1

1. Let's consider three semaphores and a mutex:

Pthread\_mutex\_t assist\_Cust;

sem\_t barber;

sem\_t customer;

sem\_t modifySeats;

Let there be two inputs:

No\_of\_chairs and cust\_total of int data type

* brb:( barber functionality)

Semaphore Customer lock - Get a customer or the barber should sleep if there is no customer

Semaphore modify seat count - Get access to seats

Let’s start with having 1 chair in the shop free and unlock the semaphore modify seat count.

Unlock the semaphore barber so he is ready to assist the customer.

Lock the mutex assist\_Cust so that the barber doesn’t get used by other threads in the process.

Assume, the customer arrived and the barber served the customer. Unlock the mutex so the barber can be reused again.

* Cust: ( customer functionality)

If the unoccupied seats in the barbershop are more than or equal to 1, then customers can occupy the seat available.

Unlock semaphore: customer, so that he is ready to be served by the barber.

Unlock semaphore: modify seat count and lock the barber for his assistance.

Using pthread\_create, create a customer thread and work on the customer and barber functionality

b) Program code:

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <semaphore.h>

#include <pthread.h>

#include <stdbool.h>

#include <time.h>

int start\_t=0, end\_t=0;

double total\_t;

int i;

sem\_t Cust\_Count;

sem\_t Mutex;

sem\_t Barber;

sem\_t Cust\_Status;

sem\_t Baarb\_Status;

void \*customer(void \*num);

void \*barber(void \*);

void \*HairCut();

void \*getHairCut();

void fun();

int chairs=6;

int currCustomers=0;

int customer\_count=3;

int hairCutTime = 0;

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

pthread\_t bid;

if (argc == 3) {

chairs = atoi(argv[1]);

customer\_count = atoi(argv[2]);

}

else if (argc != 1) {

printf("USAGE: con4b chairs customers\n");

exit(EXIT\_FAILURE);

}

int Cust\_Arr[customer\_count];

pthread\_t cid[customer\_count];

for (i=0; i<customer\_count; i++) {

Cust\_Arr[i] = i;

}

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

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

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

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

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

pthread\_create(&bid, NULL, barber, NULL);

for (i=0; i<customer\_count; i++) {

pthread\_create(&cid[i], NULL, customer, (void \*)&Cust\_Arr[i]);

int rNum = rand()%5 + 1;

sleep(rNum);

}

for (i=0; i<customer\_count; i++) {

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

}

end\_t = clock();

total\_t = (double)(end\_t - start\_t) / CLOCKS\_PER\_SEC;

printf("Total time taken by CPU: %f clocks per second\n", total\_t );

printf("Program End\n");

}

void \*customer(void \*b){

int custNum = \*(int \*)b;

sem\_wait(&Mutex);

if(currCustomers == chairs){

printf("customer %d turned away, chairs full\n", custNum);

sem\_post(&Mutex);

fun();

}

else{

currCustomers++;

}

printf("customer %d arrived. Chairs Avail: %d\n", custNum, (5-\currCustomers) );

sem\_post(&Mutex);

sem\_post(&Cust\_Count);

sem\_wait(&Barber);

printf("customer %d getting haircut\n", custNum);

hairCutTime = rand()%10 + 1;

HairCut();

sem\_post(&Cust\_Status);

sem\_wait(&Baarb\_Status);

sem\_wait(&Mutex);

currCustomers--;

sem\_post(&Mutex);

if(custNum == customer\_count){

printf("Barbershop closing\n");

exit(0);

}

}

void fun(){

while(1){

}

}

void \*barber(void \*a){

while(1){

sem\_wait(&Cust\_Count);

sem\_post(&Barber);

printf("barber cutting hair\n");

getHairCut();

printf("barber finished cutting hair\n");

sem\_wait(&Cust\_Status);

sem\_post(&Baarb\_Status);

}

}

void \*HairCut(){

sleep(hairCutTime);

