Assignment 5

Concurrency, Parallelism and Networking

Operating Systems and Networks, Monsoon 2021

INTERNATIONAL INSTITUTE OF INFORMATION TECHNOLOGY, HYDERABAD

Due on: 11:55 PM @ 23rd November, 2021

The purpose of this assignment is to introduce students to basics of multi-threaded programming; synchronizing multiple threads using locks, condition variables and semaphores ; basic networking concepts.

Question 1: An alternate course allocation Portal [35 points]

Scenario:

Sanchirat and Pratrey, students of the college, have been asked to use their skills to design an unconventional course registration system for the college, where a student can take trial classes of a course and can withdraw and opt for a different course if he/she does not like the course. Different labs in the college have been asked to provide students who can act as course TA mentors temporarily and take the trial tutorials.

Following are the details:

Defining the entities in play

- There are 4 main entities in this simulation:
  - Courses offered
  - Various labs around the college (each consisting of a set of student mentors)
  - Each student mentor belongs to exactly one lab and is available for mentoring courses for which the lab has been shortlisted
  - Students participating in the course registration to study the content in the course

Defining the Labs

- Each lab has **n_i** students associated with it, who are willing to be TA mentors for this semester
- Each lab has its own limit on the number of times a member of the lab can be TA mentor in the semester (Eg: if a lab's limit is 3 and a member has tutored for CPro 2 times and DSM once, then he/she is not allowed to become a TA mentor any further.).

Defining the courses

- There are **"num_courses"** number of courses available for registration
- Each course has an **"interest"** associated with it, which defines the general student interest in the course over the past year. This value is a number between 0 and 1.
- Each course wants to select its TA mentors from a set of assorted labs [Let this set of labs be L].
- Each course will have at most one TA mentor assigned to it at any given time.
- A course C is first assigned one TA mentor (T1) such that T1 belongs to one of the labs in L and is currently free, T1 allots 'D' slots for the students who have course C as their current preference and takes a tutorial for the course; T1 leaves after this and becomes free again ; Then C is assigned another TA (T2), T2 allots slots and takes a tutorial for the course and so on.
- Also, each course's tutorial can be attended by a maximum of **"course_max_slot"** students i.e. the value of "D" is constrained by **"course_max_slot"**.

Defining how Students fill their preferences

- Each student lists 3 courses, in the decreasing order of priority, as his/her course preferences. The student fills in his preferences at a **time 't' (in seconds),** which is provided as input. Note that the student should be considered for the course allocation only after he fills in the priorities.
- Let's say that the student's current preference is course C. Then, the student will continue to wait for a seat at the course.
- Once a student has been selected to attend the course, the probability he likes the tutorial and finalizes course C (ie does not withdraw from it) is defined as:-

    - **Prob = (interest of the course) x (student's calibre)**
    - Please use this probability to randomly choose whether the student withdraws from the course or he finally takes it.
- A student will abandon his current preference and move to the next preference in the following cases:
    - The student attended the course but withdrew it
    - The course was removed from the portal before the student got a chance to attend the course

Working of the tutorials

- Each TA, after being assigned to a course C, decides to take a tutorial with slots for 'D' students who have course C as their current preference.
- If all the slots offered by a TA are filled or there are no more students available for the course right now, the TA can start the tutorial and once it's done, leave the course. Students who may have this course as a later priority (and not as their current priority) do not need to be considered while considering the number of students available for the course.
- In other words, for the conduction of a tutorial for course C,
    - **Decide D as a random integer between 1 and "course_max_slots" (inclusive)**.
    - Let W be the number of students who currently have course C as their current preference.
    - If W>=D, fill the D slots and start conducting the tutorial.
    - If W<D, then leave the remaining (D-W) slots unfilled and start conducting the tutorial.
- A new TA can be allotted for a course only after the previous TA has left i.e, at any given point in time, a course can have at most 1 TA mentor taking a tutorial for it.
- A TA mentor can take a tutorial for atmax one course at any given instant.

