Assignment 4

Name: Ashutosh Soni
ID: 2018ucp1505

Q1.Using Thread.

```
#include<bits/stdc++.h>
#include<pthread.h>
#include<semaphore.h>
#include <unistd.h>
using namespace std;
int d_produce=0,d_consume=0;
sem_t not_Empty;
void* produce(void *arg){
  while(1){
        cout<<"Producer appending Data"<<"\n";
        cout<<"Total Data produced = "<<++d_produce<<"\n";
        sem_post(&not_Empty);
  }
}
void* consume(void *arg){
  while(1){
        sem_wait(&not_Empty);
        cout<<"Consumer taking Data"<<"\n";
        cout<<"Total Data consumed = "<<(--d_consume)*-1<<"\n";
  }
}

int main(int argv,char *argc[]){
   int p;
  pthread_t Producer,Consumer;
  pthread_attr_t attr;
```

```
   sem_init(&not_Empty,0,0);
   pthread_attr_init(&attr);
   pthread_attr_setdetachstate(&attr,PTHREAD_CREATE_JOINABLE);
   p=pthread_create(&Producer,&attr,produce,NULL);
   if(p){
           cout<<"Error in creating thread"<<"\n";
           exit(-1);
   }
   p=pthread_create(&Consumer,&attr,consume,NULL);
   if(p){
           cout<<"Error in creating thread"<<"\n";
           exit(-1);
   }
   pthread_attr_destroy(&attr);
   p=pthread_join(Producer,NULL);
   if(p){
           cout<<"Error in joining thread"<<"\n";
           exit(-1);
   }
   p=pthread_join(Consumer,NULL);
   if(p){
           cout<<"Error in joining thread"<<"\n";
           exit(-1);
   }
   pthread_exit(NULL);

   return 0;
}
```

2. Solutions for Readers and Writers problem.

Using Moniters:-

```
#include<bits/stdc++.h>
#include<pthread.h>
#include <unistd.h>
using namespace std;
pthread_mutex_t Mutex;
int con=0, n, data=1,srd=0;
class monitor
{
    public:
        int readers, writers, bk_writer, bk_reader;
        pthread_cond_t OKtoRead, OKtoWrite;
        //pthread_cond_init(&notZero,NULL);
            void StartRead(int id){
            if(writers!=0 || bk_writer!=0){

                bk_reader++;
                cout<<("Reader %d blocked\n",id);
                pthread_cond_wait(&OKtoRead, &Mutex);

            }
            readers++;
            pthread_cond_signal(&OKtoRead);
        }

        void EndRead(int id){
            readers--;
            cout<<("Reader %d leaving database\n",id);
            if(readers==0){
```

```
            pthread_cond_signal(&OKtoWrite);
        }

    }

    void StartWrite(int id){
        if(writers!=0 || readers!=0){
            bk_writer++;
            cout<<("Writer %d blocked\n",id);
            pthread_cond_wait(&OKtoWrite, &Mutex);
        }
        writers++;
    }

    void EndWrite(int id){
        writers--;
        cout<<("Writer %d leaving database\n",id);
        if(bk_reader==0){
            pthread_cond_signal(&OKtoWrite);
        }
        else{
            pthread_cond_signal(&OKtoRead);
        }

    }
    monitor(){
        readers=0, writers=0, bk_writer=0, bk_reader=0;
        pthread_cond_init(&OKtoRead, NULL);
        pthread_cond_init(&OKtoWrite, NULL);
    }
};
monitor obj;
int reader_id=1 ,writer_id=1;
```

```
void *readerFunc(void *arg)
{
        while(1){

                obj.StartRead(reader_id);
                cout<<("***** Reader %d reading database
*****\n\n",reader_id);
                usleep(1000000);
                obj.EndRead(reader_id);
                usleep(1000000);
                reader_id++;
        }
}

void *writerFunc(void *arg)
{
        while(1)
        {
                //usleep(1000000);
                obj.StartWrite(writer_id);
                cout<<("<<<<<< Writer %d writing database >>>>>>
\n\n",writer_id);
                usleep(1000000);
                obj.EndWrite(writer_id);
                usleep(1000000);
                writer_id++;
        }
}

int main(){
        pthread_t reader, writer;
        pthread_mutex_init(&Mutex,NULL);
```

```
    pthread_create(&reader, NULL, &readerFunc, NULL);
    pthread_create(&writer, NULL, &writerFunc, NULL);
    pthread_exit(NULL);
    pthread_mutex_destroy(&Mutex);
    return 0;
}
```

3. solutions for Dining Philosopher.

