Report

Overview :

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

- headerq2.h
- q2.c

The program can be run as follows :

```
gcc -pthread q2.c -o q2
./q2
```

Explanation for q2 :

1. headerq2.h :

in headerq2.h we declare all the required structs and arrays for threads, lists, mutexes, conditional variables.

I have used 4 structures for this question.

Convention followed for zone numbers:

```
// 0 is H
// 1 is A
// 2 is N
```

- Spectator :

```
struct spectator {
    int specid;                          // id for the spectator.
    int groupid;                         // group id for the spectator.
    int specsleeptime;                   // sleeptime for the spectator.
    int patience;                        // patience value for the spectator.
    int rmax;                            // max goals rage capacity for the spectator.
    int supportingteam;                  // supporting team index (H,A,N)
    char specname[SIZE];                 // name of the spectator
    int is_avail;                        // 0 while sleeping and 1 while awake
    int totspec;                         // total spectators stored cuz it isnt global
    int isalloc;                         // to check if a spectator is allocated in a zone or not.
    int zoneno;                          // the zone no that the spectator has been allocated to, -1 initially (H,A,N)
};
```

- Zone :

```
struct zone {
    int zoneid;                          // Zone id for (H,A,N)
    int tot_capacity;                    // Total capacity for each zone.
    int curr_capacity;                   // number of cuurently filled seats, 0 initally.
};
```

- Team :

```
struct team {
    int team_id;                         // Team id (0 or 1), 0 for H and 1 for A
    int goalsscored;                     // goals scored untill now for each team.
};
```

- Match details :

```
struct matchdetails {
    int eventid;                         // Id for the event taking place
    int whichteam;                       // teamid of the team that has the scoring chance.
    int sleeptime;                       // time at which the event takes place.
    float successprob;                   // success probability of the goal
    int maxid;                           // stores the total events ( not used tho :( )
};
```

The comments describe each of the structures.

I have used 3 functions one for handling spectators, one for allocating zones to the spectators and one to track goals scored by each team during the course of the match.

```
void * handlespectators(void * arg);
void * zoneallocater(void * arg);
void * matchongoing (void * arg);
```

The thread arrays, list array and mutex arrays, conditional variable array are as follows :

```
Zone zones[3];
Spectator spectators[SIZE];
Matchdetails goals[SIZE];
Team teams[2];

pthread_t goalthreads[SIZE];
pthread_t spectatorthreads[SIZE];
pthread_t zonethreads[SIZE];

pthread_mutex_t goalsmutex[SIZE];
pthread_mutex_t spectatorsmutex[SIZE];
pthread_mutex_t zonemutex[SIZE];
pthread_mutex_t teammutex[SIZE];

pthread_cond_t spectatorwait[SIZE];
```

A semaphore is used to handle busy waiting in the zoneallocater function.

```
sem_t sem;
```

2. q2.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 spectator threads have exited. (waiting for goalthreads is optional and can be uncommented out.)

```
for(int i=0;i<3;i++) {
    pthread_create(zonethreads+i,NULL,zoneallocater,zones+i);
    pthread_mutex_init(zonemutex+i,NULL);
}
for(int i=0;i<idspec;i++) {
    pthread_create(spectatorthreads+i,NULL,handlespectators,spectators+i);
}
for(int i=0;i<G;i++) {
    pthread_create(goalthreads+i,NULL,matchongoing,goals+i);          // no need mutex cuz same time 2 events cant happen?
}
```

Handler functions are as follows :

2.1 handlespectators :

In this function we initially put the spectators to sleep which is determined by the input, after waking up, we make the spectator available. We also post to the zone allocater function that there are spectators available who are waiting to be allocated.

```
sleep(spec->specsleeptime);
sem_post(&sem);
spec->is_avail=1;              // avail now
```

Now we have 2 cases to deal with, if the person is neutral supporter or a H,A supporter. If the person is a neutral supporter, everything remains same, except that the supporter cannot rage quit. A condition less to be checked.

We will visit the case of H,A fan as the N fan is just a subset of this case.

The first thing that we are supposed to do is to keep the spectator in a conditional wait for a fixed time which is determined by the patience value given during input.