Courses being removed from the Portal

Let's say that course C accepted TAs from **'G' labs** and let the sum of members across all 'G' labs be "S". Then, if all the S people have reached the maximum times they could be a TA mentor; then this would mean that it is impossible for course C to have a TA allotted to it in the future. In such a case, **course C would be withdrawn from the portal and all the students who had course C as their current preference would be moved on to the next preference**.

BONUS [5 points]

In the interest of equal opportunity for all, a member of the lab (let's say M) who has done 't' TAships can take up his 't+1' th TA ship only when all the other members within the lab have also done 't' TA ships or are currently doing their 't' th TA ship. If the said condition is not satisfied, then M will wait for his fellow members so that the condition can be satisfied. Implement this to be eligible for Bonus.

A summary of the states the various entities can take:

- Student
    - Not registered yet
    - Waiting for a seat at the course he filled as his first preference (course 1)
    - Attending tutorial for course 1
    - Waiting for a seat at the course he filled as his second preference (course 2)
    - Attending tutorial for course 2
    - Waiting for a seat at the course he filled as his second preference (course 3)
    - Attending tutorial for course 3
    - Selected a course permanently
    - Did not like all 3 courses and has exited the simulation
- Course
    - Is currently waiting for a TA to become available
    - Has an allotted TA, waiting for slots to be filled
    - Has an allotted TA who is conducting tutorials
    - Has been withdrawn
- Lab
    - At Least one TA mentor who is a part of the lab has not completed his quota
    - All people in the lab have exhausted the limit on the number of times someone can accept a TAship and so, the lab will no longer send out any TAs

---

Input format

- The first 3 lines contain the number of students, number of labs and number of courses respectively.
- The next “num_courses” line contains 4 tokens each and then a list: the first token contains the name of the course and the second token contains the “interest quotient” of the course.The third token is “course_max_slots”. The fourth token is the number of labs from which the course accepts TAs (let’s call this ‘p’). Then, there are ‘p’ numbers which are the IDs of the corresponding labs (0 indexed).
- The next “num_stu” lines contain 5 tokens each: the first token contains “the calibre quotient of the student”, the next 3 tokens are the course IDs of the student’s preferred courses (in decreasing order of preference) and the last token is the time (in seconds) after which he registers on the portal. Please assume the first student to have ID as 0, second student to have ID as 1 and so on.
- The next “num_labs” lines contain 3 tokens each: name of the lab, number of TAs in the lab and the maximum number of times a TA in the said lab is allowed to TA a course over the entire simulation.

```
<num_students> <num_labs> <num_courses>

<name of course 1> <interest quotient for course 1> <maximum number of slots which can be allocated by a TA> <number of labs from which course accepts TA> [list of lab IDs]
    .
    .
<name of the last course> <interest quotient for the last course> <maximum number of slots which can be allocated by a TA> <number of labs from which course accepts TA> [list of lab IDs]

<calibre quotient of first student> <preference 1> <preference 2> <preference 3> <time after which he fills course preferences>
    .
    .
<calibre quotient of last student> <preference 1> <preference 2> <preference 3> <time after which he fills course preferences>

<name of first lab> <number of students/TA mentors in first lab> <number of times a member of the lab can TA in a course>
.
.
.
<name of last lab> <number of students.TA Mentors in the last lab> <number of times a member of the lab can TA in a course>
```

Output Format:

Print the relevant output statements in case of the following events for each entity:

Students:

Use 0-based indexing for the students.

- Event 1: Student filled in preferences for course registration
  “Student i has filled in preferences for course registration”
- Event 2 : Student has been allocated a seat in a particular course
  “Student i has been allocated a seat in course c_j”
- Event 3 : Student has withdrawn from a course
  “Student i has withdrawn from course c_j”
- Event 4 : Student has changed their preference
  “Student i has changed current preference from course_x (priority k) to course_y (priority k+1)”
- Event 5 : Student has selected a course permanently
  “Student i has selected course c_j permanently”
- Event 6 : Student either didn’t get any of their preferred courses or has withdrawn from them and has exited the simulation
  “Student i couldn’t get any of his preferred courses”

Courses:

Use course name for output mentioning courses.

