Report

Overview :

There are 2 files for this question, one is the header file and the other is the runner file

- header.h
- q1.c

The program can be run as follows :

```
gcc -pthread q1.c -o q1
./q1
```

Explanation for q1 :

1. header.h

In header.h we declare all the structs and thread arrays we would need to simulate the tutorials.

I have used 4 structs mainly courses, students, labs, TAs.

These structures contain the variables that are used later in the program.

- Courses :

```
struct courses {
    int course_id;                      // course id
    char name_course[SIZE];             // name of the course
    float interest_quot;                // interest of that course
    int course_max;                     // max students that can potentially be allocated by a TA
    int no_of_labs;                     // the number of labs for that course
    int lab_list[SIZE];                 // the list of labs for that course
    int is_avail;                       // 0 implies course is there, 1 implies its ded
    int tot_courses;                    // total courses isnt global so
};
```

- Students :

```
struct students {
    int stud_id;                        // student id
    float calibre;                      // calibre of the student
    int pri[3];                         // list of priority courses.
    int curr_pri;                       // current priority course that studnet is aiming for, 0 initially
    int reg_time;                       // sleeptime
    int is_avail;                       // shows the availability of the student, 1 when student is asleep or in tut.
    int assigned_course;                // allocated course number, -1 if nothing is allocated.
    int tot_studs;                      // total students isnt global so
    int is_done;                        // extra check for when the student leaves simulation
};
```

- Labs :

```
struct labs {
    int lab_id;                         // ID of the lab.
    char name_lab[SIZE];                // Lab name
    int no_of_TAs;                      // number of TAs in this lab
    int max_allocation_TA;              // max times a TA from this lab can TA a course.
    int tot_labs;                       // total labs isnt global so
    int is_done;                        // extra check, 1 when the lab leaves, 0 otherwise
};
```

- TAs :

```
struct TAs {
    int TA_id;                      // TA id of that Lab.
    int lab_id;                     // the lab id form which the TA is from.
    int tot_TAs;                    // total TAs isnt global so
    int is_avail;                   // to see if the TA is available or taking a tut.
    int numdone;                    // number of TAships done
};
```

The comments describe each of the structures.

I have used 2 main functions, one for students and one for courses :

```
void * handlestudents(void * arg);
void * handlecourses(void * arg);
```

The thread arrays and mutex arrays and conditional var are as follows :

```
#define SIZE 200
pthread_cond_t cond[SIZE];

pthread_t coursethreads[SIZE];
pthread_t studentsthreads[SIZE];
pthread_t labsthreads[SIZE];

pthread_mutex_t studmutex[SIZE];
pthread_mutex_t coursemutex[SIZE];
pthread_mutex_t labmutex[SIZE];
pthread_mutex_t TAmutex[SIZE][SIZE];
```

2. q1.c

we take the input from the user in int main() and spawn all the threads to their corresponding handler functions and wait until all the student threads have exited.

```
for(int i=0;i<num_of_studs;i++) {
    pthread_create(&studentsthreads[i],NULL,handlestudents,studlist + i);
    pthread_mutex_init(studmutex+i,NULL);
}
for(int i=0;i<num_of_courses;i++) {
    pthread_create(&coursethreads[i],NULL,handlecourses,courselist + i);
    pthread_mutex_init(coursemutex+i,NULL);
}
```

Handler functions are as follows :

2.1 handlestudents

In this function we initally put all the student threads to a sleep that is determined by the input for each student. And make the student available after the sleep.

```
sleep(student->reg_time);
student->is_avail=1;
```

after the sleep, the student is now available and would have filled the course registration form for which we use a printf to notify.

Now, we put all the student threads in a while loop, untill their priority courses have been exhausted. Initially we put each of the students in a cond wait until the course that they are aiming for has been removed, if the course that the student is aiming for has been removed, the student has to change priority.

```
if(courselist[student->pri[student->curr_pri]].is_avail==0)
        pthread_cond_wait(cond+student->stud_id,studmutex+student->stud_id);
```

Now after the cond wait, we have 2 cases, either the student has been allocated his priority course, or the course that he was aiming for has been removed. We can handle both these cases separately.