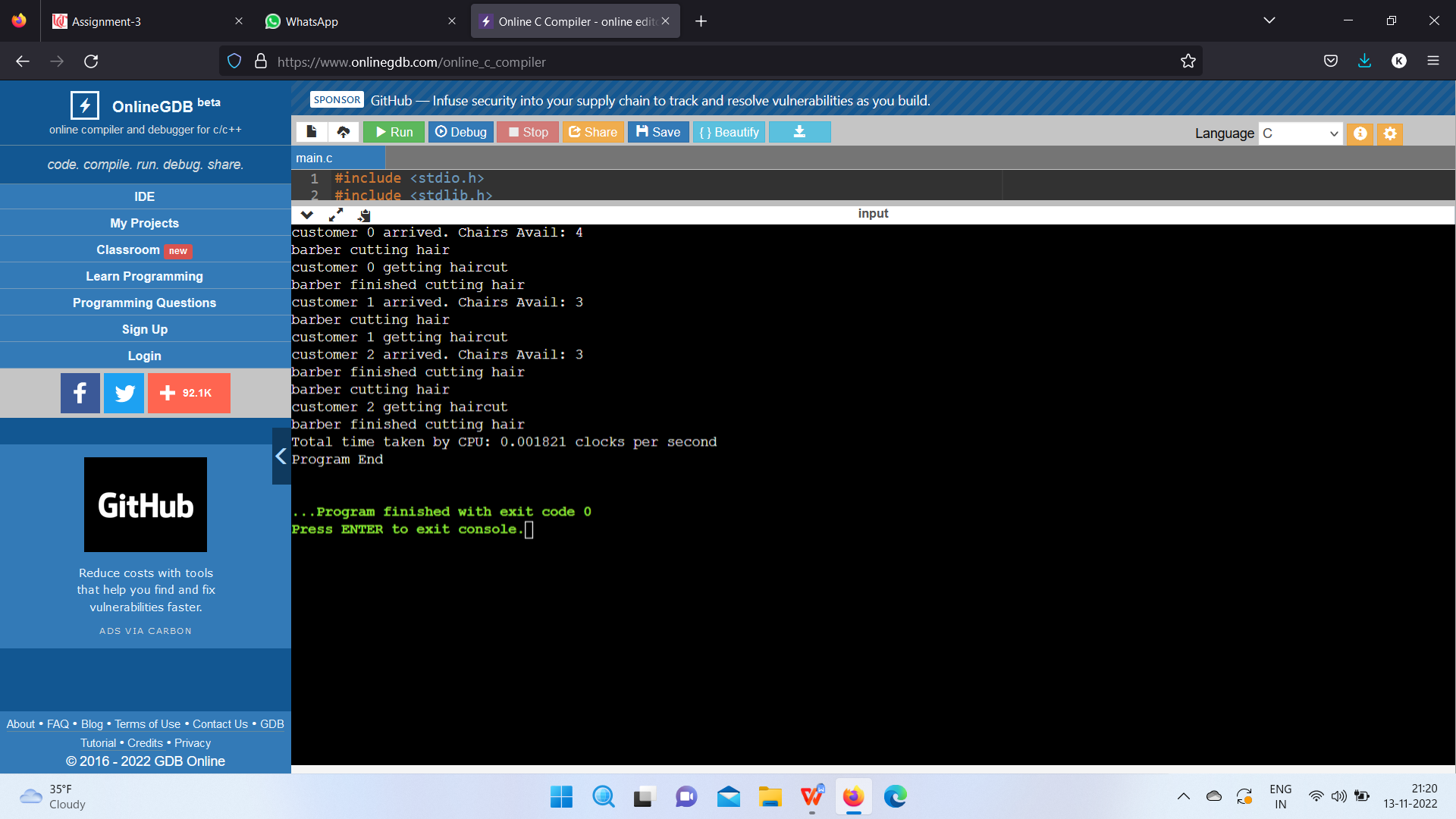
}

void \*getHairCut(){

sleep(hairCutTime);

}

c) Provide the test suite and the results of the Alpha Testing.



2

Program code:

import time

import threading

import random

def cThread(function,name,items):

return threading.Thread(target=function,name=name,args=items)

class CigaretteSmoking:

def \_\_init\_\_(self):

self.iList=['Tobacco','Paper','Matches']

self.itemsAvailable=[False,False,False]

self.terminate=False

self.mutex=threading.Condition()

self.agentWait=threading.Semaphore(0)

t1=cThread(self.smokerFun,"Tobacco",(1,2))

t2=cThread(self.smokerFun,"Paper",(0,2))

t3=cThread(self.smokerFun,"Matches",(0,1))

t1.start()

t2.start()

t3.start()

self.agentThread=threading.Thread(target=self.agentFun)

self.agentThread.start()

def agentFun(self):

for i in range(3):

rand1=random.randint(0,2)

rand2=random.randint(0,2)

if(rand1==rand2):

rand2+=1

rand2%=3

randomItems=[rand1,rand2]

self.mutex.acquire()

print('The two items produced by the Agent: {0} and {1}'.\

format(self.iList[randomItems[0]],\

self.iList[randomItems[1]]))

self.itemsAvailable[randomItems[0]]=True

self.itemsAvailable[randomItems[1]]=True

self.mutex.notify\_all()

self.mutex.release()

self.agentWait.acquire()

def smokerFun(self,itemReq1,itemReq2):

currentThread=threading.currentThread().getName()

while(True):

self.mutex.acquire()

while(self.itemsAvailable[itemReq1]==False or self.itemsAvailable[itemReq2]==False):

self.mutex.wait()

self.mutex.release()

if(self.terminate==True):

break

self.itemsAvailable[itemReq1]=False

self.itemsAvailable[itemReq2]=False

print('{0} supplied,and the smoking began.'.format(currentThread))

randomTime=random.randint(1,5)

time.sleep(randomTime+1)

print('The smoking process was completed successfully.')

self.agentWait.release()

def waitForCompletion(self):

self.agentThread.join()

self.mutex.acquire()

self.terminate=True

self.itemsAvailable=[True,True,True]

self.mutex.notify\_all()

