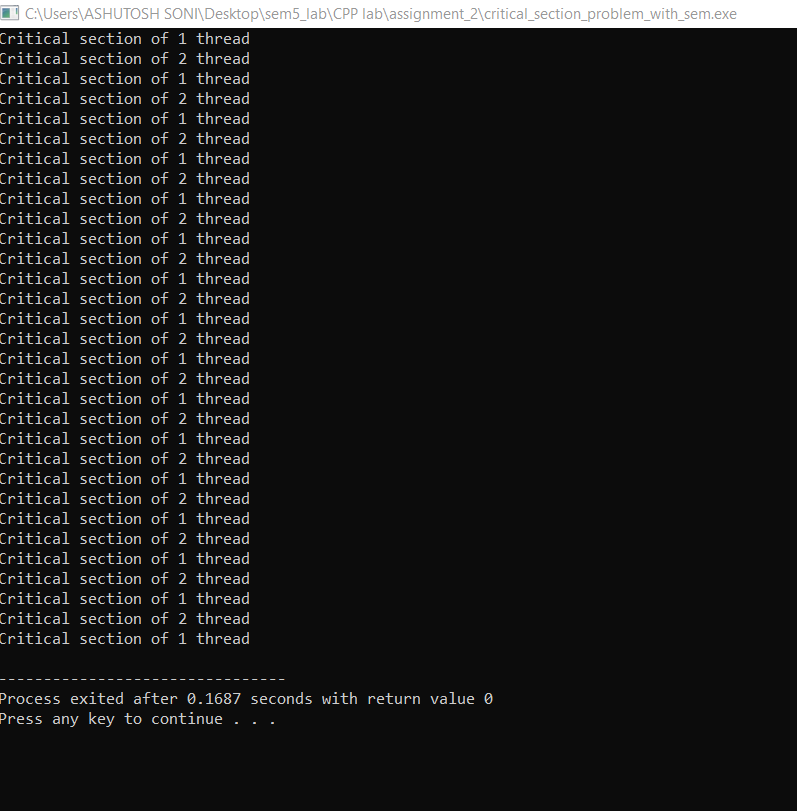
exit(-1);

}

// Exiting pthread

pthread\_exit(NULL);

}



**Output of the**

**Program:**

Q2: Implement solution of Critical Section problem with Semaphores (N processes).

Ans:

**Critical Section problem with Semaphores (N processes)**

#include<bits/stdc++.h>

#include<pthread.h>

#include<semaphore.h>

using namespace std;

// Declaration

sem\_t semaphore;

int a=0,x=0;

// Critical Section

void critical\_section(){

// Here -1 because lower thread is 2......so to show readability

cout<<"Critical section of "<<pthread\_self()-1<<" thread"<<endl;

x++;

}

// Process p

void\* p1\_start(void \*arg){

while(x<30){

// Non critical section

a=(a+1)%2;

sem\_wait(&semaphore);

critical\_section();

sem\_post(&semaphore);

}

}

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

//declaration

int r1,N;

// Taking input of number of process

cout<<"Enter the number you want to Enter"<<endl;

cin>>N;

// Declaration of thread

pthread\_t process[N];

// Declaration of attribute......

pthread\_attr\_t attr;

// Initialization of semaphore

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

// pthread\_attr\_t initialization

pthread\_attr\_init(&attr);

pthread\_attr\_setdetachstate(&attr,PTHREAD\_CREATE\_JOINABLE);

// creation of process

for(int i=0;i<N;i++){

r1=pthread\_create(&process[i],&attr,p1\_start,NULL);

if(r1){

cout<<"Error in creating thread"<<endl;

exit(-1);

}

}

// destroying the pthread\_attr

pthread\_attr\_destroy(&attr);

// Joining the process

for(int i=0;i<N;i++){

r1=pthread\_join(process[i],NULL);

if(r1){

cout<<"Error in joining thread"<<endl;

exit(-1);