2.1.1 Course has been removed/Student has not been allocated a course :

in this case, all we need to do is to increase the priority of the student to his next priority or to remove the student if his last priority course has been removed. We also need to unlock the student at the end as when the control comes back to cond wait, the lock of the student is acquired.

```
if(student->curr_pri==2) {
    student->is_avail=0;
    student->is_done=1;
    printf(HRED"Student %d got no course :(.\n",student->stud_id);
    printf(HRED "Student %d exited simulation.\n",student->stud_id);
    return NULL;
}
printf(YELLOW "Student %d has changed his current preference from %s (priority %d) to %s (priority %d)\n",student->stud_id,courselist[stude
student->curr_pri++;
student->is_avail=1;
pthread_mutex_unlock(studmutex+student->stud_id);
```

2.1.2 Student has been allocated a course :

We can again encounter 2 cases here, if the student likes the course he has been allocated, he will take it or he will withdraw from that course. This is determined by the formula given in the pdf. And can be simulated by a random number generator.

2.1.2.1 Student takes the course :

We will just print what the student took finally and exit him from the simulation.

```
int val = 100*(student->calibre*courselist[c_id].interest_quot);
int check_val = rng(1,100);
if(check_val <= val) {
    // accepted
    printf(YELLOW "Student %d has selected the course %s permanently\n",student->stud_id,courselist[c_id].name_course);
    student->is_avail=0;
    student->is_done=1;
    printf(HRED "Student %d got %s :) %d priority.\n",student->stud_id,courselist[student->pri[student->curr_pri]].name_course,student->cur
    printf(HRED "Student %d exited simulation.\n",student->stud_id);
    pthread_mutex_unlock(studmutex+student->stud_id);
    return NULL;
}
```

2.1.2.2 Student withdraws from the course :

We will change the student priority or exit him from the simulation if he has exhausted his final priority course.

```
printf(YELLOW "Student %d has withdrawn from course %s\n",student->stud_id,courselist[c_id].name_course);
if(student->curr_pri==2) {
    student->is_avail=0;
    student->is_done=1;
    printf(HRED"Student %d got no course :(.\n",student->stud_id);
    printf(HRED "Student %d exited simulation.\n",student->stud_id);
    return NULL;
}
printf(YELLOW "Student %d has changed his current preference from %s (priority %d) to %s (priority %d)\n",student->stud_id,courselist[stude
student->curr_pri++;
student->is_avail=1;
student->assigned_course=-1;
pthread_mutex_unlock(studmutex+student->stud_id);
```

2.2 handlecourses

in our courses function, the first thing we need to check is if the course is available for the takings or not. This can be easily done by looping through all the TAs of all the labs of this course and checking if they have already finished their max TAship or not.

2.2.1 Removal of course check :

the check for TAs is done as follows :

```
int lablist[SIZE];
for(int i=0;i<course->no_of_labs;i++) {
    lablist[i]=course->lab_list[i];
}
int flag=0;
for(int i=0;i<course->no_of_labs;i++) {
    int labno = lablist[i];
    for(int j=0;j<labs_list[labno].no_of_TAs;j++) {
        if(TA_list[labno][j].numdone<labs_list[labno].max_allocation_TA) {
            flag=1;
        }
    }
}
```

if the flag is still 0, that means all the TAs have exhausted their TAship else we can proceed to rest of the code and try allocating a TA for this course for a tut.

flag = 0 case, we need to conditionally signal all the students who are waiting on this particular course too, this is an important step, as when the course is removed, the students with current priority course as this course have to change courses to their next priority too.

```
if(!flag) {
    pthread_mutex_lock(coursemutex+course->course_id);
    printf(HRED "Course %s does not have any TA mentors eligible and is removed from course offerings\n",course->name_course);
    course->is_avail=1;
    printf(HRED "Course %s has exited simulation.\n",course->name_course);
    for(int i=0;i<studlist[0].tot_studs;i++) {
        if(studlist[i].pri[studlist[i].curr_pri]==course->course_id) {
            pthread_cond_signal(cond+i);
        }
    }
    pthread_mutex_unlock(coursemutex+course->course_id);
    return NULL;
}
```

To ensure course removal is an atomic operation for that particular course, we acquire and release the lock of the course.