- Event 7 : Seats have been allocated for the course
  “Course c_i has been allocated j seats”
- Event 8 : TA has started the tutorial with k seats, where k <= number of seats allocated for the course (j from the previous step)
  “Tutorial has started for Course c_i with k seats filled out of j”
- Event 9 : TA has completed the tutorial and left the course

“TA t_j from lab l_k has completed the tutorial and left the course c_i”

- Event 10 : The course doesn’t have any eligible students for TA ship available and is removed from the course offerings.
  “Course c_i doesn’t have any TA’s eligible and is removed from course offerings”

Labs:

Use lab name for output mentioning courses and 0 based indexing for the TAs in a lab.

- Event 11 : TA from the lab has been allocated to a course for his nth TA ship
  “TA t_i from lab l_j has been allocated to course c_k for nth TA ship”
- Event 12 : All students from the lab have completed the maximum number of TA ships which can be done by them and the lab no longer has students available for TA ships.
  “Lab l_i no longer has students available for TA ship”

NOTE: Events have been numbered for convenience while referring to Moodle, etc and not in chronological order . For example: We don’t mean to convey that a message for event 10 needs to be printed for a message for event 12.

Example

Sample Input:

```
10 3 4
SMAI 0.8 3 2 0 2
NLP 0.95 4 1 0
CV 0.9 2 2 1 2
DSA 0.75 5 3 0 1 2
0.8 0 3 1 1
0.6 3 1 2 3
0.85 2 1 0 1
0.5 1 2 3 2
0.75 0 2 1 3
0.95 1 0 2 2
0.4 3 0 2 3
0.1 0 3 1 2
0.85 1 0 3 1
0.3 0 1 2 1
PRECOG 3 1
CVIT 4 2
RRC 1 3
```

Sample Output (Please note that since the simulation is not deterministic, many other valid outputs would be acceptable as well):

```
Student 0 has filled in preferences for course registration
Student 2 has filled in preferences for course registration
Student 9 has filled in preferences for course registration
Student 8 has filled in preferences for course registration
TA 0 from lab PRECOG has been allocated to course SMAI for his 1st TA ship
TA 1 from lab CVIT has been allocated to course CV for his 1st TA ship
TA 1 from lab PRECOG has been allocated to course NLP for his 1st TA ship
Course CV has been allocated 2 seats
Course SMAI has been allocated 2 seats
Student 7 has filled in preferences for course registration
Student 3 has filled in preferences for course registration
Student 5 has filled in preferences for course registration
Course NLP has been allocated 4 seats
Student 3 has been allocated a seat in course NLP
Student 0 has been allocated a seat in course SMAI
Student 8 has been allocated a seat in course NLP
Student 7 has been allocated a seat in course SMAI
Student 2 has been allocated a seat in course CV
Tutorial has started for Course SMAI with 2 seats filled out of 2
Student 5 has been allocated a seat for course NLP
Tutorial has started for course CV with 1 seat filled out of 2
Student 4 has filled in preferences for course registration
Student 6 has filled in preferences for course registration
```

```
TA 0 from lab PRECOG has completed the tutorial for course SMAI
Student 7 has withdrawn from course SMAI
Student 0 has selected the course SMAI permanently
Student 7 has changed current preference from SMAI (priority 1) to DSA (priority 2)
TA 0 from lab RRC has been allocated to course DSA for his 1st TA ship
Tutorial has started for course NLP with 3 seats filled out of 4
TA 2 from lab PRECOG has been allocated to course SMAI for his 1st TA ship
Lab PRECOG no longer has students available for TA ship
Student 2 has selected the course CV permanently
Course SMAI has been allocated 3 seats
Student 9 has been allocated a seat in course SMAI
Course DSA has been allocated 3 slots.
TA 1 from lab PRECOG has completed the tutorial for the course NLP
Student 4 has been allocated a seat in course SMAI
Student 3 has selected the course NLP permanently
Student 1 has been allocated  a seat in course DSA
Student 5 has selected the course NLP permanently
Tutorial has started for course SMAI with 2 slots filled out of 3
Student 8 has selected the course NLP permanently
Course NLP does not have any TA mentors eligible and is removed from course offerings
Student 6 has been allocated a seat in course DSA
Student 7 has been allocated a slot in the course DSA
Tutorial has started for course DSA with 3 slots filled out of 3
TA 2 from lab PRECOG has completed the tutorial for the course SMAI
Student 4 has selected the course SMAI permanently
Student 9 has withdrawn from the course SMAI
TA 0 from lab RRC has completed the tutorial for the course DSA
Student 9 has changed current preference from SMAI (priority 1) to NLP (priority 2)
Student 6 has selected the course DSA permanently
Student 7 has withdrawn from the course DSA
Student 9 has changed current preference from NLP(priority 2) to CV(priority 3)
Student 1 has selected the course DSA permanently
Student 7 has changed current preference from DSA(priority 2) to NLP(priority 3)
Student 7 could not get any of his preferred courses
TA 0 from lab RRC has been allocated to course CV for his 2nd TA ship
Course CV has been allocated TA 0 from lab RRC
Course CV has been allocated 2 slots
Student 9 has been allocated a slot in course CV
Tutorial has started for course CV with 1 slots filled out of 2
TA 0 from lab RRC has completed the tutorial for the course CV
Student 9 has selected the course CV permanently
```

