Course Allocation Portal

Assumptions

1. I have implemented the list objects of students , courses , labs , and Tas using the static allocation (just for the ease of implementation) , whose max size is defined by the global variable `MAX_LEN` , which could be changed as per the test case .
2. The Tutorial Could be conducted with 0 students.

Main Entities

Students

- The student entity is simulated using threads , I have defined student object as :

```
struct student
{
  int id;             // id of the student
  int curr_pref;      // 0 , 1 , or 2
  int course_pref[3]; // the three preference courses
  int time_to_fill;   // store the time to fill in the preferences
  float callibre;     // to store the current callibre of the student

  int curr_state; // -1 if waiting for the slot
                  // 0 if slot allocated
                  // 1 if attending tutorial
                  // 2 if course_finalised

  pthread_t thread_idx;
  pthread_mutex_t mutex; // mutex lock
  pthread_cond_t std;    // for student to be waiting for a tutorial slot to be assigned
                         // so that the student thread does not end up doing busy waiting
};
```

  - `id` : `range : [0 - num_students - 1 ]` : is the student id [0 based indexing]
  - `curr_pref` : `range : [ 0 - 2 ]` : the current choice of the student amongst the 3 preferences he filled
  - `course_pref` : stores the course ids , of the filled in preferences
  - `time_to_fill` : time taken to fill in the preferences
  - `callibre` : the caliber of the student
  - `curr_state` : determines the current state of the student , which could be one of the following :

```
// states for student's parameter curr_state
#define WAITING_SLOT 0       // 0 if waiting for the slot to be allocated
#define ALLOCATED_SLOT 1     // 1 if allocated the slot for the tutorial
#define ATTENDING_TUTORIAL 2 // 2 if attending the tutorial
#define COURSE_FINALISED 3   // 3 if likes the course and has finished the course
```

  - `thread_idx` : the thread object for simulating the student object
  - `mutex` : the lock for students , for protecting the critical section of the student object
  - `conditional varaiable` std : it is used by the student thread while , it waits for course to be allocated to it , (to avoid **busy waiting** it is signaled by the courses threads upon successful allocation of the tutorial seat or if the course becomes unavailable , and then the student thread continues it's execution)

Courses

- Courses are simulated using the threads I have defined the courses as :

```
struct course
{
  /*** INPUT VARIABLES FOR THE COURSE ***/

  int id;             // course id
  char name[MAX_LEN]; // course name
  float interest;     // interest quotient of the course
  int num_labs;       // number of labs eligible for giving TAs
  int labs[MAX_LEN];  // store the labs ids
  int max_slots;      // maximum number of slots a TA can take up

  /*** SELF_DEFINED VARIABLES , SHOULD BE INITIALISED ***/

  bool look_ta;          // 1 if looking for a TA else 0
  bool is_active;        // to check if course has no eligible TA , is_active is set to 0 ( or false )
  pthread_t thread_idx;  //  thread to simulate courses
  pthread_cond_t allot;  //  conditional variables , for taking up the tutorial
  pthread_mutex_t mutex; // mutex

  int slots;        // total slots created
  int slots_filled; // available slots
  int curr_ta;      // the current TA idx
  int curr_lab;     // the current lab , from which TA was alloted
};
```

  - `id` : The id of the course , in range `[ 0 - num_courses - 1]`
  - `name` : name of the course
  - `interest` : %chance that interest student develops for the course (0 - 1)
  - `num_labs` : the number of labs from which ta could be taken
  - `labs` : stores the list of labs
  - `max_slots` : the maximum slots a ta can allot
  - `look_ta` : boolean expression that tells to look for TA or not to look for TA
  - `is_active` : boolean variable that , is **1** if course is **active** and **0** if course is **inactive**
  - `thread_idx` : thread object , for simulating course
  - `allot` : **conditional variable** used by students , to wait while the tutorial is being conducted (to avoid **busy waiting**)
  - `mutex` : mutex lock to protect the critical section of the course object
  - `slots` : slots assigned for the tutorial
  - `slots_filled` : slots filled , out of the total slots allocated for the tutorial
  - `curr_ta` : stores the current id of the TA for the course
  - `curr_lab` : stores the current id of the lab from which the TA is selected

---

Labs

- Labs are treated as a resource that is common (or shared between the multiple course threads) . I defined the lab objects as :

```
struct labs
{
  int lab_id;            // id of the lab
  char name[MAX_LEN];    // name of the lab
  int num_tas;           // total TAs in the lab
  struct ta TA[MAX_LEN]; // store the TA objects
  int max_tashipS;       // max TAs
  int num_tut;           // number of tutorials taken by the lab , used for checking the fact if all the TAs have finished their quota
```

```
  pthread_mutex_t mutex; // mutex
};
```

- `lab_id` : id of the lab
- `name` : name of the lab
- `num_tas` : number of tas , in the lab
- `TA` : list of TAs
- `max_taships` : the maximum Ta ships a TA could do
- `num_tut` : stores the number of tutorials conducted by the TAs of the lab at any moment , used to check if a lab has exhausted all of it's ta resources.
- `mutex` : lock to protect the critical section

TAs

- TAs are treated as resource , which could be defined as :

```
struct ta
{
    int ta_id;             // TA id
    int lab_id;            // Lab id to which the TA belongs
    int number_of_turns;   // number of tutorials taken by the TA
    int is_taking_tut;     // 1 if tut is ongoing else 0
    pthread_mutex_t mutex; // mutex for TA
};
```

- `ta_id`
- `lab_id`
- `number of turns` : the number of TAships TA has done
- `is_taking_tut` : boolean expression to check if TA is taking/conducting a tutorial
- `mutex` : to protect the critical section of the code.