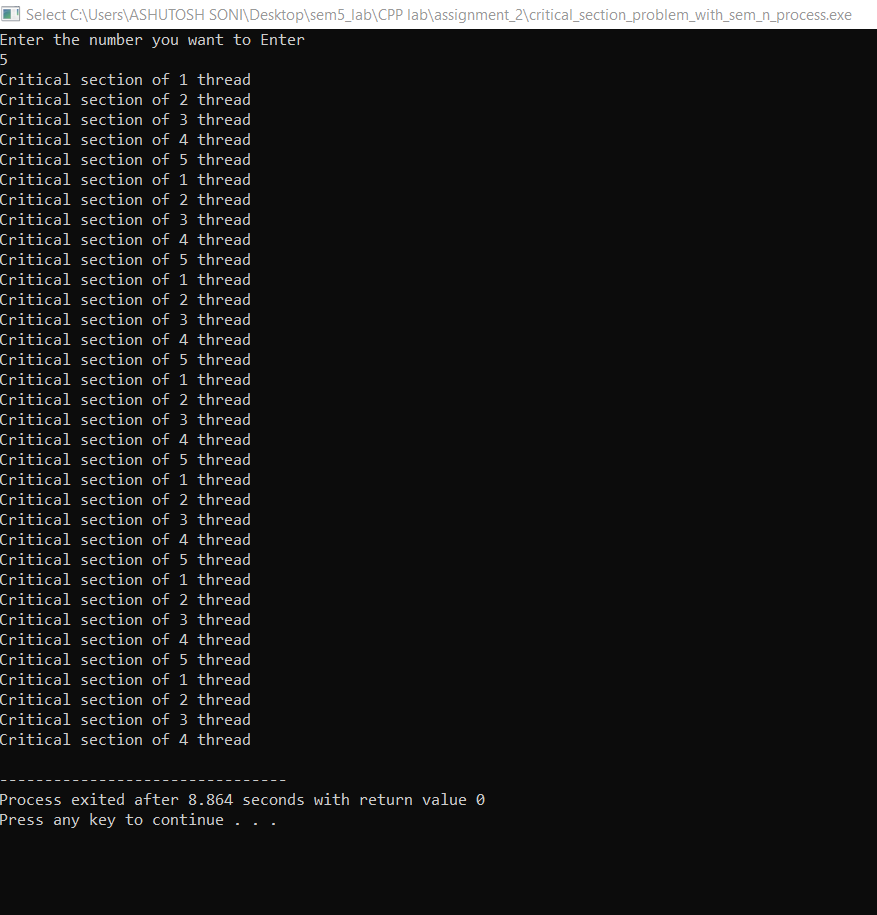
}

}

// Exiting pthread

pthread\_exit(NULL);

}



**Output of the**

**Program:**

Q3: Implement producer-consumer problem with Semaphores (infinite buffer).

Ans:

**Producer-consumer problem with Semaphores (infinite buffer)**

#include<bits/stdc++.h>

#include<pthread.h>

#include<semaphore.h>

#include <unistd.h>

using namespace std;

// Declaration

int r1,total\_produced=0,total\_consume=0;

// Semaphore declaration

sem\_t notEmpty;

// Producer Section

void\* produce(void \*arg){

while(1){

cout<<"Producer produces item."<<endl;

cout<<"Total produced = "<<++total\_produced<<" Total consume = "<<total\_consume\*-1<<endl;

sem\_post(&notEmpty);

sleep(rand()%100\*0.01);

}

}

// Consumer Section

void\* consume(void \*arg){

while(1){

sem\_wait(&notEmpty);

cout<<"Consumer consumes item."<<endl;

cout<<"Total produced = "<<total\_produced<<" Total consume = "<<(--total\_consume)\*-1<<endl;

sleep(rand()%100\*0.01);

}

}

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

// thread declaration

pthread\_t producer,consumer;

// Declaration of attribute......

pthread\_attr\_t attr;

// semaphore initialization

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

// pthread\_attr\_t initialization

pthread\_attr\_init(&attr);

pthread\_attr\_setdetachstate(&attr,PTHREAD\_CREATE\_JOINABLE);

// Creation of process

r1=pthread\_create(&producer,&attr,produce,NULL);

if(r1){

cout<<"Error in creating thread"<<endl;

exit(-1);

}

r1=pthread\_create(&consumer,&attr,consume,NULL);

if(r1){

cout<<"Error in creating thread"<<endl;

exit(-1);

}

// destroying the pthread\_attr

pthread\_attr\_destroy(&attr);

// Joining the thread

r1=pthread\_join(producer,NULL);

if(r1){

cout<<"Error in joining thread"<<endl;

exit(-1);

}

r1=pthread\_join(consumer,NULL);

if(r1){

cout<<"Error in joining thread"<<endl;

exit(-1);