Note: You can always add more print statements but these are the minimum requirement.

Instructions

- Each student, course and lab must be simulated using threads. Also, trying developing sound logic using locks, semaphores and conditional variables. [Note: By “student”, we mean to refer to the people who have registered for course registration and not the TA mentors]
- Your code must allow for tutorials for different courses to be conducted parallely at the same time
- Your code must allow several courses to fill their slots with the students in parallel at the same time
- Different courses must be able to choose TAs parallely at the same time. In other words, you are not allowed to implement courses C1 and C2 to look for TAs one by one as this would defeat the purpose of multithreading.
- Stop the simulation when all the students have either finalised their courses or couldn’t get any of their choices.
- **Use appropriate small delays for the simulation.** A small delay must be added when the TA is conducting a tutorial.
- The thread corresponding to the students is **not allowed to “busy wait”** for it to be allotted a course. In other words, the thread must give up the CPU while waiting.
- We **do not** expect the thread corresponding to course C to simultaneously look for mentor TAs in different labs. It is acceptable if course C checks for unoccupied mentor TAs within the labs sequentially again and again.

Question 2: The Clasico Experience [35 points]

A football match is taking place at the Gachibowli Stadium. There are 2 teams involved: *FC Messilona (Home Team)* and *Benzdrid CF (Away Team)*. People from all over the city are coming to the stadium to watch the match. You have to create a simulation where people enter the stadium, buy tickets to a particular zone (stand), watch the match and then exit.

Defining the entities at play

There are 4 main entities at play here:

- The Zones in the Stadium
- The person (spectator)
- The Goals scored by the Home/Away Teams
- The Groups (to which a person belongs)

Defining the ZONES in the stadium

- There are 3 zones in the stadium:
    - Zone H
    - Zone A
    - Zone N
- Each zone has a fixed capacity beyond which it cannot accommodate people.
- However, if a person currently in zone ‘x’ leaves the zone and goes to wait at the gate, an extra seat becomes free at the zone which can be accommodated by another waiting person now or in the future.

Defining the concept of Spectators

- Let there be a total of **‘num_people’** people who enter the stadium to buy tickets for the match. Since people get free from work at different times, each person enters the stadium after a **time ‘T’** (in seconds) which is specific to each person.
- Each person is either a HOME FAN, an AWAY FAN or a NEUTRAL FAN and hopes to buy a ticket in one of the zones for which he is eligible. **At no time should a ZONE have an empty seat if some person who is eligible to buy a ticket at this zone is waiting.**

Person Type	Zones for which he/she can buy tickets for
H (Home) Fan	Zones H and N
A (Away) Fan	Zone A only
N (Neutral) Fan	Zones H, A and N

- Each person has a **“PATIENCE VALUE P”** [P is an integer] which is the time (in sec) for which he will wait to get a ticket in any of his eligible zones.
- People of Hyderabad are busy. Hence, no one wants to watch the match for more than **‘X’ units (SPECTATING TIME)** [X is an integer]. This value is fixed across all people.