We can do this as follows :

```
struct timespec t;
time_t T;
time(&T);
t.tv_sec=T+spec->patience;
pthread_cond_timedwait(spectatorwait+spec->specid,spectatorsmutex+spec->specid,&t);
```

After the cond_wait() we can have 2 cases, one case is when the cond_signal() from zoneallocater() happened before the patience amount of time, and one case is when it has timed out and the spectator has exhausted his patience limit. Lets have a look at both these cases :

2.1.1 Spectator has gotten a seat in one of the zones :

In this case, the spectator has been signalled from the zone allocator function. In this case the spectators isalloc parameter would have been set by the zone function.

```
spec->isalloc=1
```

Now the spectator has been allocated a seat, we can again have 2 cases now, either the spectator stays and watches the match for X amount of time(given during input) and leave the stadium or watch the match until the opposition team has scored R or more number of goals. We will first put the spectator in another cond_timed_wait() for X seconds. And if it has been interrupted by the matchongoing() function, that would mean the spectator rage quit as opponent team got greater than equal to R goals. the cond_timed_wait() is done as follows :

```
struct timespec t2;
time_t T2;
time(&T2);
t2.tv_sec=T2+spectatingtime;
pthread_mutex_lock(spectatorsmutex+spec->specid);
int x = pthread_cond_timedwait(spectatorwait+spec->specid,spectatorsmutex+spec->specid,&t2);
```

We will have a look at both these cases one by one.

2.1.1.1 Spectator rage quit :

This can be checked by taking cond_timed_wait() into a variable x, and if the value of x is not ETIMEDOUT that means, the spectator rage quit.

```
if(x!=ETIMEDOUT) {
    printf(HRED "Person %s is leaving due to the bad defensive performance of his team.\n",spec->specname,spectatingtime);
    printf(HRED "Person %s is waiting for their friends at the exit.\n",spec->specname);
    spec->is_avail=0;
    pthread_mutex_trylock(zonemutex+spec->zoneno);
    printf(CYAN "Decreasing curr capacity of zone %d\n",spec->zoneno);
    zones[spec->zoneno].curr_capacity--;
    pthread_mutex_unlock(zonemutex+spec->zoneno);
    pthread_mutex_unlock(spectatorsmutex+spec->specid);
    return NULL;
}
```

The spectators is_avail is set to 0 as he is not available anymore. We also decrease the zone capacity of the zone from which the spectator is from.

2.1.1.2 Spectator leaves after X time :

This can be checked by taking cond_timed_wait() into a variable x, and if the value of x is ETIMEDOUT that means, the spectator has left as X seconds have passed.

```
if(x == ETIMEDOUT) {
    printf(HMAG "Person %s has watched the match for %d secs and is leaving.\n",spec->specname,spectatingtime);
    printf(HMAG "Person %s is waiting for their friends at the exit.\n",spec->specname);
    spec->is_avail=0;
    pthread_mutex_trylock(zonemutex+spec->zoneno);
    printf(CYAN "Decreasing curr capacity of zone %d\n",spec->zoneno);
    zones[spec->zoneno].curr_capacity--;
    pthread_mutex_unlock(zonemutex+spec->zoneno);
    pthread_mutex_unlock(spectatorsmutex+spec->specid);
    return NULL;
}
```

The spectators is_avail is set to 0 as he is not available anymore. We also decrease the zone capacity of the zone from which the spectator is from.

These are the 2 cases to be handled when the spectator gets a zone.

2.1.2 Spectator did not get any seat in any of the zones :

This happens if the patience value of the spectator has been exhausted.

In this case isalloc of the spectator would still be 0, thus we can check for this and be sure that the spectator did not get a zone.

```
if(spec->isalloc==0) {
    printf(HRED "Person %s couldn't get a seat.\n",spec->specname);
    printf(HRED "Person %s is waiting for their friends at the exit.\n",spec->specname);
    spec->is_avail=0;
    pthread_mutex_unlock(spectatorsmutex+spec->specid);
    return NULL;
}
```

This is all we need to for the spectators, The zoneallocater function() checks for the suitable and available fans and allocates them the zones.

2.2 zoneallocater :

The zone allocater function is in a while(1) loop, to avoid busy waiting we make the zoneallocater() proceed only when there are spectators available else it gives up the CPU.

```
sem_wait(&sem);
```

We check for each zone for the suitable and available fans and allocate those fans.

We will have a look at zone H for example, other zones follow the same method.

Zone H has a zone id of 0 by the convention and by the question, only H and N fans can get allocated into Zone H.

There are two steps we need to do, we first need to find the spectators that are available and suitable for Zone H and then we need to conditional signal all these spectators at the end.