}

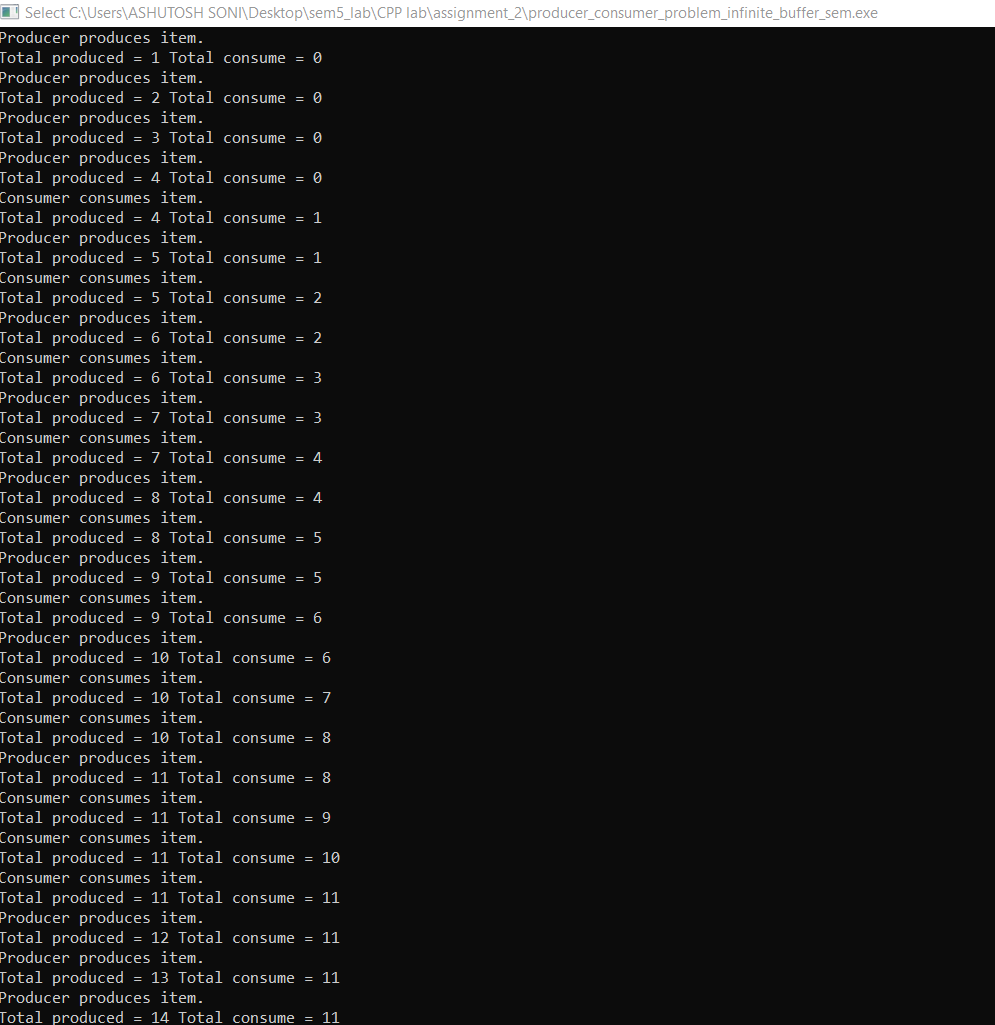
// Exiting thread

pthread\_exit(NULL);

return 0;

}

**Output of the Program:**



Q4: Implement producer-consumer problem with Semaphores (finite buffer).

Ans:

**Producer-consumer problem with Semaphores (finite buffer)**

#include<bits/stdc++.h>

#include<pthread.h>

#include<semaphore.h>

#include <unistd.h>

using namespace std;

// Declaration

int r1,items=0;

// Semaphore declaration

sem\_t notEmpty,notFull;

// Producer Section

void\* produce(void \*arg){

while(1){

sem\_wait(&notFull);

sleep(rand()%100\*0.01);

cout<<"Producer produces item.Items Present = "<<++items<<endl;

sem\_post(&notEmpty);

sleep(rand()%100\*0.01);

}

}

// Consumer Section

void\* consume(void \*arg){

while(1){

sem\_wait(&notEmpty);

sleep(rand()%100\*0.01);

cout<<"Consumer consumes item.Items Present = "<<--items<<endl;

sem\_post(&notFull);

sleep(rand()%100\*0.01);

}

}

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

int N;

cout<<"Enter the capacity of the buffer"<<endl;

cin>>N;

// thread declaration

pthread\_t producer,consumer;

// Declaration of attribute......

pthread\_attr\_t attr;

// semaphore initialization

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

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

// pthread\_attr\_t initialization

pthread\_attr\_init(&attr);

pthread\_attr\_setdetachstate(&attr,PTHREAD\_CREATE\_JOINABLE);

// Creation of process

r1=pthread\_create(&producer,&attr,produce,NULL);

if(r1){

cout<<"Error in creating thread"<<endl;

exit(-1);

}

r1=pthread\_create(&consumer,&attr,consume,NULL);

if(r1){

cout<<"Error in creating thread"<<endl;

exit(-1);

}

// destroying the pthread\_attr

pthread\_attr\_destroy(&attr);

// Joining the thread

r1=pthread\_join(producer,NULL);

if(r1){

cout<<"Error in joining thread"<<endl;

exit(-1);

}

r1=pthread\_join(consumer,NULL);

if(r1){

cout<<"Error in joining thread"<<endl;

exit(-1);