Defining the concept of Goals

- People who are HOME/AWAY Fans cannot see their teams perform badly defensively. So, they get enraged and leave the STAND to wait at the gate if the opponent team scores/has scored **‘R’ or more goals**.
- For Eg: If a Away Fan who is present in some ZONE watching the match sees the opponent team score their R th goal in the entire match so far, he/she will immediately leave.
- For eg: If an Away Fan buys a ticket, enters the zone and notices that the opponent team has already scored greater than or equal to ‘R’ goals, even in such a case he leaves the ZONE and goes to wait at the gate.
- Both teams have goal scoring chances which they are able to convert to Goals with some probability. Please refer to INPUT FORMAT to see how “Goal Scoring” is to be simulated.

Simulating a person’s exit from the Stadium

- To summarize , the person can leave the match and **will proceed to wait at the EXIT gate** in the following 3 scenarios:
    - **Scenario 1:** Person arrived at time ‘T’ but was not able to procure a seat till time (T+P).

    - **Scenario 2:** Person arrived at time ‘T’, got a seat at time J (where T <= J <= T+P) and decided to exit the stand at time (J+X).
    - **Scenario 3:** Person (who is a HOME or AWAY fan) was allotted a seat but decides to leave before time (J+X) because of his team’s Poor performance. (Please see the section called “concept of Goals” for more details)
- Each person belongs to a group. The friends of a person are all the people within the same group as that person.
- After the match, each person wants to have a nice meal. His exit from the simulation after reaching the EXIT Gate should be simulated as follows:

In case Bonus is not implemented

- As soon as the person reaches the EXIT Gate, he does not wait for his friends and drives off right away (i.e. exits the simulation).

For bonus marks [10 points], simulate the person’s exit in the following way:

- As soon as the person reaches the EXIT gate, he/she waits at the gate for his friends to reach the EXIT gate as well.
- Once all the friends within the same group reach the gate, all of them leave the Stadium and drive off, thus exiting the simulation.

Summarizing the states

- Person (Spectator)
    - Has not reached the stadium yet
    - Is waiting for a seat to become available in the appropriate zones
    - Is watching the match currently as a spectator in one of the ZONES
    - Has reached the EXIT gate
    - Is waiting at the gate for his friends [SKIP this state if bonus not implemented]
    - Drives off for dinner ie EXITS the simulation

Input format

- The first line contains 3 integers describing the capacities of the 3 zones.
- The next line contains the value of SPECTATING TIME ie X.
- The next line contains the total number of groups i.e. ‘num_groups’.
- This is followed by a description for each of the groups.
    - The 1st line for each group description will be the number of people within the group. Let this number be ‘k’.
    - This is followed by ‘k’ lines with each line describing a person.
    - Each person is described as follows: <Name of person> <Home/Away/neutral denoted by H/A/N> <Time at which he reaches the stadium> <Patience Time> <Number of goals which the opponent needs to score to make the person enraged>
    - For neutral fans, the value of input given for <Number of goals which the opponent needs to score to make the person enraged> will be -1.
- Next line contains the **NUMBER OF GOALSCORING CHANCES ‘G’** throughout the match.
- Next ‘G’ lines contains the description of each chance using 3 tokens:
    - **Desc:** <Team with the chance ie H/A> <Time in sec elapsed since the beginning of the match after which the chance was created> <Probability that the chance actually results in a goal>
    - You can assume that no 2 chances are created at the same instant i.e. the second token for each chance’s description would be distinct.
    - You can assume that the chances are inputted in increasing order of times at which they were created.
    - Please assume that the match starts the moment the simulation begins
    - Also, probability would be a number between 0 and 1 (inclusive).

```
<Capacity of Zone H>  <Capacity of Zone A>  <Capacity of Zone N>
<Spectating time X>

<Number of groups>
```

<Number of people in group 1>
<Name of person 1> <H/N/A> <Time of reaching stadium> <Patience time for person 1> <Num goals>
.
.
.
<Name of last person in group> <H/N/A> <Time of reaching stadium> <Patience time for last person> <Num goals>

<Few lines describing group 2>
…
<Few lines describing the last group>

<Number of goal scoring chances throughout the match>
<Team with the first chance> <Time elapsed> <Probability of chance being converted into a goal>
.
.
.
<Team with the last chance> <Time elapsed> <Probability of chance being converted into a goal>

Output

Print relevant output statements for the following events for each entity:

Spectator:

Use name of the spectator for output.

