Operating Systems and Networks

Assignment 5 - Question 1

Name: Akshett Rai Jindal

Roll Number: 2019114001

Contents

An Alternative Course Allocation Portal

1) Assumptions

- The TAs can start the tutorial even when none of the seats are filled. Appropriate delays have been added for this.

2) Entities

2.1) Course

Each course is stored as a `struct` and has the following structure:

```c
typedef struct Course
{
    llint id;                           /* The id of the course (0 indexed) */
    char name[COURSE_NAME_LEN+1];       /* Name of the course */
    double interest_quotient;           /* Interest Quotient of the course */
    llint tut_slots_limit;              /* The maximum slots that can be allotted for a tut */
    llint num_labs;                     /* Number of labs from which it will take TAs */
    llint *lab_ids;                     /* The lab ids */

    /*
        * Lock that will be used to update the critical values:
        *   1. is_open
        *   2. tut_slots
    */
    pthread_mutex_t lock;
```

```
    pthread_cond_t tut_slots_cond;          /* The cond var on which students will wait for getting tut slots */
    pthread_cond_t tut_session_cond;        /* The cond var on which students will wait for tut to end */
    llint tut_slots;                        /* The current available slots in the tutorial */
    Mentor *ta;                             /* The current TA of the course OR NULL */
    Lab *ta_lab;                            /* The lab of the current TA OR NULL */
    bool is_open;                           /* Holds if the course is still open for registration */

} Course;
```

2.2) Students

Each student is also stored as a `struct` which has following structure:

```
typedef struct Student
{
    llint id;                                               /* The id of the student */
    double calibre_quotient;                                /* The calibre quotient of the student */
    llint preferences[STUDENT_NUM_PREFERENCES];             /* The course ids of the students's pref. */
    llint preferences_filling_time;                         /* The time student will take to fill pref. */

    /*
        * Function that returns true if student will withdraw from
        * the current preference after attending its tutorial.
        * Uses the rand() function and compares the random prob.
        * with the value student->calibre_quotient * course->interest_quotient
    */
    bool (*withdraw_from_course)(struct Student*, Course*);

}
Student;
```

2.3) Labs

Each lab is also stored as a `struct` which has following structure:

```
typedef struct Lab
{
    llint id;                           /* ID of the lab */
    char name[LAB_NAME_LEN];            /* Name of the lab */
    llint num_mentors;                  /* The number of mentors in the lab */
    llint taship_limit;                 /* The max time a mentor can become a TA */

    Mentor **mentors;                   /* Array storing the pointers to the mentors */

    /*
        * Lock to access and modify critical variables:
            * 1. curr_max_taship
            * 2. num_mentors_wo_max_taship
    */
    pthread_mutex_t lock;

    /* ================ For Bonus ============== */

    /*
        * Initialized with 1
        * Only the mentors who have become TA less than this much
        * times can be assigned to a course as a TA.
    */
    llint curr_max_taship;

    /*
        * Initialized with num_mentors
        * Stores the number of mentors who have not
        * done curr_max_taship number of TAships
        * When it reaches 0, is reset to num_mentors
```

```
         * and curr_max_taship is increased by one
    */
    llint num_mentors_wo_max_taship;

    /* ======================================== */

    bool lab_mentors_available;         /* Becomes false when all mentors reach the TAship limit */
}
Lab;
```

2.4) Mentors

Each Mentor is also stored as a `struct` which has following structure:

```
typedef struct Mentor
{
    llint id;                   /* The ID of the mentor (unique within a lab) */
    llint lab_id;               /* The Lab ID to which the mentor belongs */
    llint taships_done;         /* The number of times they have become a TA */
    pthread_mutex_t lock;       /* Lock for changing taships_done value */
}
Mentor;
```

3) Threads

3.1) Course

- Each course has its own separate thread.
- The thread functions run a while loop as long as its registration is open by checking the `course->is_open` variable.
- In this while loop, the course will keep on looking for TAs and conduct tutorials.

```
while (course->is_open)
{
    // small sleep
    // reset the values
    // find TA
    // conduct tutorial if TA found
}
```

- It sleeps for a while so that the students from the last tutorial can
- First of all the values from last round of tutorial are reset.
- A `bool can_find_ta` is also kept (initially `false`) which will be `true` if after iterating over the labs, we could not find the TAs only because they were busy in another course or due to the bonus part and not because their TAship limits were reached.