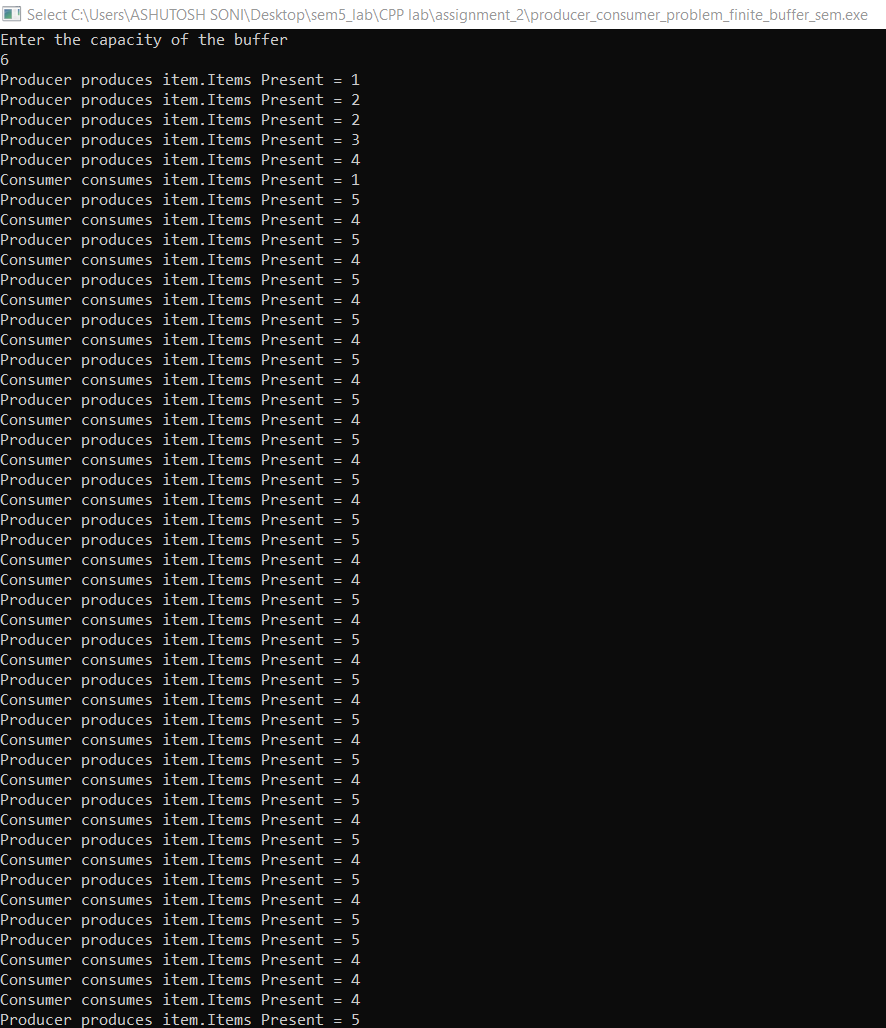
}

// Exiting thread

pthread\_exit(NULL);

return 0;

}



**Output of the**

**Program:**

Q5: Implement Merge-sort using Semaphores.

Ans:

**Merge-sort using Semaphores**

// Merge Sort Implementation using Semaphore

#include<bits/stdc++.h>

#include<pthread.h>

#include<semaphore.h>

#include <unistd.h>

using namespace std;

// Decalaration

int r1;

long N;

vector<int> array;

// Declaration of Semaphore

sem\_t S1,S2;

// sort first part of array

void\* sort\_first(void \*arg){

N=\*(long\* )arg;

int mid=N/2;

sort(array.begin(),array.begin()+mid);

sem\_post(&S1);

}

void\* sort\_second(void \*arg){

N=\*(long\*)arg;

int mid=N/2;

sort(array.begin()+mid,array.end());

sem\_post(&S2);

}

void\* merge\_array(void \*arg){

N=\*(long\*)arg;

int mid=N/2;

sem\_wait(&S1);

sem\_wait(&S2);

vector<int> left,right;

for(int i=0;i<mid;i++){

left.push\_back(array[i]);

}

for(int i=mid;i<N;i++){

right.push\_back(array[i]);

}

int m=left.size(),n=right.size();

int i=0,j=0,k=0;

while(i<m and j<n){

if(left[i]<=right[j]){

array[k]=left[i];

i++;

k++;

}

else{

array[k]=right[j];

j++;

k++;

}

}

while(i<m){

array[k]=left[i];

i++;

k++;

}

while(j<n){

array[k]=right[j];

k++;

j++;

}

// After merging Final array will be

cout<<"Final array is : "<<endl;

for(int i=0;i<N;i++){

cout<<array[i]<<" ";

}

cout<<endl;

}

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

// Initialization....

long N;

cout<<"Enter the total number of array you want to enter"<<endl;

cin>>N;

cout<<"Enter the array"<<endl;

for(int i=0;i<N;i++){

int num;

cin>>num;

array.push\_back(num);

}

// Declaration of thread

pthread\_t sort\_1,sort\_2,merge;

// Declaration of attribute......

pthread\_attr\_t attr;

// semaphore initialization

sem\_init(&S1,0,0);

sem\_init(&S2,0,0);

// pthread\_attr\_t initialization

pthread\_attr\_init(&attr);

pthread\_attr\_setdetachstate(&attr,PTHREAD\_CREATE\_JOINABLE);

// Creating thread

void \*ptr=&N;

r1=pthread\_create(&sort\_1,&attr,sort\_first,ptr);

if(r1){

cout<<"Error in creating thread"<<endl;

exit(-1);

}

r1=pthread\_create(&sort\_2,&attr,sort\_second,ptr);

if(r1){

cout<<"Error in creating thread"<<endl;

exit(-1);

}

r1=pthread\_create(&merge,&attr,merge\_array,ptr);

if(r1){

cout<<"Error in creating thread"<<endl;

exit(-1);

}

// destroying the pthread\_attr

pthread\_attr\_destroy(&attr);

// Joining the thread

r1=pthread\_join(sort\_1,NULL);

if(r1){

cout<<"Error in joining thread"<<endl;

exit(-1);

}

r1=pthread\_join(sort\_2,NULL);

if(r1){

cout<<"Error in joining thread"<<endl;

exit(-1);

}

r1=pthread\_join(merge,NULL);

if(r1){

cout<<"Error in joining thread"<<endl;

exit(-1);

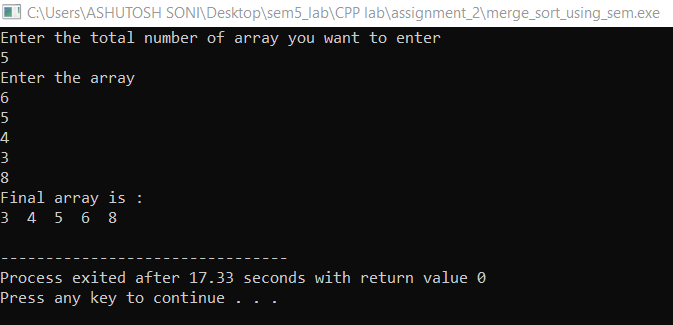
}

// Exiting thread

pthread\_exit(NULL);

return 0;

}



**Output of the**

**Program:**

Name: Ashutosh Soni

Id: 2018ucp1505

**ASSIGNMENT-3**

Q1: Implement Critical Section problem using semaphores with a monitor.