- Person has reached the stadium
  “Person p_i has reached the stadium”
- Person has got a seat in a zone
  “Person p_i has got a seat in zone z_j”
- Person couldn’t get a seat
  “Person p_i couldn’t get a seat”
- Person watched the match for X time and is leaving
  “Person p_i watched the match for X seconds and is leaving”
- Person is leaving due to the bad performance of his team
  “Person p_i is leaving due to the bad defensive performance of his team”
- Person is waiting for his friends at the exit [Only if BONUS has been implemented]
  “Person p_i is waiting for their friends at the exit”

Team:

Use team name for teams in the output

- Team converted the goal scoring chance to a goal
  “Team t_i have scored their Yth goal”
- Team failed to convert the goal scoring chance to a goal
  “Team t_i missed the chance to score their Yth goal”

Events related to entities exiting the simulation:

Use 1-based indexing for groups

- If bonus has been implemented, : All members of the group have reached the exit gate and the group is leaving for dinner ie exiting the simulation
  “Group g_i is leaving for dinner”
- If bonus has NOT been implemented : Person has reached the EXIT gate and is finally leaving
  “Person P_i is leaving for dinner”

EXAMPLE

Sample Input

```
2 1 2
3
2
3
Vibhav N 3 2 -1
Sarthak H 1 3 2
```

```
Ayush A 2 1 4
4
Rachit H 1 2 4
Roshan N 2 1 -1
Adarsh A 1 2 1
Pranav N 3 1 -1
5
H 1 1
A 2 0.95
A 3 0.5
H 5 0.85
H 6 0.4
```

Sample Output [There are other possible valid outputs as well]

```
t=1 : Sarthak has reached the stadium
t=1: Adarsh has reached the stadium
t=1 : Rachit has reached the stadium
t=1:Team H has scored their 1st goal
t=2: Rachit has got a seat in zone H
t=2: Sarthak has got a seat in zone N
t=2: Adarsh has got a seat in zone A
t=2: Adarsh is leaving due to bad performance of his team
t=2: Roshan has reached the stadium
t=2: Ayush has reached the stadium
t=2: Roshan has got a seat in zone A
t=2: Team A has scored their 1st goal
t=3: Adarsh is waiting for their friends at the exit
t=3: Team A missed the chance to score their 2nd goal
t=3: Pranav has reached the stadium
t=3: Pranav has got a seat in zone N
t=3: Vibhav has reached the stadium
t=4: Vibhav has got a seat in zone H
t=4: Ayush could not get a seat
t=5: Sarthak watched the match for 3 seconds and is leaving
t=5: Ayush is waiting for their friends at the exit
t=5: Rachit watched the match for 3 seconds and is leaving
t=5: Rachit is waiting for their friends at the exit
t=5: Roshan watched the match for 3 seconds and is leaving
t=5: Roshan is waiting for their friends at the exit
t=5: Team H has scored their 2nd goal
t=5: Sarthak is waiting for their friends at the exit
t=6: Pranav watched the match for 3 seconds and is leaving
t=6: Team H missed the chance to score their 3rd goal
t=7: Vibhav watched the match for 3 seconds and is leaving
t=7: Pranav is waiting for their friends at the exit
t=7: Group 2 is leaving for dinner
t=8: Vibhav is waiting for their friends at the exit
t=8: Group 1 is leaving for dinner
```

Note: The ‘t’ in the above output is provided just for explanation. You do not need to print the time. Also, feel free to choose colors of your own choice.

Instructions

- You are expected to use multithreading for simulating different (not necessarily all) entities.
- You might also treat certain entities as resources.
- You are free to use locks, conditional variables, semaphores as you wish.
- Exit the simulation when all people have exited the stadium. You can assume that input given will be such that all goal scoring chances will be created before the last person exits the simulation.
- **No busy waiting [repeatedly checking if a certain condition is satisfied] is allowed by any thread.**
- Ignore the time it takes to move from the zone to the exit gate.
- Please use randomization to simulate probabilistic events like conversion of chance to a goal.
- Your implementation should be such that it must be possible for different people to:

    - Arrive and buy tickets at the same time
    - Leave the stand and proceed to the gate at the same time
    - Exit the simulation at the same time
- You are not allowed to deal with the above 3 cases one-by-one for each person. Dealing one-by-one would defeat the purpose of this assignment.
- The Fans do not have any preference among the eligible zones. For Eg: If a seat is available in both H and N zones, the person can buy a ticket for either of them.

