Name - SAURAV SINGH
Roll No – 70
Section – K18SB
GitHub Link -

Q1 Consider the dining philosophers problem. If we place all five chopsticks in the center of the table, how could we use semaphores to implement the philosophers eat() method? Be sure to declare and initialize all variables you use. Your solution should be as efficient as possible. You should define two methods begin() and end().
SOLUTION

```
#include <pthread.h>
#include <semaphore.h>
#include <stdio.h>

#define N 5
#define THINKING 2
#define HUNGRY 1
#define EATING 0
#define LEFT (phnum + 4) % N
#define RIGHT (phnum + 1) % N

int state[N];
int phil[N] = { 0, 1, 2, 3, 4 };

sem_t mutex;
sem_t S[N];

void eat(int phnum)
{
        if (state[phnum] == HUNGRY
                && state[LEFT] != EATING
                && state[RIGHT] != EATING) {

                state[phnum] = EATING;

                sleep(2);

                printf("Philosopher %d takes fork %d and %d\n",
                                    phnum + 1, LEFT + 1, phnum + 1);

                printf("Philosopher %d is Eating\n", phnum + 1);

                sem_post(&S[phnum]);
        }
}

void begin(int phnum)
{

        sem_wait(&mutex);


        state[phnum] = HUNGRY;

        printf("Philosopher %d is Hungry\n", phnum + 1);
        eat(phnum);

        sem_post(&mutex);


        sem_wait(&S[phnum]);

        sleep(1);
}


void end(int phnum)
{

        sem_wait(&mutex);


        state[phnum] = THINKING;

        printf("Philosopher %d putting fork %d and %d down\n",
                phnum + 1, LEFT + 1, phnum + 1);
        printf("Philosopher %d is thinking\n", phnum + 1);

        eat(LEFT);
        eat(RIGHT);

        sem_post(&mutex);
}

void* philospher(void* num)
{

        while (1) {

                int* i = num;

                sleep(1);

                begin(*i);

                sleep(0);

                end(*i);
        }
}

int main()
{

        int i;
        pthread_t thread_id[N];


        sem_init(&mutex, 0, 1);

        for (i = 0; i < N; i++)
```

```
                sem_init(&S[i], 0, 0);

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


                pthread_create(&thread_id[i], NULL,
                                        philospher, &phil[i]);

                printf("Philosopher %d is thinking\n", i + 1);
        }

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

                pthread_join(thread_id[i], NULL);
}
```

****
****

Q2. WAP to create 3 threads and 3 binary semaphores S1, S2 and S3 that are initialized with value 1. Threads should acquire semaphore in such a manner that deadlock is achieved.
SOLUTION:

```
#include<stdio.h>
#include<pthread.h> #include<semaphore.h>
void *fun1(); void *fun2(); void *fun3(); sem_t s1,s2,s3;
int x = 1; int main()
{
printf("Initial Value %d\n",x); sem_init(&s1,3,1); sem_init(&s2,3,1); sem_init(&s3,3,1);
pthread_t t1,t2,t3; pthread_create(&t1,NULL,fun1,NULL);
pthread_create(&t2,NULL,fun2,NULL); pthread_create(&t3,NULL,fun3,NULL);
pthread_join(t1,NULL); pthread_join(t2,NULL); pthread_join(t3,NULL); printf("%d\n",x);
}
void *fun1()
{
int x1;
sem_wait(&s1); sem_wait(&s2); sem_wait(&s3);
x1 = x; x1++; x = x1; printf("%d\n",x); sem_post(&s1); sem_post(&s2); sem_post(&s3);
}
void *fun2()
{
int x2;
sem_wait(&s2); sem_wait(&s3); sem_wait(&s1);
x2 = x; x2--; x = x2; printf("%d\n",x); sem_post(&s2); sem_post(&s3); sem_post(&s1);
}
void *fun3()
{
int x3;
sem_wait(&s2); sem_wait(&s1); sem_wait(&s3);
x3 = x;
x3++; x = x3; printf("%d\n",x); sem_post(&s2); sem_post(&s1); sem_post(&s3);
}
```

****
****

Q3. Consider the following three threads and assume x and y are initially 0. Describe the race conditions in terms of these threads. Be specific. Using semaphore(s), initialized appropriately, add code to insure that thread T1 prints the value 3. You may assume the semaphores are initialized before any threads begin. Thread T1: print(x+y); Thread T2: x=1; Thread T3: y=2;

