The Clasico Experience

How to run the program

1. Run the command `gcc -pthread clasico.c`
2. Run `./a.out`
3. Input the details of the zones, spectators and goal chances.

Context - 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. We have created a simulation where people *enter* the stadium, *buy tickets* to a particular *zone* (stand), *watch the match* and then *exit*.

Variables

1. `struct zone` - describes a given zone *Home/Away/Neutral*. It has a single data member - `int capacity`.
2. `zone zone_A, zone_H, zone_N`
3. `num_groups` - total number of groups in the simulation.
4. `num_people` - total number of people in the simulation.
5. `struct spectator` - describes a person who visits the stadium to watch the match during the simulation.
   Data members

```
int id; //index in the match_specs array
char name[MAX];    //person's name
int group_num;     //group id of the group to which the person belongs
char fan_type;     // H/A/N
int entry_time;    // time in secs after which the person enters the
stadium - specific to each person.
int patience_time; //time (in sec) for which he will wait to get a ticket
in any of his eligible zones
int patience_num_goals;
pthread_t thread;   //thread that simulates the spectator
int is_seated;
// 0 - spectator is waiting for a seat
// 1 - spectator got a seat in zone H
// 2 - spectator got a seat in zone A
// 3 - spectator got a seat in zone N
// -1 - spectator is not in the stadium

pthread_mutex_t spec_mutex;
pthread_cond_t get_seat;
pthread_cond_t opponent_goals;
```

1. `int X` - SPECTATING TIME - this value is fixed across all spectators.
2. `struct goal_chance` describes a goal chance

```
    int id;
    char team_str[2]; //input
    char team;        // H/A
    int time_elapsed;
    float prob;       //probability of the goal chance actually converting
into a goal
    pthread_t thread;
```

8. `int total_goal_chances` - total number of goal chances for either of the teams throughout the simulation.
9. `pthread_mutex_t goals_mutex` - acquired and released whenever some thread accesses/modifies `goals_home` or `goals_away`.
10. `int goals_home` and `int goals_away` - total number of goals scored by either of the teams till any moment in the simulation.

Threads

- A thread for each spectator
- A thread for each goal_chance
- A thread `goal_signal`

Locks

- pthread_mutex_t spec_mutex;
- pthread_mutex_t goals_mutex;

Condition Variables

- pthread_cond_t get_seat;
- pthread_cond_t opponent_goals;

Semaphores

- sem_t zoneA_sem;
- sem_t zoneH_sem;
- sem_t zoneN_sem;

Routines

1. In the main thread, we take the input, initialize the locks, CVs and the semaphores and start the simulation. Next we create separate threads for each *spectator* and each *goal chance*, only after all the spectator threads *join back* the main thread, the simulation ends.
2. In the `spectator_thread` (*spectator id* is passed as an argument in pthread_create), the thread sleeps for the time in secs after which the spectator enters the stadium. First we make a check, for the Home and Away fans, if the number of goals of the opponent team is greater than equal to the

number of goals required to enrage the spectator to leave the stadium. If yes, the spectator leaves the stadium, otherwise we call the `get_seat` routine to allocate a seat to the spectator based on whether he is a home fan, away fan or neutral fan. The `get_seat` routine sets the is_seated data member of the spectator structure as 0, if no seat is allocated, 1 if the spectator is allocated a seat in *HOME* zone, 2 in *AWAY* zone, 3 in *NEUTRAL* zone. If after returning from *get_seat*, is_seated $= 0$, it implies that he couldn't get a seat within his *patience time* thus he leaves the stadium. If is_seated > 0, `watch_match()` routine is called, and the spectator watches the match for a time X, with exceptions when he gets enraged due to the bad performance of his team, in case of home or away fans, and leaves the stadium. After the spectator returns from the `watch_match` routine, we use sem_post for the semaphore corresponding to the zone the spectator had seated in.

3. 'In the `get_seat` routine, if the spectator is a **neutral fan**, we create 3 threads,

```
pthread_t away_thread, neutral_thread, home_thread;
pthread_create(&(neutral_thread), NULL ,zone_neutral_thread, (void *)(&
(sid)));
pthread_create(&(home_thread), NULL, zone_home_thread, (void *)(&(sid)));
pthread_create(&(away_thread), NULL, zone_away_thread, (void *)(&(sid)));
```