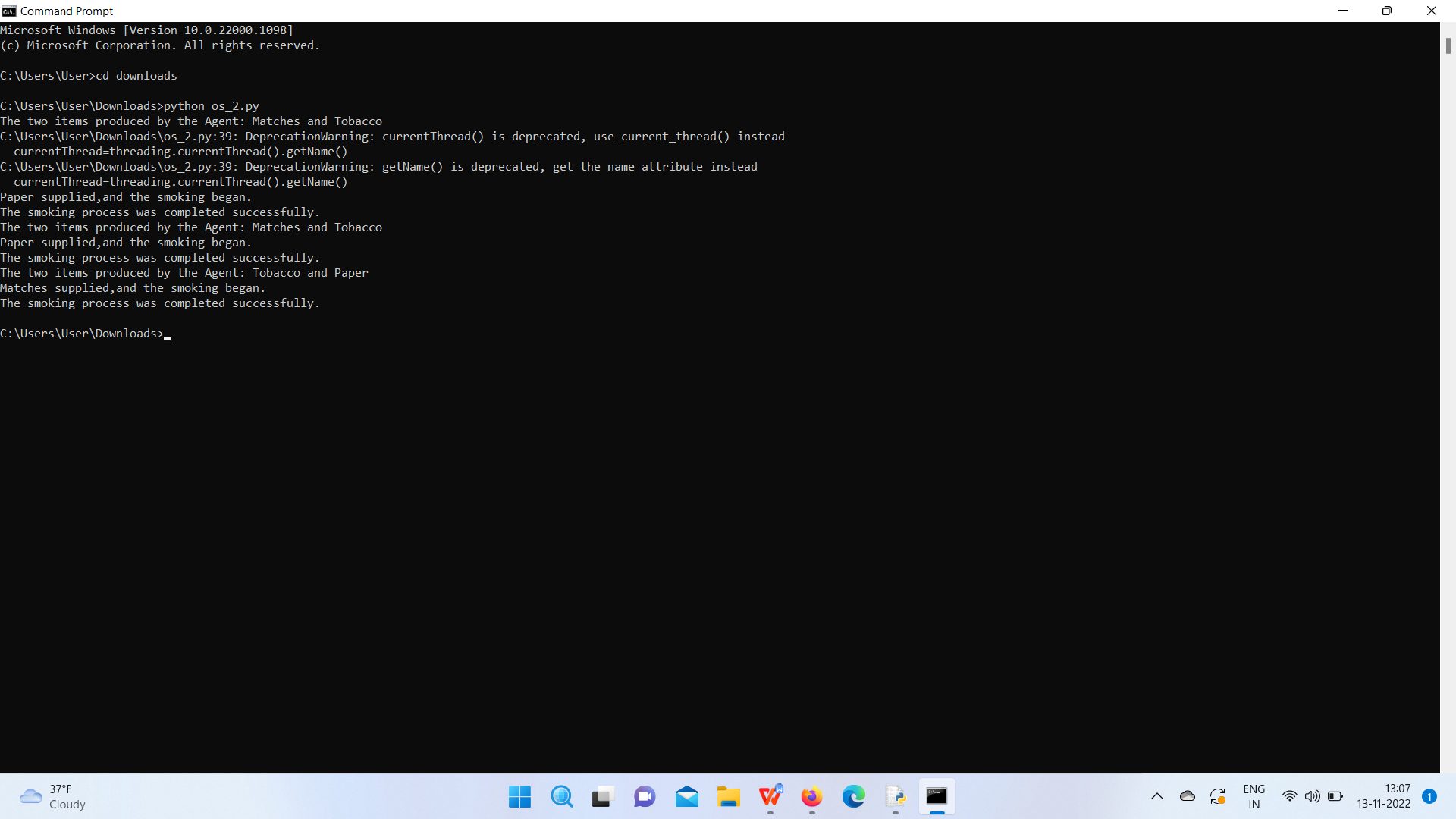
self.mutex.release()

if \_\_name\_\_== "\_\_main\_\_":

instance=CigaretteSmoking()

instance.waitForCompletion()

Output:



3

b) Program Code:

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

#include <semaphore.h>

#include <stdint.h>

#include <malloc.h>

#ifdef \_WIN32

#include <Windows.h>

#else

#include <unistd.h>

#endif

//Define the some constant values using macros

#define MAX\_SLEEP\_TIME 3

#define NUM\_OF\_STUDENTS 4

#define NUM\_OF\_HELP 2

#define NUM\_OF\_SEATS 2

//Declare an array of threads

pthread\_t student\_array[NUM\_OF\_STUDENTS];

//Declare a thread for teaching assistant

pthread\_t ta\_thread;

//Initiale the Pthread mutex.

pthread\_mutex\_t mutex\_lock;

//Create semaphores

sem\_t students\_sem;

sem\_t ta\_sem;

//Initialize a variable of int type

int waiting\_students = 0;

//Declare an aaray of type int

int thread\_helped\_counter[NUM\_OF\_STUDENTS];

// just a random number

unsigned int seed = 92;

//Declare a variabe

int sleep\_time;

//Declare an array

int waiting\_queue[NUM\_OF\_HELP];

//Declare the prototype of the functions

void \* students(void \* param);

void \* ta();

void create\_students();

void create\_ta();

void join\_students();

void join\_ta();

//Declare a variable

int value;

//Start the main function of the program

int main()

{

sem\_init(&students\_sem,0,1);

sem\_init(&ta\_sem,0,1);

create\_students();

create\_ta();

join\_students();

join\_ta();

sem\_destroy(&students\_sem);

sem\_destroy(&ta\_sem);

}

//Use the function called Join\_ta()

void join\_ta()

{

if(pthread\_join(ta\_thread, NULL)){

fprintf(stderr, "Error joining thread\n");

}

}

//Use the function called Join\_students()

void join\_students(){

// this function joins all the student thread

int i;

for(i = 0; i < NUM\_OF\_STUDENTS; i++){

if(pthread\_join(student\_array[i], NULL)){

fprintf(stderr, "Error joining thread\n");