Ans:

**Critical Section problem using semaphores with a monitor**

// Header file include

#include<bits/stdc++.h>

#include<pthread.h>

using namespace std;

int times=0;

int x=0;

class monitor {

// Variables

int s;

// condition variable for not Zero

pthread\_cond\_t notZero;

// mutex variable for synchronization

pthread\_mutex\_t condLock;

public:

// Operation wait

void wait(){

pthread\_mutex\_lock(&condLock);

if(s==0){

pthread\_cond\_wait(&notZero,&condLock);

}

s=s-1;

pthread\_mutex\_unlock(&condLock);

}

// Operation Signal

void signal(){

pthread\_mutex\_lock(&condLock);

s=s+1;

pthread\_cond\_signal(&notZero);

pthread\_mutex\_unlock(&condLock);

}

// Constructor

monitor(){

// s=k

s=2;

pthread\_cond\_init(&notZero,NULL);

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

}

// Destructor

~monitor(){

pthread\_cond\_destroy(&notZero);

pthread\_mutex\_destroy(&condLock);

}

}

// Global Object of Monitor

Sem

;

// Critical Section of the Problem

void critical\_section(){

cout<<"Enters ino critical Section"<<endl;

x++;

cout<<"Exiting critical Section of ";

}

// Main Process for P and Q

void\* process\_P(void \*arg){

// Loop Forever

while(times<100){

// Non Critical Section

times++;

// Wait Operation

Sem.wait();

cout<<"P ";

// Critical Section code

critical\_section();

// Signal Operation

cout<<"P"<<endl;

Sem.signal();

}

}

void\* process\_Q(void \*arg){

// Loop Forever

while(times<100){

// Non Critical Section

times++;

// Wait Operation

Sem.wait();

cout<<"Q ";

// Critical Section code

critical\_section();

// Signal Operation

cout<<"Q"<<endl;

Sem.signal();

}

}

int main(){

// Declaration

pthread\_t process\_p, process\_q;

pthread\_attr\_t attr;

// Initialization

pthread\_attr\_init(&attr);

pthread\_attr\_setdetachstate(&attr,PTHREAD\_CREATE\_JOINABLE);

// Creation

pthread\_create(&process\_p,&attr,process\_P,NULL);

pthread\_create(&process\_q,&attr,process\_Q,NULL);

// Joining

pthread\_join(process\_p,NULL);

pthread\_join(process\_q,NULL);

// Destroying

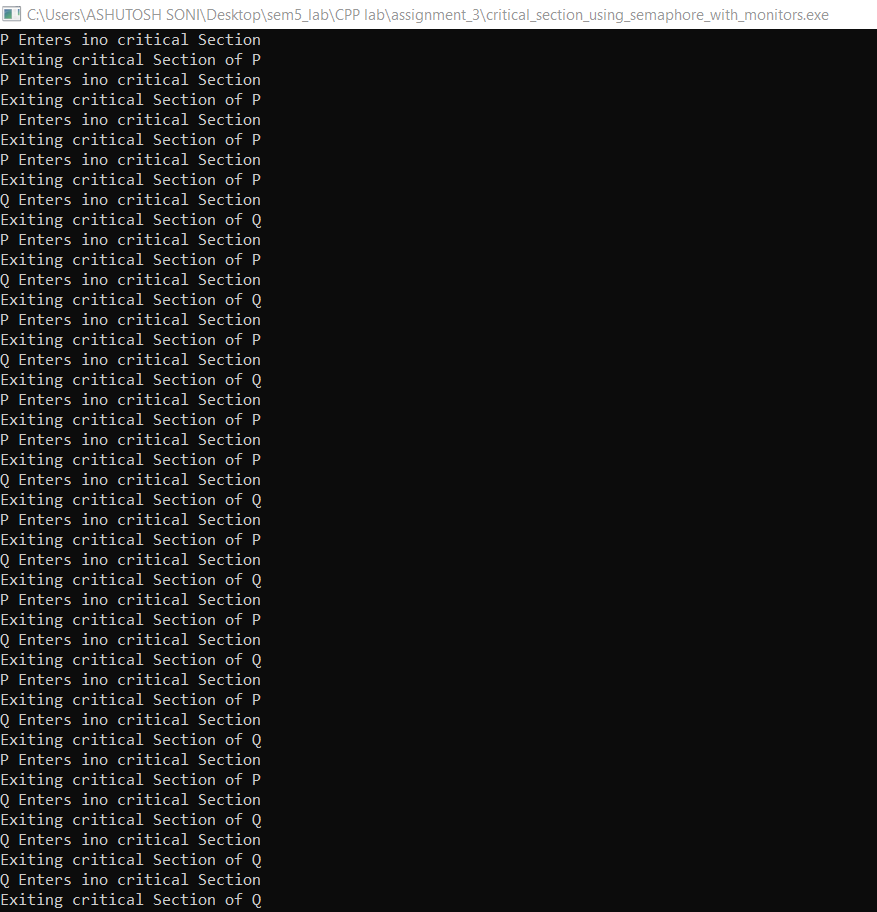
pthread\_attr\_destroy(&attr);

pthread\_exit(NULL);

return 0;

}

**Output of the program:**



Q2: Implement the solution of producer-consumer bounded buffer problem with a

monitor.