3.1.1) Finding TAs

- Then it iterates over the labs from which it can accept TAs, it checks its `lab_mentors_available` variable to confirm that it has mentors who have not reached their TAship limit.
- It then iterates over its mentors and tries to acquire its lock. If it fails to do so, it means that the mentor is being checked by another course at present or is TA in another course.
- Otherwise it checks the condition that the TA has not done as much TAships as the `course->curr_max_taship`:
    - The condition that the TA has not reached the limit set by the lab is handled by this as the lab automatically closes when the `curr_max_taship` reaches greater than the lab limit.
    - The condition that the TA is not being used by another course is also handled as the course first tries to lock the course and continues if the course who has the mentor as its TA has already locked it.

- After finding a TA, it increases the count of tashipsof the mentor by 1. It also decreases the `lab->num_mentors_wo_max_tashi` by 1 and if it reaches 0, then `lab->curr_max_tashi` is increased by 1.

If this `lab->curr_max_taship` becomes greater than the lab limit, then it means that all mentors of the lab have reached the limit of taship and are now permanently unavailable.

- If no TA has been found and `can_find_ta` is also false, then it means that the course can never have anymore TAs. So, it needs to exit from the simulation. Before exiting it broadcasts again on the `course->tut_slots_cond` so as to tell the students who were waiting for a seat to move to next preference. It also broadcasts on `course->tut_session_cond` just for safety.

3.1.2) Conducting Tutorials

- After finding the TA, random count of slots in the range `[1, course->tut_slots_limit]`.
- It then acquires the `course->lock` to update the number of available slots
- It then broadcasts on the conditional variable `course->tut_slots_cond` and unlocks the lock so that the students can take the slots.
- It sleeps for some time to wait for the students to fill the slots.
- On waking up, it again acquires the lock and changes the number of available slots to 0.
- Then it again sleeps for the tutorial duration.
- On waking up, it broadcasts on the `course->tut_session_cond` conditional variable to tell the students that the tutorial has ended.
- Then it unlocks the:
  1. `course->lock` to allow students to acquire it and wakeup from tutorial session conditional variables
  2. `ta->lock` to allow other courses to select the mentor as the TA if limit has not exceeded.

3.2) Student

- Each student has its own separate thread.
- These thread functions first sleep for the time the student takes to fill in their preferences.
- And then iterates over preferences to find a seat and decide whether to accept or withdraw.

```
sleep(student->preferences_filling_time);
for (int pref_num=0; pref_num<3; pref_num++)
{
    // try to get seat
    if (not_get_seat) continue;

    // attend tutorial
    // Decide whether to withdraw
}
```

3.2.1) Getting a Seat

- For getting a seat in the current preference, there is a while loop that runs until either the course becomes unavailable for registration or the there is at least one seat in the tutorial.
- In the while loop, it waits on the `course->tut_slots_cond` cond. var. and hence acquires the `course->lock` before entering the while loop

```
pthread_mutex_lock(&course->lock);
while (course->is_open && course->tut_slots == 0) {
    pthread_cond_wait(&course->tut_slots_cond, &course->lock);
}
```

- It wakes from the wait and breaks from the while loop if the course registration closes or there is a seat that the student can take.
- If the course registration ends then the student moves to next preference.

3.2.2) Attending Tutorial

- After getting a seat, the student starts waiting on the cond. var. `course->tut_session_cond` and then it wakes up only when the session has ended.
- After waking up, the student uses the `student->withdraw_from_course` function to decide whether to withdraw or accept the course.