Question 3: Multithreaded client and server [30 points]

Context: Multiple clients making requests to a single server program
Imagine a simple scenario where multiple clients are making requests to a single server.

Client program

- Each user request has 2 main characteristics:
    - The time at which the request has been made
    - The nature of the request/the command issued
- Let's say there are **'m'** user requests throughout the course of the simulation.
- In order to simulate different users querying to the same server, you are supposed to create 'm' threads: each representing a single user's request.
- Each of these user request threads will then try to connect with the server independent of each other (to give the impression of multiple users from different parts of the world, sending commands to the same server).
- Once the connection to the server has been made, the user request thread can communicate with the assigned worker thread using a **TCP socket**.
- Whatever response the user request thread receives from the server, it must output it. (Please see sample for more details)

Server program

- The server will maintain a dictionary. A dictionary is a container that stores mappings of unique keys to values.
- The dictionary values must be common across all clients. The key-value pairs created by one client should be visible to all other connected clients.
- In order to correctly handle multiple concurrent clients, the server should have a multithreaded architecture (with a **pool of worker threads**).
- The server will have **'n' worker threads** in the thread pool. These 'n' threads are the ones which are supposed to deal with the client requests. As a result, atmax 'n' client requests will be handled by the server at any given instant.
- As soon as the server starts, it must spawn these 'n' worker threads.
- Then, the server begins to listen for client requests to connect.
- Whenever a new client's connection request is accepted (using the accept() system call), the server is expected to then designate one of the worker threads to deal with the client's request. [Hint: use a queue in which the server can push client request entities and worker threads can pop to deal with client requests]
- The worker threads, while dealing with the client requests, might need to perform read/write operations on the dictionary. They are also expected to send an apt output to the client thread (Please see format below). Also, whenever a worker thread completes a task, put an artificial sleep of 2 seconds before sending the result back to the client.

Following are the commands which can be inputted by the user:

- **insert <key> <value> :** This command is supposed to create a new "key" on the server's dictionary and set its value as <value>. Also, in case the "key" already exists on the server, then an appropriate error stating "Key already exists" should be displayed. If successful, "Insertion successful" should be displayed.
- **delete <key> :** This command is supposed to remove the <key> from the dictionary. If no key with the name <key> exists, then an error stating "No such key exists" should be displayed. If successful, display the message "Deletion successful".
- **update <key> <value> :** This command is supposed to update the value corresponding to <key> on the server's dictionary and set its value as <value>. Also, in case the "key" does not exist on the server, then an

appropriate error stating “Key does not exist” should be displayed. If successful, the updated value of the key should be displayed.

- **concat <key1> <key2> :** Let values corresponding to the keys before execution of this command be {key1: value_1, key_2:value_2}. Then, the corresponding values after this command's execution should be **{key1: value_1+value_2, key_2: value_2+value_1}**. If either of the keys do not exist in the dictionary, an error message “`Concat failed as at least one of the keys does not exist`” should be displayed. Else, the final value of key_2 should be displayed. No input will be provided where <key_1> = <key_2>.
- **fetch <key>** : must display the value corresponding to the key if it exists at the connected server, and an error “Key does not exist” otherwise