}

}

}

//Use the function called create\_ta()

void create\_ta(){

if(pthread\_create(&ta\_thread, NULL, ta, NULL)){

fprintf(stderr, "TA thread creation error\n");

}

}

//Use the function called create\_students()

void create\_students(){

int i;

for(i = 0; i < NUM\_OF\_STUDENTS; i++){

int \*arg = malloc(sizeof(\*arg));

\*arg = i;

if(pthread\_create(&student\_array[i], NULL, students, (void \*)arg)){

fprintf(stderr, "Student thread creation error\n");

}

}

if(pthread\_create(&ta\_thread, NULL, ta, NULL)){

fprintf(stderr, "TA thread creation error\n");

}

}

//Define the function called students

void \* students(void \* param)

{

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

int sleeptime = (rand\_r(&seed)% MAX\_SLEEP\_TIME)+1;

printf("Student %d programming for %d seconds.\n", i, sleeptime);

sleep(sleeptime);

pthread\_mutex\_lock(&mutex\_lock);

if(thread\_helped\_counter[i] >= NUM\_OF\_HELP){

pthread\_mutex\_unlock(&mutex\_lock);

return NULL;

}

pthread\_mutex\_unlock(&mutex\_lock);

pthread\_mutex\_lock(&mutex\_lock);

sem\_getvalue(&ta\_sem, &value);

pthread\_mutex\_unlock(&mutex\_lock);

if(value > 0){

sem\_wait(&ta\_sem);

pthread\_mutex\_lock(&mutex\_lock);

if(waiting\_students >= 0 && waiting\_students < NUM\_OF\_SEATS){

if(waiting\_students == 0){

waiting\_queue[0] = i;

printf("\t\t Student %d takes a seat, # of waiting students = %d\n",

waiting\_queue[0], waiting\_students);

waiting\_students++;

}

else if(waiting\_students == 1){

waiting\_queue[1] = i;

printf("\t\t Student %d takes a seat, # of waiting students = %d\n",waiting\_queue[1], waiting\_students);

waiting\_students++;

}

pthread\_mutex\_unlock(&mutex\_lock);

sem\_post(&students\_sem);

sleep\_time = (rand\_r(&seed)% MAX\_SLEEP\_TIME)+1;

sleep(sleep\_time);

}

else{

printf("\nwaiting list > 2\n");

pthread\_mutex\_unlock(&mutex\_lock);

printf("\t\t\tStudent %d will try later\n", i);

sleep((rand\_r(&seed)% MAX\_SLEEP\_TIME)+1);

students(&i);

}

}

else{

sem\_post(&students\_sem);

}

//Call the students()

students((void\*)&i);

}

//Define the ta() which return type is void pointer

void \* ta()

{

int i,result;

for(i=0;i< 4; i++){

result+=thread\_helped\_counter[i];

}

if(result == 8){

return NULL;

}

sem\_wait(&students\_sem);

pthread\_mutex\_lock(&mutex\_lock);

sleep\_time = (rand\_r(&seed)% MAX\_SLEEP\_TIME)+1;

if(waiting\_students == 1 && thread\_helped\_counter[waiting\_queue[0]] < 2){

waiting\_students--;

printf("Helping a student for %d , # of students waiting = %d\n",sleep\_time, waiting\_students);

printf("Student %d recieving help\n", waiting\_queue[0]);

thread\_helped\_counter[waiting\_queue[0]]++;

}

else if(waiting\_students == 2 && thread\_helped\_counter[waiting\_queue[1]] < 2){

waiting\_students--;

printf("Helping a student for %d , # of students waiting = %d\n",sleep\_time, waiting\_students);

printf("Student %d recieving help\n", waiting\_queue[1]);

thread\_helped\_counter[waiting\_queue[1]]++;

}

pthread\_mutex\_unlock(&mutex\_lock);