Ans:

**Producer-consumer bounded buffer problem with a monitor**

// Header file include

#include<bits/stdc++.h>

#include<pthread.h>

#include<unistd.h>

using namespace std;

int times=0;

class Monitor{

// buffer for the store

int buffer=0;

// capacity of the store

int capacity;

// condtion variable for Not Empty and Not Full

pthread\_cond\_t notEmpty,notFull;

// mutex variable for synchorization

pthread\_mutex\_t condLock;

public:

// Append operation

void append(){

pthread\_mutex\_lock(&condLock);

cout<<"Producer is producing"<<endl;

// Wait for buffer to not Full

if(buffer==capacity){

pthread\_cond\_wait(&notFull,&condLock);

}

buffer++;

pthread\_cond\_signal(&notEmpty);

pthread\_mutex\_unlock(&condLock);

}

// Take operation

void take(){

pthread\_mutex\_lock(&condLock);

cout<<"Consumer is taking"<<endl;

// Wait for Buffer to not Empty

if(buffer==0){

pthread\_cond\_wait(&notEmpty,&condLock);

}

buffer--;

pthread\_cond\_signal(&notFull);

pthread\_mutex\_unlock(&condLock);

}

// Constructor

Monitor(){

capacity=25;

pthread\_cond\_init(&notEmpty,NULL);

pthread\_cond\_init(&notFull,NULL);

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

}

// Destructor

~Monitor(){

pthread\_cond\_destroy(&notEmpty);

pthread\_cond\_destroy(&notFull);

pthread\_mutex\_destroy(&condLock);

}

}

// Global variable of monitor where producer is storing and consumer is taking.....

store;

// Produce Function

void\* produce(void \*arg){

while(times<1000){

sleep((rand()%100)\*0.01);

store.append();

times++;

}

}

// Consumer Function

void\* consume(void \*arg){

while(times<1000){

sleep((rand()%100)\*0.02);

store.take();

times++;

}

}

int main(){

// Declaration...

pthread\_t producer, consumer;

pthread\_attr\_t attr;

// Initialization

pthread\_attr\_init(&attr);

pthread\_attr\_setdetachstate(&attr,PTHREAD\_CREATE\_JOINABLE);

// Creation

pthread\_create(&producer,&attr,produce,NULL);

pthread\_create(&consumer,&attr,consume,NULL);

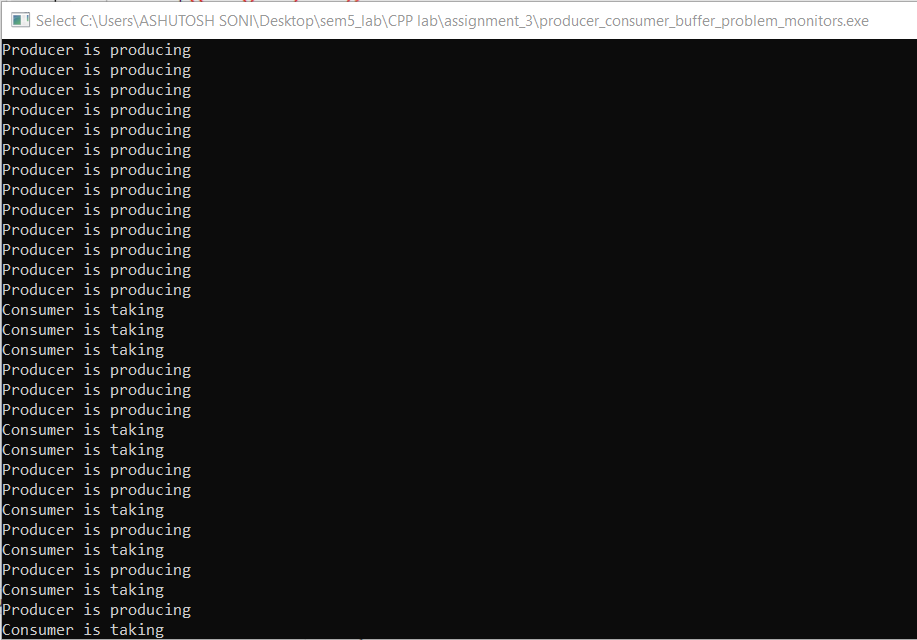
// Destroying

pthread\_attr\_destroy(&attr);

pthread\_exit(NULL);

return 0;

}

**Output of the program:**

Q3: Implement the solution of Readers and writers with a monitor.

Ans:

**Readers and writers problem with a monitor**

// Header file include

#include<bits/stdc++.h>

#include<pthread.h>

#include<unistd.h>

using namespace std;

int items=10;

class monitor {

// number of readers

int readers;

// number of writers

int writers;

// number of readers waiting

int waitreaders;

// number of writers waiting

int waitwriters;

// condition variable for readers

pthread\_cond\_t canread;

// condtion variable for writers

pthread\_cond\_t canwrite;

// mutex for synchornization

pthread\_mutex\_t condLock;

public:

// Start read Function

void start\_read(int i){

pthread\_mutex\_lock(&condLock);

if(writers == 1 and waitwriters > 0){

waitreaders++;

pthread\_cond\_wait(&canread,&condLock);

waitreaders--;

}

readers++;

cout<<"Reader "<< i <<" is reading"<<endl;

pthread\_mutex\_unlock(&condLock);

pthread\_cond\_broadcast(&canread);