2.2.1 Finding the suitable fans :

```
pthread_mutex_lock(zonemutex+currzone->zoneid);
int spectatorsqueue[SIZE]; int start=0;
for(int i=0;i<spectators[0].totspec;i++) {
    pthread_mutex_lock(spectatorsmutex+spectators[i].specid);
    if(zones[0].curr_capacity < zones[0].tot_capacity && spectators[i].is_avail==1 && spectators[i].isalloc==0 && (spectators[i].supporting
        printf(YELLOW "%s has got a seat in Zone H\n",spectators[i].specname);
        spectatorsqueue[start++]=spectators[i].specid;
        zones[0].curr_capacity++;
        spectators[i].isalloc=1;
        spectators[i].zoneno=0;
    }
    else {
        pthread_mutex_unlock(spectatorsmutex+spectators[i].specid);
    }
}
// cond signals to patience cond timed wait
pthread_mutex_unlock(zonemutex+currzone->zoneid);
```

We have now acquired the spectators for this zone by checking if the zone can handle more spectators or not, by checking if the spectator is available by is_avail parameter and also by checking if the spectator supports team H or is Neutral.

2.2.2 Signalling all the spectators that have been allocated :

It is to be noted that this cond_timed_signal() will signal cond_timed_wait() for the patience, as isalloc parameter for the spectators just changed to 1.

```
for(int i=0;i<start;i++) {
    pthread_mutex_unlock(spectatorsmutex+spectatorsqueue[i]);
    // printf(BHMAG "signalling %s rn.\n",spectators[spectatorsqueue[i]].specname);
    pthread_cond_signal(spectatorwait+spectatorsqueue[i]);
}
```

Similarly we can do for Zone H and Zone N as well.

2.3 matchongoing :

This function is used to simulate all the goal activities taking place in the match over the course of the whole match.

We initially make the event sleep for the sleep time corresponding to the event that is taken as input.

```
sleep(goal->sleeptime);
```

We check the probability of the goal taking place by random variable check.

```
int team = goal->whichteam;
float p = goal->successprob;
int d = rng(1,100);
int val = (int)(p*100);
```

We now have two cases, one where the success probability is true, implies d≤val (goal scored) and one when the probability is false and the goal is not scored.

Lets have a look at both these cases.

2.3.1 Successful in scoring a goal :

The team that is successful ins coring a goal can be acquired by :

```
int team = goal->whichteam
```

If the team is 0, that means team H has scored a goal else team A has scored the goal, both these cases are equivalent thus , we will have a look at the case when team H has scored the goal.

There are again 2 steps to be followed here, one is to increment the goals scored of team H and acquire all the spectators that have reached their rage capacity, and the other is to signal all these spectators.

2.3.1.1 Acquiring the spectators :

We first acquire the team lock and increase the number of goals scored for this team and then loop through all of the spectators and check if the spectator is still present and is supporting the opponent team and we check if the rage capacity has been reached for this spectator, if these are satisfied we add these spectators into an array.

```
pthread_mutex_lock(&teammutex[0]);
printf(GREENBG "Team H has scored their %d th goal.\n",teams[0].goalsscored+1);
teams[0].goalsscored++;
int spectatorsqueue[SIZE]; int start=0;
for(int i=0;i<spectators[0].totspec;i++) {
    if(spectators[i].isalloc==1 && spectators[i].is_avail==1 && spectators[i].supportingteam==1 && spectators[i].rmax<=teams[0].goalsscored
        pthread_mutex_lock(spectatorsmutex+i);
        spectatorsqueue[start++]=i;
    }
}
```

After this we need to conditional signal all the spectators that have been added into the queue.

2.3.1.2 Signalling all the spectators :

We first unlock the spectators and then signal them to the 2nd pthread_cond_wait() that is waiting for X seconds. Lastly we unlock the team.

```
for(int i=0;i<start;i++) {
    pthread_mutex_unlock(spectatorsmutex+spectatorsqueue[i]);
    pthread_cond_signal(spectatorwait+spectatorsqueue[i]);
}
pthread_mutex_unlock(&teammutex[0]);
```

2.3.2 Unsuccessful in scoring the goal :

In this case we can just print that team missed their chance to which could possibly be their teams[0].goalsscored + 1 th chance.

```
if(team==0) {
    printf(MAG "Team H has missed their chance to score their %d th goal.\n",teams[0].goalsscored+1);
}
else {
    printf(MAG "Team A has missed their chance to score their %d th goal.\n",teams[1].goalsscored+1);
}
```