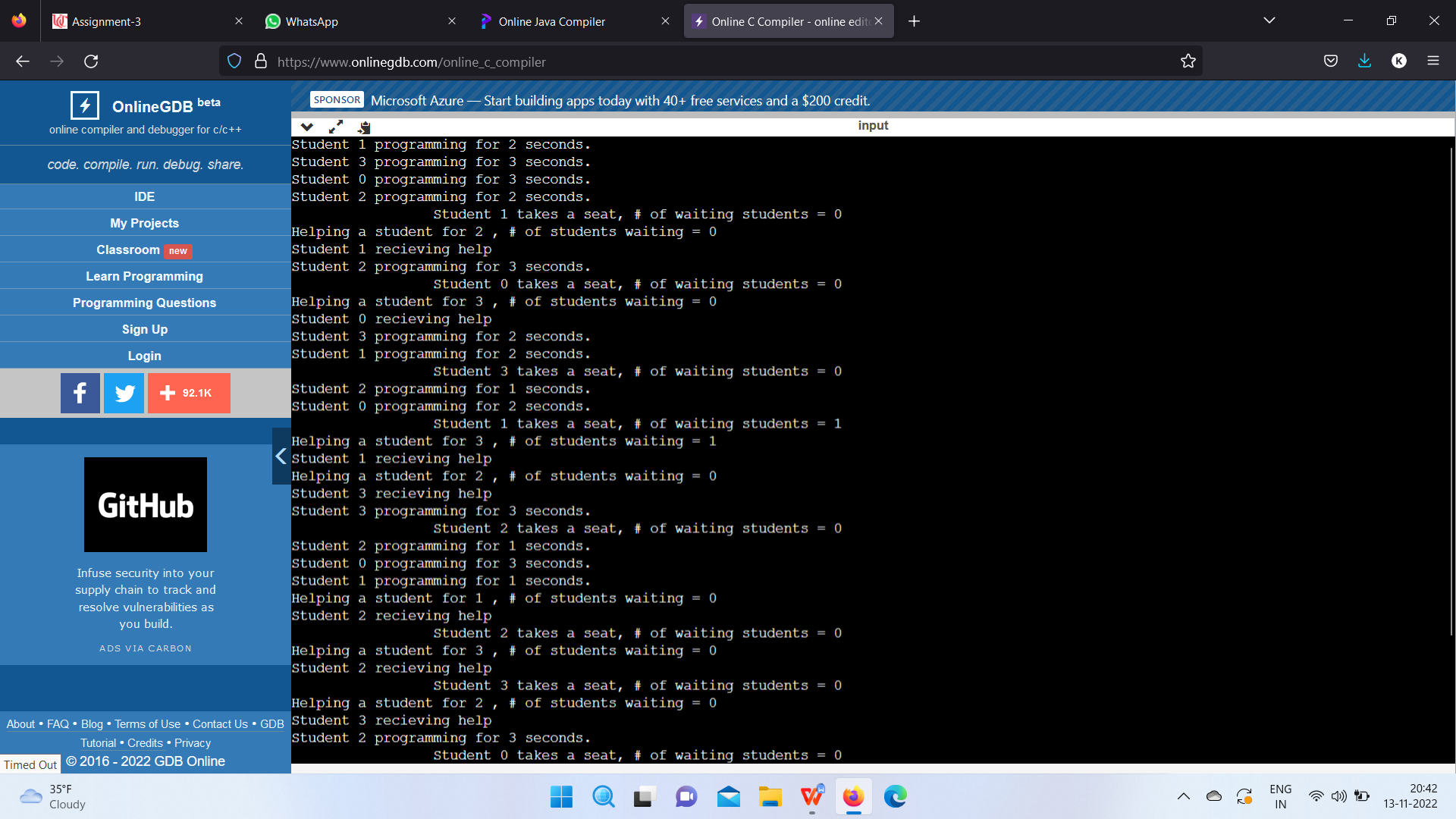
sleep(sleep\_time);

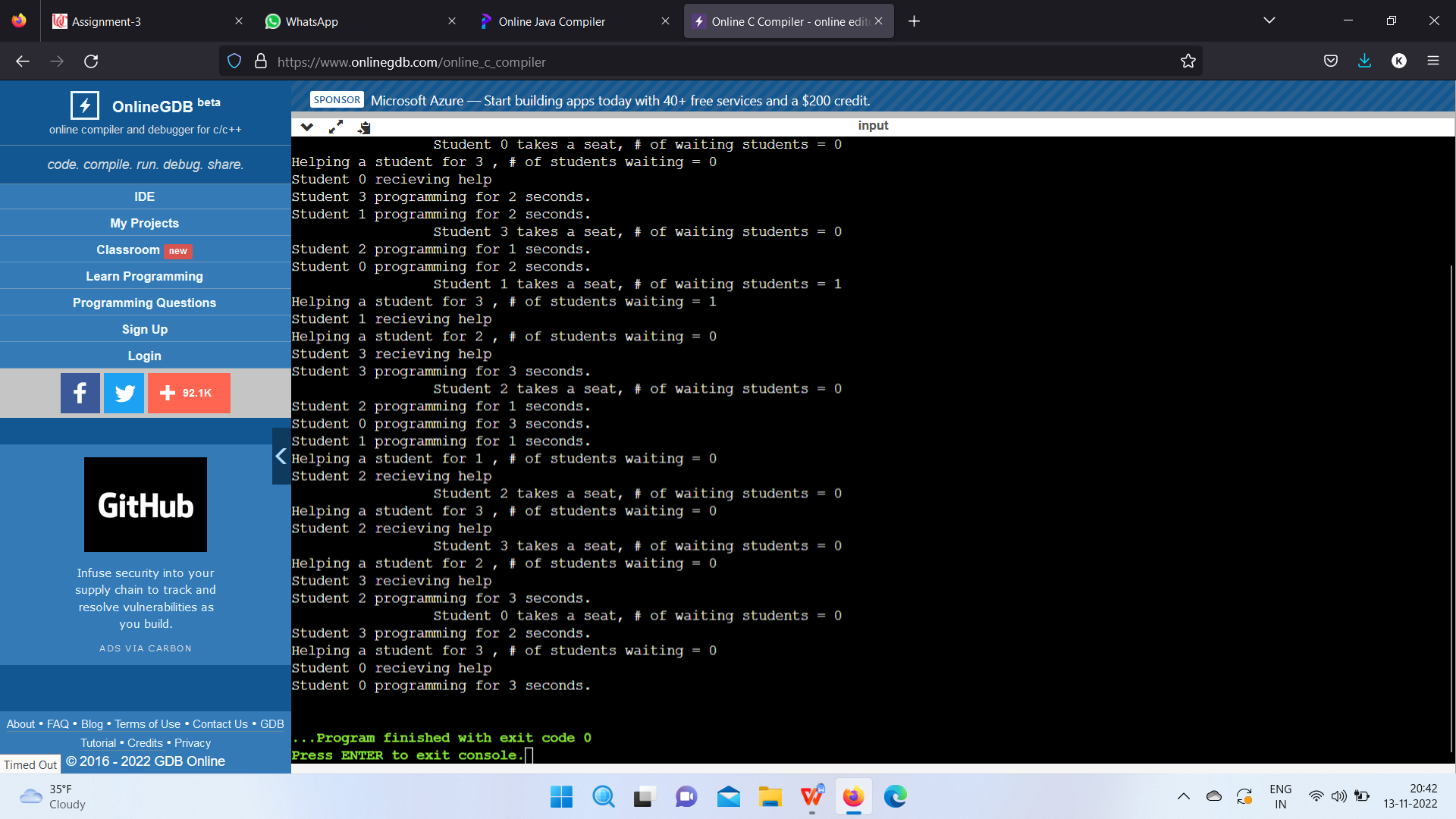
sem\_post(&ta\_sem);