}

// End read function

void end\_read(int i){

pthread\_mutex\_lock(&condLock);

if(--readers == 0){

pthread\_cond\_signal(&canwrite);

}

pthread\_mutex\_unlock(&condLock);

}

// Start write Function

void start\_write(int i){

pthread\_mutex\_lock(&condLock);

if(writers == 1 or readers > 0){

++waitwriters;

pthread\_cond\_wait(&canwrite,&condLock);

--waitwriters;

}

writers = 1;

cout<<"Writer "<<i<<" is writing"<<endl;

pthread\_mutex\_unlock(&condLock);

}

// End Write Function

void end\_write(int i){

pthread\_mutex\_lock(&condLock);

writers =0;

if(waitreaders > 0){

pthread\_cond\_signal(&canread);

}

else{

pthread\_cond\_signal(&canwrite);

}

pthread\_mutex\_unlock(&condLock);

}

// constrcutor

monitor(){

readers=0;

writers=0;

waitreaders=0;

waitwriters=0;

pthread\_cond\_init(&canread,NULL);

pthread\_cond\_init(&canwrite,NULL);

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

}

// destructor

~monitor(){

pthread\_cond\_destroy(&canread);

pthread\_cond\_destroy(&canwrite);

pthread\_mutex\_destroy(&condLock);

}

}

// Global Object of monitor class handles readers and writers

library

;

// Reader funciton

void\* reader(void \*arg){

int c=0;

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

// Read items

while(c < items){

sleep(1);

library.start\_read(i);

library.end\_read(i);

c++;

}

}

// Writers function

void\* writer(void \*arg){

int c=0;

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

while(c < items){

sleep(1);

library.start\_write(i);

library.end\_write(i);

c++;

}

}

int main(){

// Declaration

pthread\_t read[items] ,write[items];

pthread\_attr\_t attr;

int id[items];

// Initalization

pthread\_attr\_init(&attr);

pthread\_attr\_setdetachstate(&attr,PTHREAD\_CREATE\_JOINABLE);

for(int i=0;i<items;i++){

id[i]= i;

// Creating thread

// for readers

pthread\_create(&read[i],&attr,reader,&id[i]);

// for writers

pthread\_create(&write[i],&attr,writer,&id[i]);

}

// Joining threads

// readers

for(int i=0;i<items;i++){

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

}

// writers

for(int i=0;i<items;i++){

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

}

// destroying

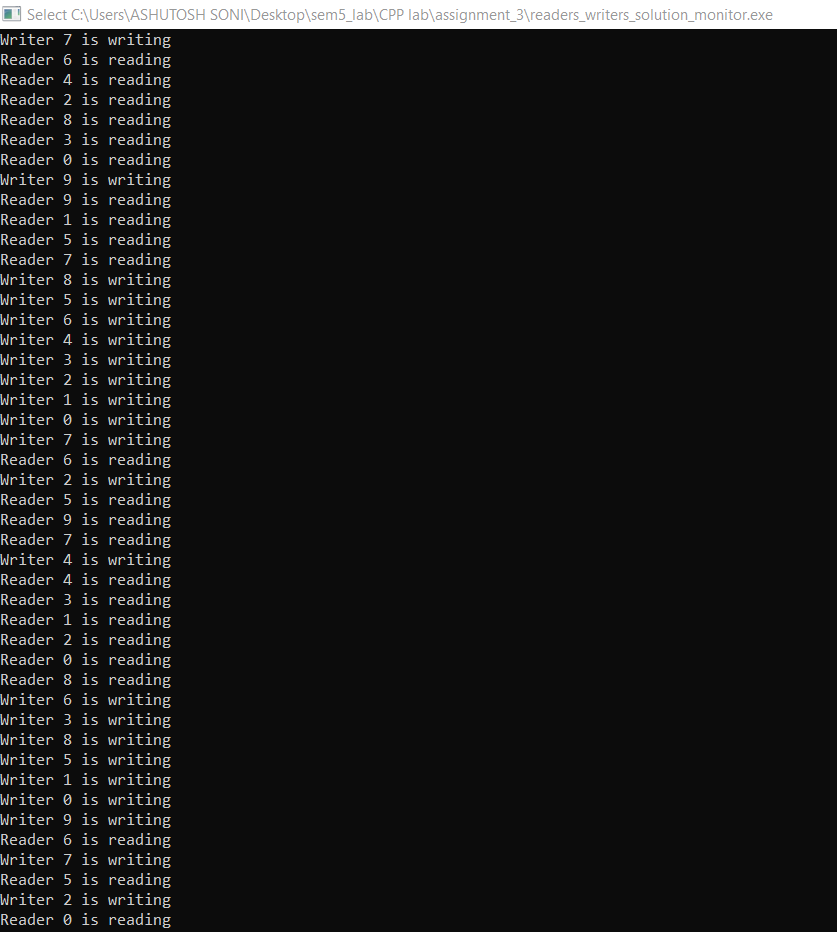
pthread\_attr\_destroy(&attr);

pthread\_exit(NULL);

return 0;

}

**Output of the Program:**

****

Q4: Implement the solution of Dining philosophers with a monitor.