Thread Functions

- Student Thread :
  - As the student thread is born it goes into a loop looping over the 3 filled preferences filled by the student , and goes on sleep on the conditional variable `std` as defined above , until it's allocated a slot or the course becomes inactive and is withdrawn from the portal.
  - In the main loop function , student first waits for until it's `status` from `WAITING_SLOTS` to `ALLOCATED_SLOTS` , (this is done by sleeping on the conditional variable `std` , which is signaled in the course thread , after changing it's status to `ALLOCATED_SLOTS` using `pthread_cond_signal` .
  - Students threads continues it's execution and , now enters into a state where it waits on the conditional variable `allot` of the course of current preference of the student. Which is later signaled when course completes it's tutorial. Once the tut is completed , the thread is signaled using the `pthread_cond_broadcast` that wakes up all the student threads waiting on the conditional variable.
  - Then we check for the probability if the student likes the course , using the random number generator and comparing it with probability factor `interest*calibre`

```
float prob_liking = probability(student_list[idx].callibre, course_list[curr_course_idx].interest); // probability of him liking th
float random_prob = (float)rand() / RAND_MAX;                                                       // get random probability for c
if (random_prob <= prob_liking)
{
```

```
    pthread_mutex_lock(&student_list[idx].mutex);
    student_list[idx].curr_state = COURSE_FINALISED;
    pthread_mutex_unlock(&student_list[idx].mutex);
    printf("Student %d has selected course %s permanently\n", idx, course_list[student_list[idx].course_pref[choice]].name);
    break; // exit the simualtion
}
```

- if student likes the course , status is changed to `COURSE_FINALISED` and corresponding statements are printed , and student exits the simulation.
- Else the student thread continues to look for other courses in the preferences filled , by the student.

- Course Thread :
  - For simulating the course , I divided the problem into 3 sub problems :
    1. Look for a TA for conducting the tutorial
    2. Generate random number of slots (between 1 & `course_max_slots`)
    3. Allocate these slots to the students waiting for the course to be allocated
    4. Conduct the tutorial
    5. Repeat until tutorial could be conducted (i.e. if there are students left in the simulation and TAs that could be assigned for the course)

The functions :

1. `assign_ta` : It loops over the list of labs , available for the course until a TA could be allotted for the course , the function returns -1 in case there are no more `TAs` that could be assigned to the course , 0 if there are `TAs` available (but no TA has been allotted as the TA might be busy conducting tut for some other course) , so we wait loop until the return status is 0 , and break if return status is non-zero .

```
int flag = assign_tas(idx);
while (flag == 0)
{
 // keep looking for the TAs until u find one
flag = assign_tas(idx);
}
```

2. if the return status was 1 , that means the course has been allocated a `TA` , and hence , we generate random number of slots between 1 and course_max_slots and for the course

```
int random_int_in_range(int lower_limit, int upper_limit)
{
    int range = upper_limit - lower_limit + 1;
    int random_num = rand() % range + lower_limit;
    return random_num;
}
```

```
// and assign random slots
if (flag == 1)
{
  // got the TA allocate the slots
  course_list[idx].slots_filled = 0;
  course_list[idx].slots = random_int_in_range(1, course_list[idx].max_slots);
  // ~ allocated the slots for the course , done the second part
  // it is a blocking function , waits until there is atleast one student avaialble for the tut to be conducted
  assign_slots(idx);
  // got atleast one student , now conduct the tutorial
  conduct_tutorial(idx);
}
```

3. In the `assign_ta` function , I am checking if A lab is available for giving TAs or not by comparing the values of `num_tut` and the product `num_tas * max_taships` , It allocates a TA to the course.
4. Next , we call the `assign_slot` function , that allocates the generated slots to the students **NOTE THAT , I CHOSE TO CONDUCT A TUTORIAL WITH 0 STUDENTS AS WELL , IN CASE THERE ARE NO STUDENTS AVAILABLE TO TAKE UP THE TUTORIAL , IN ORDER TO AVOID THE INFINITE LOOPING AND HALTING OF THE PROGRAM .**
    a. It loops over the list of students and checks their status `WAITING_SLOT` and course_preference matches with the course id of the thread , and changes the status of the students to `ALLOCATED_SLOTS` and signals the waiting thread , signalling them to continue their execution .
5. Next , we conduct the tutorial that is implemented in the function `conduct_tutorial` , wherein , we signal the students waiting on the conditional variable `course[id].allot` , using `pthread_cond_broadcast` after changing status for the students with matching status and course id , and wake them up to continue their further execution.
6. In case the return value is -1 from the `assign_ta` function , we need to signal the student threads waiting for the course slot to be allocated , and withdraw the course that i have implemented as shown below :

```
// remove the course and signal the waiting students if any
pthread_mutex_lock(&course_list[idx].mutex);
course_list[idx].is_active = false; // change the course status to inactive
pthread_mutex_unlock(&course_list[idx].mutex);
for (int i = 0; i < num_students; i++)
{
    if (student_list[i].course_pref[student_list[i].curr_pref] == idx)
    {
        int br = pthread_cond_signal(&student_list[i].std);
        assert(!br);
    }
}
```