Other details

- We constrain the ‘keys’ to be **non-negative integers (<=100)** and ‘values’ to be non-empty strings containing only alphabets and numbers.
- For each of the above commands, any other error (which might have not been mentioned at the top), must also be handled and corresponding error messages should be displayed.
- Note that the above messages specify the commands given by the user to the client.c/client.cpp. There is no restriction on the format of the messages exchanged between the client and the server. Also, you are free to use whatever PORT Numbers you want.
- We may automate checking for this question, so please follow the exact output instructions.
- As mentioned, it might be possible for two or more worker threads trying to make a change at the exact same key-value pair due to their corresponding client’s requests. So, you must take steps to make the **dictionary thread-safe**.
- The server and client programs may be invoked from different directories.
- You are allowed to use both C and C++ for this question. **Even if you use C++, you are still expected to use libraries “pthread.h” and “semaphore.h”. The use of C++ is allowed for your convenience while dealing with strings and data structures**.
- No busy waiting [repeatedly checking if a certain condition is satisfied] is allowed by any thread.
- The server is not allowed to create more than “n” threads during the duration of the entire program.
- Also, if two commands are not accessing common keys, then it must be possible for them to be executed parallely. For eg: If cmd_1 is “insert 1 bakul” and cmd_2 is “concat 3 4”, then it should be possible for 2 worker threads to execute these commands respectively at the same time.

Input and running the code

Server: The server will be run by using the following command:

```
$ ./server <number of worker threads in the thread pool>
```

Client: The client will be run by using the following command:

```
$ ./client
```

- The first line of input would be the total number of user requests throughout the simulation (m).
- The next ‘m’ lines contain description of the user requests in non-decreasing order of the first token
  - <Time in sec after which the request to connect to the server is to be made> <cmd with appropriate arguments>

Sample input

```
11
1 insert 1 hello
2 insert 1 hello
2 insert 2 yes
2 insert 3 no
3 concat 1 2
3 concat 1 3
4 delete 3
5 delete 4
6 concat 1 4
7 update 1 final
8 concat 1 2
```

- Assume that the requests are 0 indexed, i.e. in the above example, assume request "`8 concat 1 2`" to have index 10 and "`2 insert 1 hello`" to have index 1.
- For each concluded user request, the user thread must print the **request index** ; followed by a colon ; followed by the thread worker ID on the server end which catered to the request; followed by the verdict/error/key value as already described above.
- The thead_worker_id can either be 0/1-based indexing numbers or the actual thread ID. You can use any other method for the thread_worker_id with the only restriction being that the thread ID should be distinct for different worker threads.

Sample output:

```
0:140584115967744:Insertion successful
2:140584099182336:Insertion successful
3:140584107575040:Insertion successful
1:140584090789632:Key already exists
4:140584082396928:yeshello
5:140584115967744:nohelloyes
6:140584099182336:Deletion successful
7:140584107575040:No such key exists
8:140584090789632:Concat failed as at least one of the keys does not exist
9:140584115967744:final
10:140584082396928:yeshellofinal
```

General instructions:

- You are required to code Q1 and Q2 in C. For Q3, you can use C++ as well.
- You can use a MAKEFILE to compile your code (if it involves multiple files). Also, mention in the README how to run your code.
- Libraries allowed are pthread.h and semaphore.h
- A **REPORT for each question is a must.** It will have a high weightage during grading and must comprehensively describe your logic.
- The code shouldn't result in any deadlocks/infinite loops/segmentation faults. Ensure that your code terminates successfully for all kinds of test cases.
- Submission format: Include all files in a folder named with your roll number

```
|__ q1/
      |__ files corresponding to Q1
      |__ REPORT
|__ q2/
      |__ files corresponding to Q2
      |__ REPORT
|__ q3/
      |__ files corresponding to Q3
      |__ REPORT
```

- Compress and submit it as **<roll_number>_assignment_5.tar.gz**
- Plagiarism is strictly prohibited.
- You should be able to explain each and every part of your code. Failure to do so will result in heavy penalization.
- You are encouraged to briefly discuss your implementation logic with the TAs.
- It is a **must to use coloring while printing your outputs corresponding to different events for Q1 and Q2**. Check out this link. This will not only make your debugging life easier but will also make it easy for the TA's to understand your code so you can be graded faster.
- Use concise, relevant comments in your code to mark the purpose of variables and conditional statements. This will make the reading of the code easier.
- If there is any possible ambiguity in the problem statement, please get it clarified on Moodle. Unreasonable assumptions during evals will not be entertained.
- A detailed introduction to the pthreads API is here: OSTEP Ch. 27. You can also find several tutorials like this , this and this online.