ta();

}

c)





4

The readers-writers problem relates to an object such as a file that is shared between multiple processes. Some of these processes are readers i.e. they only want to read the data from the object and some of the processes are writers i.e. they want to write into the object.The readers-writers problem is used to manage synchronization so that there are no problems with the object data. For example - If two readers access the object at the same time there is no problem. However if two writers or a reader and writer access the object at the same time, there may be problems.To solve this situation, a writer should get exclusive access to an object i.e. when a writer is accessing the object, no reader or writer may access it. However, multiple readers can access the object at the same time

.

The readers-writers problem is a classical problem of process synchronization, it relates to a data set such as a file that is shared between more than one process at a time. Among these various processes, some are Readers - which can only read the data set; they do not perform any updates, some are Writers - can both read and write in the data sets.

The readers-writers problem is used for managing synchronization among various reader and writer process so that there are no problems with the data sets, i.e. no inconsistency is generated.

Let's understand with an example - If two or more than two readers want to access the file at the same point in time there will be no problem. However, in other situations like when two writers or one reader and one writer wants to access the file at the same point of time, there may occur some problems, hence the task is to design the code in such a manner that if one reader is reading then no writer is allowed to update at the same point of time, similarly, if one writer is writing no reader is allowed to read the file at that point of time and if one writer is updating a file other writers should not be allowed to update the file at the same point of time. However, multiple readers can access the object at the same time.

Step 2

Case Process 1 Process 2 Allowed / Not Allowed

Case 1 Writing Writing Not Allowed

Case 2 Reading Writing Not Allowed

Case 3 Writing Reading Not Allowed

Case 4 Reading Reading Allowed

The solution of readers and writers can be implemented using binary semaphores.

We use two binary semaphores "write" and "mutex", where binary semaphore can be defined as:

Semaphore: A semaphore is an integer variable in S, that apart from initialization is accessed by only two standard atomic operations - wait and signal, whose definitions are as follows:

wait( S ){

while( S <= 0) ;

S--;

}

signal( S ){

S++;

}

Step 3

The code of the reader process is given below -

static int readcount = 0;

wait (mutex);

readcount ++; // on each entry of reader increment readcount

if (readcount == 1){

  wait (write);

}

signal(mutex);

--READ THE FILE?

wait(mutex);

readcount --; // on every exit of reader decrement readcount

if (readcount == 0) {

signal (write);

}

signal(mutex);

In the above code of reader, mutex and write are semaphores that have an initial value of 1, whereas the readcount variable has an initial value as 0. Both mutex and write are common in reader and writer process code, semaphore mutex ensures mutual exclusion and semaphore write handles the writing mechanism.

The readcount variable denotes the number of readers accessing the file concurrently. The moment variable readcount becomes 1, wait operation is used to write semaphore which decreases the value by one. This means that a writer is not allowed how to access the file anymore. On completion of the read operation, readcount is decremented by one. When readcount becomes 0, the signal operation which is used to write permits a writer to access the file.

The code that defines the writer process is given below:

wait(write);

WRITE INTO THE FILE

signal(wrt);

If a writer wishes to access the file, wait operation is performed on write semaphore, which decrements write to 0 and no other writer can access the file. On completion of the writing job by the writer who was accessing the file, the signal operation is performed on write.

**CASE 1:** WRITING - WRITING → NOT ALLOWED. That is when two or more than two processes are willing to write, then it is not allowed. Let us see that our code is working accordingly or not? Explanation :

The initial value of semaphore write = 1

Suppose two processes P0 and P1 wants to write, let P0 enter first the writer code, The moment P0 enters

Wait( write ); will decrease semaphore write by one, now write = 0

And continue WRITE INTO THE FILE

Now suppose P1 wants to write at the same time (will it be allowed?) let's see.

P1 does Wait( write ), since the write value is already 0, therefore from the definition of wait, it will go into an infinite loop (i.e. Trap), hence P1 can never write anything, till P0 is writing.

Now suppose P0 has finished the task, it will

signal( write); will increase semaphore write by 1, now write = 1

if now P1 wants to write it since semaphore write > 0

This proofs that, if one process is writing, no other process is allowed to write.