Using Moniters:-

```
#include <bits/stdc++.h>
#include <unistd.h>
#include <pthread.h>
#include <ctype.h>
#include <string.h>
#include <semaphore.h>

#define N 5
#define THINK 0
#define HUNGRY 1
#define EAT 2
#define LEFT (i+N-1)%N
#define RIGHT (i+1)%N

void initialization();
void test(int i);
void take_chopsticks(int i);
void put_chopsticks(int i);

//pthread_mutex_t lock;

void *philosopher(void *i)
```

```
{
  while(1)
  {
     //variable representing philosopher
     int self = *(int *) i;
     int j,k;
     j = rand();
     j = j % 11;
     printf("\nPhilosopher %d is thinking for %d seconds\n",self,j);
     usleep(j);
     //philosopher take chopsticks
     take_chopsticks(self);
     k = rand();
     k = k % 4;
     printf("\nPhilosopher %d is eating for %d seconds\n",self,k);
     usleep(k);
     //philosopher release chopsticks
     put_chopsticks(self);
  }

}
sem_t mutex;
sem_t next;
//count varaible for philosophers waiting on semaphore next
int next_count = 0;

//implementing condition variable using semaphore
//semaphore and integer variable replacing condition variable
typedef struct
{
  sem_t sem;
  //count variable for philosophers waiting on condition semaphore sem
  int count;
```

```
}condition;
condition x[N];

//state of each philosopher(THINKING, HUNGRY or EATING)
int state[N];

//turn variable corresonding to each chopstick
//if philosopher i wants to each the turn[i] and turn[LEFT]must be set to i
int turn[N];

//wait on condition
void wait(int i)
{
  x[i].count++;
  if(next_count > 0)
  {
     //signal semaphore next
     sem_post(&next);
  }
  else
  {
     //signal semaphore mutex
     sem_post(&mutex);
  }
  sem_wait(&x[i].sem);
  x[i].count--;
  //  printf("\nX.count -> %d",x.count);
}

//signal on condition
void signal(int i)
{
  if(x[i].count > 0)
  {
      next_count++;
      //signal semaphore x[i].sem
      sem_post(&x[i].sem);
      //wait semeaphore next
      sem_wait(&next);
      next_count--;
  }
}
void test(int i)
{
  if(state[i] == HUNGRY && state[LEFT] != EAT && state[RIGHT] != EAT &&
turn[i] == i && turn[LEFT] == i)
  {
      state[i] = EAT;

      //signal on condition
      signal(i);

      /*  printf("\nNext Count -> %d, X_count -> %d,state[%d] ->
%d,state[%d] -> %d,state[%d] -> %d",

next_count,x[i].count,i,state[i],LEFT,state[LEFT],RIGHT,state[RIGHT]);*/

  }
}

void take_chopsticks(int i)
{
  //wait semaphore mutex
  sem_wait(&mutex);
  state[i] = HUNGRY;
  test(i);
  while(state[i] == HUNGRY)
  {
     //printf("\nThread %d is waiting on condition",i);

     //wait on condition
     wait(i);
  }
  if(next_count > 0)
  {
     //signal semaphore next
     sem_post(&next);
  }
  else
  {
     //signal semaphore mutex
     sem_post(&mutex);
  }
}

void put_chopsticks(int i)
{
  //wait semaphore mutex
  sem_wait(&mutex);
  state[i] = THINK;
  //set turn variable pointing to LEFT and RIGHT philosophers
  turn[i] = RIGHT;
  turn[LEFT] = LEFT;

  test(LEFT);
  test(RIGHT);

  if(next_count > 0)
  {
    //signal semaphore next
    sem_post(&next);
  }
  else
  {
    //signal semaphore mutex
    sem_post(&mutex);
  }
}

void initialization()
{
  int i;
  sem_init(&mutex,0,1);
  sem_init(&next,0,0);
  for(i = 0;i < N;i++)
  {
    state[i] = THINK;
    sem_init(&x[i].sem,0,0);
    x[i].count = 0;
    turn[i] = i;
  }
  //setting turn variables such that Philosophers 0,2 or 4 can grab both
chopsticks initially
  turn[1] = 2;
  turn[3] = 4;
  turn[6] = 0;

}

int main()
{
  int i, pos[N];
  //one thread corresponding to each philosopher
  pthread_t thread[N];
  pthread_attr_t attr;

  //initilize semaphore and other variables
  initialization();

  pthread_attr_init(&attr);

  for (i = 0; i < N; i++)
  {
     pos[i] = i;
     //create thread corresponding to each philosopher
     pthread_create(&thread[i], NULL,philosopher, (int *) &pos[i]);
  }
  for (i = 0; i < N; i++)
  {
     pthread_join(thread[i], NULL);
  }

  return 0;
}
```