2.2.2 Taking tutorials :

If the course is available, our next step is to find a TA for the course to and make him take a tut with a certain capacity of students.

2.2.2.1 Acquiring TA :

We make the TA status unavailable as he/she needs to take a tut now, and we increase his/hers numdone count (it represents num of TAships done) .

```
for(int i=0;i<course->no_of_labs;i++) {
    int labno = lablist[i];
    for(int j=0;j<labs_list[labno].no_of_TAs;j++) {
        if(TA_list[labno][j].is_avail==0 && TA_list[labno][j].numdone<labs_list[labno].max_allocation_TA) {
            // assign this TA to the course
            pthread_mutex_lock(&TAmutex[labno][j]);
            TA_list[labno][j].is_avail=1;
            TA_list[labno][j].numdone++;
```

2.2.2.2 Allocating students for the tutorial :

We decide on D by random allocation and check for students such that we do not have more than D students for the tutorial. The allocation can be done in 2 ways as mentioned, I have implemented for the case where tutorial can be taken even for 0 students to avoid busy waiting condition. This can however easily be modified to the case where at least one student is reuqired for the tutorial by just un commenting the while loop.

```
// while(stu==0) {
      int cid = course->course_id;
      for(int i=0;i<studlist[0].tot_studs;i++) {
          if(stu < D && studlist[i].is_done==0) {
              if(studlist[i].is_avail==1 && studlist[i].assigned_course!=cid && studlist[i].pri[studlist[i].curr_pri]==cid) {
                  pthread_mutex_lock(studmutex+i);
                  printf(GREENBG"Student %d has been allocated a seat in course %s\n",i,course->name_course);
                  studlist[i].assigned_course=cid;
                  stu++;
                  studarr[start]=i;
                  start++;
              }
          }
      }
  //     sleep(1);
  // }
```

We acquire the locks of all the students that have been allocated the tutorial and save them in an array called studarr.

2.2.2.3 Completion of the tutorial :

When the tutorial has been completed, we need to signal all the students who have taken the tutorial too and change the status of the TA who took the tutorial to active.

```
printf(GREENBG"Tutorial has started for course %s with %d seats filled out of %d\n",course->name_course,stu,D);
sleep(3);             // change to sleep(1) if the tuts need to happen when atleast 1 student is to be alloted.
printf("TA %d from lab %s has completed the tutorial for course %s\n",j,labs_list[labno].name_lab,course->name_course);
for(int i=0;i<start;i++) {
   studlist[studarr[i]].is_avail=0;
   pthread_mutex_unlock(&studmutex[studarr[i]]);
   pthread_cond_signal(cond+studarr[i]);
}
TA_list[labno][j].is_avail=0;
```

the sleep(3) can be changed to sleep(1) if we are going with the case of atleast 1 student required for the tutorial.

2.2.2.4 Lab removal check :

As the TA who just conducted the tut has increased his numdone variable, it is possible that all the TAs of a Lab have exhausted their TAship limit. Thus we need to check for the same.

```
int fl=0;
for(int j=0;j<labs_list[labno].no_of_TAs;j++) {
    if(TA_list[labno][j].numdone<labs_list[labno].max_allocation_TA && labs_list[labno].is_done==0) {
        fl=1; break;
    }
}
pthread_mutex_lock(labmutex+labs_list[labno].lab_id);
if(fl==0) {
    if(labs_list[labno].is_done==0) {
        printf(HRED"Lab %s no longer has students available for TA ship.\n",labs_list[labno].name_lab);
        labs_list[labno].is_done=1;
    }
}
pthread_mutex_unlock(labmutex+labs_list[labno].lab_id);
pthread_mutex_unlock(&TAmutex[labno][j]);
```

We unlock the TA and the labs when everything is done and we do not have anything else to do with the TA/lab.

Busy waiting for student thread has been taken care of with the use of conditional waits and conditional signals.