**CASE 2:** READING - WRITING → NOT ALLOWED. That is when one or more than one process is reading the file, then writing by another process is not allowed. Let us see that our code is working accordingly or not? Explanation:

Initial value of semaphore mutex = 1 and variable readcount = 0

Suppose two processes P0 and P1 are in a system, P0 wants to read while P1 wants to write, P0 enter first into the reader code, the moment P0 enters

Wait( mutex ); will decrease semaphore mutex by 1, now mutex = 0

Increment readcount by 1, now readcount = 1, next

if (readcount == 1)// evaluates to TRUE {

wait (write); // decrement write by 1, i.e. write = 0(which clearly proves that if one or more than one reader is reading then no writer will be allowed)

}

signal(mutex); // will increase semaphore mutex by 1, now mutex = 1 i.e. other readers are allowed to enter.

And reader continues to --READ THE FILE?

Suppose now any writer wants to enter into its code then:

As the first reader has executed wait (write); because of which write value is 0, therefore wait(writer); of the writer, code will go into an infinite loop and no writer will be allowed.

This proofs that, if one process is reading, no other process is allowed to write.

Now suppose P0 wants to stop the reading and wanted to exit then

Following sequence of instructions will take place:

wait(mutex); // decrease mutex by 1, i.e. mutex = 0

readcount --; // readcount = 0, i.e. no one is currently reading

if (readcount == 0) // evaluates TRUE{

signal (write); // increase write by one, i.e. write = 1

}

signal(mutex);// increase mutex by one, i.e. mutex = 1

Now if again any writer wants to write, it can do it now, since write > 0

**CASE 3:** WRITING -- READING → NOT ALLOWED. That is when if one process is writing into the file, then reading by another process is not allowed. Let us see that our code is working accordingly or not?Explanation:

The initial value of semaphore write = 1

Suppose two processes P0 and P1 are in a system, P0 wants to write while P1 wants to read, P0 enter first into the writer code, The moment P0 enters

Wait( write ); will decrease semaphore write by 1, now write = 0

And continue WRITE INTO THE FILE

Now suppose P1 wants to read the same time (will it be allowed?) let's see.

P1 enters reader's code

Initial value of semaphore mutex = 1 and variable readcount = 0

Wait( mutex ); will decrease semaphore mutex by 1, now mutex = 0

Increment readcount by 1, now readcount = 1, next

if (readcount == 1)// evaluates to TRUE  {

wait (write); // since value of write is already 0, hence it  will enter into an infinite loop and will not be allowed to proceed further (which clearly proves that if one writer is writing then no reader will be allowed.

}

The moment writer stops writing and willing to exit then

This proofs that, if one process is writing, no other process is allowed to read.

The moment writer stops writing and willing to exit then it will execute:

signal( write); will increase semaphore write by 1, now write = 1

if now P1 wants to read it can since semaphore write > 0

**CASE 4:** READING - READING → ALLOWED. That is when one process is reading the file, and other process or processes is willing to read, then they all are allowed i.e. reading - reading is not mutually exclusive. Let us see that our code is working accordingly or not? Explanation :

Initial value of semaphore mutex = 1 and variable readcount = 0

Suppose three processes P0, P1 and P2 are in a system, all the three processes P0, P1, and P2 want to read, let P0 enter first into the reader code, the moment P0 enters

Wait( mutex ); will decrease semaphore mutex by 1, now mutex = 0

Increment readcount by 1, now readcount = 1, next

if (readcount == 1)// evaluates to TRUE  {

 wait (write); // decrement write by 1, i.e. write = 0(which clearly proves that if one or more than one reader is reading then no writer will be allowed.

}

signal(mutex); // will increase semaphore mutex by 1, now mutex = 1 i.e. other readers are allowed to enter.

And P0 continues to --READ THE FILE?

→Now P1 wants to enter the reader code

current value of semaphore mutex = 1 and variable readcount = 1

let P1 enter into the reader code, the moment P1 enters

Wait( mutex ); will decrease semaphore mutex by 1, now mutex = 0

Increment readcount by 1, now readcount = 2, next

if (readcount == 1)// eval. to False, it will not enter if block

signal(mutex); // will increase semaphore mutex by 1, now mutex = 1 i.e. other readers are allowed to enter.

Now P0 and P1 continues to --READ THE FILE?

→Now P2 wants to enter the reader code

current value of semaphore mutex = 1 and variable readcount = 2

let P2 enter into the reader code, The moment P2 enters

Wait( mutex ); will decrease semaphore mutex by 1, now mutex = 0

Increment readcount by 1, now readcount = 3, next

if (readcount == 1)// eval. to False, it will not enter if block

signal(mutex); // will increase semaphore mutex by 1, now mutex = 1 i.e. other readers are allowed to enter.

Now P0, P1, and P2 continues to --READ THE FILE?

Suppose now any writer wants to enter into its code then:

As the first reader P0 has executed wait (write); because of which write value is 0, therefore wait(writer); of the writer, code will go into an infinite loop and no writer will be allowed.

Now suppose P0 wants to come out of system( stop reading) then

wait(mutex); //will decrease semaphore mutex by 1, now mutex = 0

readcount --; // on every exit of reader decrement readcount by

                 one i.e. readcount = 2

if (readcount == 0)// eval. to FALSE it will not enter if block

signal(mutex); // will increase semaphore mutex by 1, now mutex = 1 i.e. other readers are allowed to exit

→ Now suppose P1 wants to come out of system (stop reading) then

wait(mutex); //will decrease semaphore mutex by 1, now mutex = 0

readcount --; // on every exit of reader decrement readcount by   one i.e. readcount = 1

if (readcount == 0)// eval. to FALSE it will not enter if block

signal(mutex); // will increase semaphore mutex by 1, now mutex = 1 i.e. other readers are allowed to exit

→Now suppose P2 (last process) wants to come out of system (stop reading) then

wait(mutex); //will decrease semaphore mutex by 1, now mutex = 0

readcount --; // on every exit of reader decrement readcount by one i.e. readcount = 0

if (readcount == 0)// eval. to TRUE it will enter into if block  {

signal (write); // will increment semaphore write by one, i.e.  now write = 1, since P2 was the last process which was reading, since now it is going out, so by making write = 1 it is allowing the writer to write now.