Ans:

**Dining philosophers Problem with a monitor**

// Header file include

#include<bits/stdc++.h>

#include<pthread.h>

#include<unistd.h>

using namespace std;

#define N 10

#define THINKING 2

#define HUNGRY 1

#define EATING 0

#define LEFT (phnum + 4)%N

#define RIGHT (phnum + 1)%N

// Philospher index

int phil[N];

int times=200;

class monitor {

// state of the philospher

int state[N];

// Philospher condition variable

pthread\_cond\_t phcond[N];

// mutex variable for synchronization

pthread\_mutex\_t condLock;

public:

// Test for the desired condtion

// i.e. Left and Right philospher are not reading

void test(int phnum){

if(state[(phnum+1)%5] != EATING and state[(phnum+4)%5] != EATING and state[phnum] ==HUNGRY){

state[phnum] = EATING;

pthread\_cond\_signal(&phcond[phnum]);

}

}

// Take Fork function

void take\_fork(int phnum){

pthread\_mutex\_lock(&condLock);

// Indicates it is hungry

state[phnum]=HUNGRY;

// test for condition

test(phnum);

// If unable to eat.. wait for the signal

if(state[phnum]!=EATING){

pthread\_cond\_wait(&phcond[phnum],&condLock);

}

cout<<"Philospher "<<phnum<<" is Eating"<<endl;

pthread\_mutex\_unlock(&condLock);

}

// Put Fork function

void put\_fork(int phnum){

pthread\_mutex\_lock(&condLock);

// Indicates that I am thinking

state[phnum]=THINKING;

test(RIGHT);

test(LEFT);

pthread\_mutex\_unlock(&condLock);

}

// constructor

monitor(){

for(int i=0;i<N;i++){

state[i] = THINKING;

}

for(int i=0;i<N;i++){

pthread\_cond\_init(&phcond[i],NULL);

}

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

}

// destructor

~monitor(){

for(int i=0;i<N;i++){

pthread\_cond\_destroy(&phcond[i]);

}

pthread\_mutex\_destroy(&condLock);

}

}

// Global Object of the monitor

phil\_object;

void\* philospher(void \*arg){

int c=0;

while(c<times){

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

sleep(1);

phil\_object.take\_fork(i);

sleep(0.5);

phil\_object.put\_fork(i);

c++;

}

}

int main(){

// Declaration...

pthread\_t thread\_id[N];

pthread\_attr\_t attr;

// Initialization...

pthread\_attr\_init(&attr);

pthread\_attr\_setdetachstate(&attr,PTHREAD\_CREATE\_JOINABLE);

for(int i=0;i<N;i++){

phil[i]=i;

}

// Creating...

for(int i=0;i<N;i++){

pthread\_create(&thread\_id[i],&attr,philospher,&phil[i]);

cout<<"Philospher "<<i+1<<" is thinking..."<<endl;

}

// Joining....

for(int i=0;i<N;i++){

pthread\_join(thread\_id[i],NULL);

}

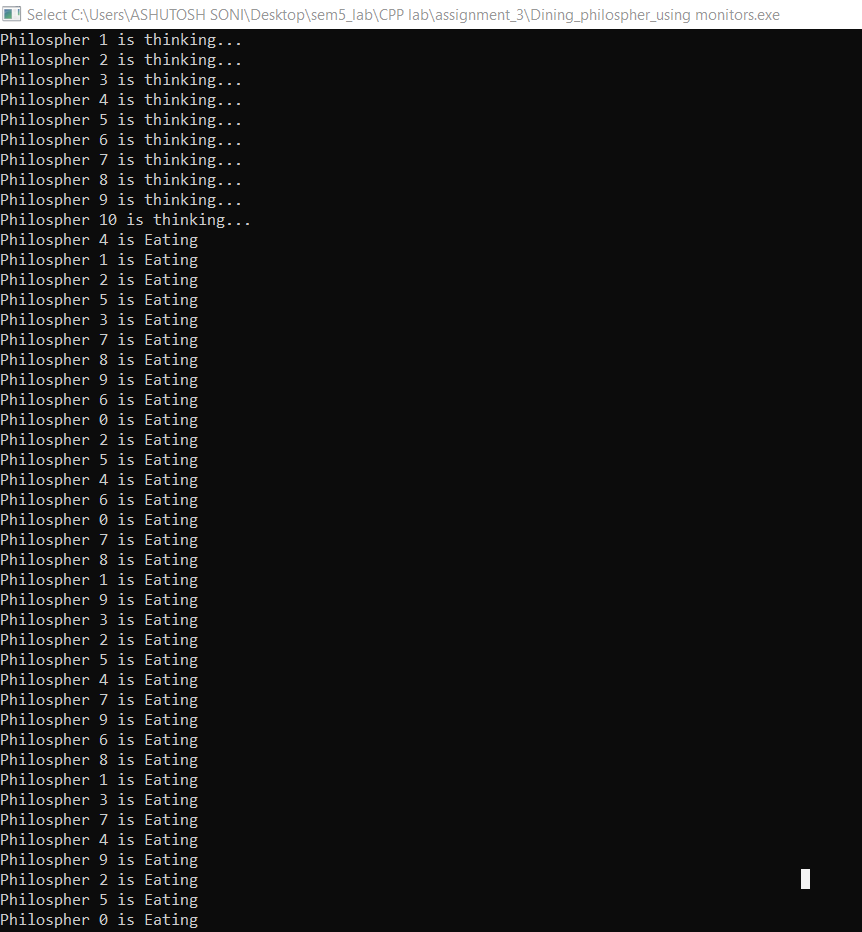
// Destroying

pthread\_attr\_destroy(&attr);

pthread\_exit(NULL);

return 0;

}

**Output of the program:**

------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------