SOLUTION:

```
#include   <string>
#include  <vector>
#include
<iostream>
#include <mutex>

#include
"exceptionhelper.h"
#include <exception>



namespace com::saurav::mock::Thread

{


          class ThreeThreadComunication : public Runnable

          {

          public:

               AtomicInteger
               *counter;
               std::vector<int>  array;
               static        std::mutex
               mutex;


               virtual ~ThreeThreadComunication()

               {

                    delete counter;

               }


               ThreeThreadComunication(std::vector<int> &array, AtomicInteger
*counter);


                                void run() override;

                      };


}

namespace com::Saurabh::mock::Thread

{




           class ThreeThreadComunicationTest

           {

                      static void main(std::vector<std::wstring> &args);


           };


}


class InterruptedException : public std::exception

{

private:
   std::string
   msg;


public:

   InterruptedException(const std::string& message = "") : msg(message)

   {

   }


   const char * what() const throw()

   {

      return msg.c_str();
   }

};

#include "snippet.h"



namespace com::saurabh::mock::Thread

{



          ThreeThreadComunication::ThreeThreadComunication(std::vector<int>
&array, AtomicInteger *counter)

          {

                    this->counter           =
                    counter;  this->array  =
                    array;

          }



          void ThreeThreadComunication::run()

          {

                    int i = 0;

                    while (i < array.size())

                    {

                              {
                                        std::scoped_lock<std::mutex> lock(mutex);

                                        if (static_cast<Integer>(Thread::currentThread().getName()) ==


counter-
>get())                                 {

                                                  std::wcout    <<    array[i]    <<
                                                  std::endl;  if  (counter->get()  ==
                                                  3)
                                        {

                                                counter->getAndSet(1);

                                        }

                                        els
                                        e

                                        {
                                                int  c  =  counter->get();
                                                counter-
                                                >getAndSet(++c);


                                        }
                                        i++
                                        ;

                                }



                                mutex.notifyAll()
                                ; try

                                {

                                        mutex.wait();

                                    }

                                catch (const InterruptedException &e)

                                {

                                        e->printStackTrace();

                                }

                        }

                }

        }

}

namespace com::sourabh::mock::Thread

{



        void ThreeThreadComunicationTest::main(std::vector<std::wstring> &args)

        {
```

```
AtomicInteger *counter = new
AtomicInteger(1); std::vector<int> array1 = {1,
4, 7};

std::vector<int> array2 = {2, 5, 8};

std::vector<int> array3 = {3, 6, 9};



ThreeThreadComunication *obj1 = new ThreeThreadComunication(array1,
counter); ThreeThreadComunication *obj2 = new
ThreeThreadComunication(array2, counter); ThreeThreadComunication *obj3
= new ThreeThreadComunication(array3, counter);



Thread *t1 = new Thread(obj1,
L"1"); Thread *t2 = new
Thread(obj2, L"2"); Thread *t3 =
new Thread(obj3, L"3");



t1->start();

t2->start();

t3->start();



delete
t3;
delete
t2;
```

```
            delete t1;
      }
}
```

****
****

Q4. WAP to implement Producer Consumer problem such that Maximum number of items produced by a producer are five. Producer should not produce any item if the buffer is full and should say

“BUFFER IS ALREADY FULL” and the consumer should not consume if the buffer is empty. Problem

should be implemented with the help of thread, semaphore and mutex lock.

Solution:

```
#include<stdio.h>
#include<pthread.h>
#include<semaphore.h>
void *producers();
void *consumers();
int buffer[5];
int i=0,j=0,data;
int x=0;
sem_t mutex,full,empty;
pthread_t p[5],c[5];
int main()
{
sem_init(&mutex,0,1);
sem_init(&full,0,0);
sem_init(&empty,0,5);
pthread_t t,t1;
pthread_create(&t,NULL,consumers,NULL);
sleep(5);
for(int s=0;s<5;s++)
{
pthread_create(&p[s],NULL,producers,NULL);
sleep(5);
pthread_create(&c[s],NULL,consumers,NULL);
sleep(5);
}
pthread_join(t,NULL);
for(int s = 0;s<5;s++)
{
pthread_join(p[s],NULL);
pthread_join(c[s],NULL);
}
pthread_create(&t1,NULL,producers,NULL);
pthread_join(t1,NULL);
}
void *producers()
{
printf("\n Producers code\n");
if(x==5)
{
printf("BUFFER IS ALREADY FULL\n");
}
else
{
sem_wait(&mutex);
sem_wait(&empty);
printf("Enter the data to be produces\n");
scanf("%d",&data);
buffer[x]=data;
x++;
sem_post(&full);
sem_post(&mutex);
printf("Producer leaving the thread\n");
}
}
void *consumers()
{
printf("\n Consumer Code\n");
printf("Consumer thread enters\n");
if(x==0)
{
printf("NO data to consume\n");
}
else
{
sem_wait(&full);
sem_wait(&mutex);
data = buffer[i];
buffer[i]=0;
i++;
printf("%d\n",i);
sem_post(&mutex);
sem_post(&empty);
printf("Leaving Consumer thread\n ");
}
}
```