}

signal(mutex); // will increase semaphore mutex by 1, now mutex = 1

The above explanation proves that if one or more than one processes are willing to read simultaneously then they are allowed.

Final answer

Consider a situation where we have a file shared between many people.

* If one of the people tries editing the file, no other person should be reading or writing at the same time, otherwise changes will not be visible to him/her.
* However if some person is reading the file, then others may read it at the same time.

5.

a.

semaphore ok\_to\_cross = 1;

void enter\_bridge() {

P(ok\_to\_cross);

}

void exit\_bridge() {

V(ok\_ to\_cross);

}

b.

monitor bridge {

int num\_waiting\_north = 0;

int num\_waiting\_south = 0;

int on\_bridge = 0;

condition ok\_to\_cross;

int prev = 0;

void enter\_bridge\_north() {

num\_waiting\_north++;

while (on\_bridge ||(prev == 0 && num\_waiting\_south > 0))

ok\_to\_cross.wait();

on\_bridge=1;

num\_waiting\_north--;

prev = 0;

}

void exit\_bridge\_north() {

on\_bridge = 0;

ok\_to\_cross.broadcast();

}

void enter\_bridge\_south() {

num\_waiting\_south++;

while (on\_bridge ||(prev == 1 && num\_waiting\_north > 0))

ok\_to\_cross.wait();

on\_bridge=1;

num\_waiting\_south--;

prev = 1;

}

void exit\_bridge\_south() {

on\_bridge = 0;

ok\_to\_cross.broadcast();

}

}

c)

monitor bridge {

int num\_waiting\_north = 0;

int num\_waiting\_south = 0;

int on\_bridge = 0;

condition ok\_to\_cross;

int prev = 0;

void enter\_bridge\_north() {

num\_waiting\_north++;

while (on\_bridge ||(prev == 0 && num\_waiting\_south > 0))

ok\_to\_cross.wait();

on\_bridge=1;

num\_waiting\_north--;

prev = 0;

}

void exit\_bridge\_north() {

on\_bridge = 0;

ok\_to\_cross.broadcast();

}

void enter\_bridge\_south() {

num\_waiting\_south++;

while (on\_bridge ||(prev == 1 && num\_waiting\_north > 0))

ok\_to\_cross.wait();

on\_bridge=1;

num\_waiting\_south--;

prev = 1;

}

void exit\_bridge\_south() {

on\_bridge = 0;

ok\_to\_cross.broadcast();

}

}

6

Each money operates in a seperate thread and executes the given code -->

typedef enum {EAST, WEST} Destination ;

void monkey(int id, Destination dest) {

WaitUntilSafeToCross(dest);

CrossRavine(id, dest);

DoneWithCrossing(dest);

}

Part A -->

int monkey\_count[2] = {0,0};

/\* monkey counter for each direction \*/

semaphore mutex[2] = {1,1};

/\* mutual exclusion for each direction\*/

semaphore max\_on\_rope = 5;

/\* ensure maximum 5 monkey on the rope\*/

semaphore rope = 1;

/\* ensure monkey on the rope is heading the same direction \*/

void WaitUntilSafeToCross(Destination dest) {

wait(mutex[dest]); monkey\_count[dest]++;

if (monkey\_count[dest] == 1) //is the first monkey in line,

waiting to acquire the rope wait(rope);

signal(mutex[dest]);

wait(max\_on\_rope); }

void DoneWithCrossing(Destination dest) {

wait(mutex[dest]);

signal(max\_on\_rope);

monkey\_count[dest]--;

if (monkey\_count[dest] == 0) //is the last monkey, release the rope

signal(rope);

signal(mutex[dest]);

}

Atmost 5 monkeys simltaneously execute CrossRavine()

All monkeys executing in CrossRavine are heading in the same direction

Part B -->

int monkey\_count[2] = {0,0};

/\*monkey counter for each direction \*/

semaphore mutex[2] = {1,1};

/mutual exclusion for each direction/

semaphore max\_on\_rope = 5;

/ensure maximum 5 monkey on the rope/

semaphore rope = 1;

/\*ensure monkey on the rope is heading the same direction \*/

semaphore order = 1;

/\*Use to prevent starvation \*/

void WaitUntilSafeToCross(Destination dest) {

wait(order);

wait(mutex[dest]);

monkey\_count[dest]++;

if (monkey\_count[dest] == 1) //is the first monkey in line,

waiting to acquire the rope wait(rope);

signal(mutex[dest]);

signal(order);

wait(max\_on\_rope); }

void DoneWithCrossing(Destination dest) {

wait(mutex[dest]);

signal(max\_on\_rope);

monkey\_count[dest]--;

if (monkey\_count[dest] == 0) //is the last monkey,

release the rope signal(rope);

signal(mutex[dest]);

}