Each of these threads *timedwait* on the corresponding zone-semaphores as follows

```
    int sid = *(int *)arg;
    struct timespec ts;
    if (clock_gettime(CLOCK_REALTIME, &ts) == -1)
    {
        perror("Error : ");
        return NULL;
    }
    ts.tv_sec = ts.tv_sec + match_specs[sid]->patience_time;
    int s;
    while ((s = sem_timedwait(&zoneN_sem, &ts)) == -1 && errno == EINTR)
        continue;

    if (s == -1)
    {
        if (errno == ETIMEDOUT)
        {
            //printf("sem_timedwait() timed out\n");
            return NULL;
        }
        else
        {

            //perror("sem_timedwait");
            return NULL;
        }
    }
    else
    {
```

```
        //printf("sem_timedwait() succeeded\n");
        pthread_mutex_lock(&match_specs[sid]->spec_mutex);
        if (match_specs[sid]->is_seated == 0)
        {
            match_specs[sid]->is_seated = 3;
            pthread_cond_signal(&match_specs[sid]->get_seat);
        }
        else
            sem_post(&zoneN_sem);
        pthread_mutex_unlock(&match_specs[sid]->spec_mutex);
    }
```

sem_timedwait waits on the semaphore for the *patience time* of the spectator, returns -1 and $errno = ETIMEDOUT$ in case no seat was available in that zone within the patience time, if a seat was made available within that time, sem_timedwait returns 0, in this case we check if the spectator hasn't been allocated yet, as 3 threads are simultaneously running with the same objective, since multiple threads can access and modify this data member of the spectator structure, we use the `spec_mutex`. If the spectator wasn't allocated any seat yet, we modify `is_seated`, signal the spectator_thread (currently waiting in `get_seat` routine). In case sem_timedwait returns 0 and spectator was already allocated a seat we call sem_post as the spectator did not take that seat. In the get_seat we use `pthread_cond_timedwait` again as in case none of the threads signal it, it is not stuck.

```
 pthread_t away_thread, neutral_thread, home_thread;
struct timespec ts;
if (clock_gettime(CLOCK_REALTIME, &ts) == -1)
{
    perror("Error : ");
    return -1;
}
ts.tv_sec = ts.tv_sec + match_specs[sid]->patience_time;

pthread_create(&(neutral_thread), NULL, zone_neutral_thread, (void *)(&
(sid)));
pthread_create(&(home_thread), NULL, zone_home_thread, (void *)(&(sid)));
pthread_create(&(away_thread), NULL, zone_away_thread, (void *)(&(sid)));
pthread_mutex_lock(&match_specs[sid]->spec_mutex);
pthread_cond_timedwait(&match_specs[sid]->get_seat, &match_specs[sid]-
>spec_mutex, &ts);
pthread_mutex_unlock(&match_specs[sid]->spec_mutex);
```

As we did for a neutral fan, similarly we create two threads for a *home* fan (pthread_t neutral_thread, home_thread) and a single thread for *away* fan.

4. In the `watch_match` routine, in case the spectator is a `neutral fan`, we call sleep(X), X is the spectating time - common across all spectators, in case of home/away fans,

```
struct timespec ts;
if (clock_gettime(CLOCK_REALTIME, &ts) == -1)
{
    perror("Error : ");
    return -1;
}
ts.tv_sec = ts.tv_sec + X;

int rc;
pthread_mutex_lock(&match_specs[sid]->spec_mutex);
rc = pthread_cond_timedwait(&match_specs[sid]->opponent_goals,
&match_specs[sid]->spec_mutex, &ts);
pthread_mutex_unlock(&match_specs[sid]->spec_mutex);

if (goals_away >= match_specs[sid]->patience_num_goals)
    printf(BLUE "%s is leaving due to the bad performance of his team\n",
match_specs[sid]->name);
else
    printf(BLUE "%s watched the match for %d seconds and is leaving\n",
match_specs[sid]->name, X);
```

We use `cond_timedwait` on the CV - `opponent goals`, for X time.

5. We create a thread other than the spectator and goal_chances threads which run continuously through the simulation,

```
 while (simulation == 1)
    {
        for (int i = 0; i < num_people; i++)
        {
            char fan = match_specs[i]->fan_type;

            if (match_specs[i]->is_seated > 0) // possible values for
is_seated = 0, 1, 2, 3, -1
            {
                switch (fan)
                {
                case 'H':
                    pthread_mutex_lock(&goals_mutex);
                    if (goals_away >= match_specs[i]->patience_num_goals)
                        pthread_cond_signal(&match_specs[i]-
>opponent_goals);
                    pthread_mutex_unlock(&goals_mutex);
                    break;
                case 'A':
                    pthread_mutex_lock(&goals_mutex);
                    if (goals_home >= match_specs[i]->patience_num_goals)
                        pthread_cond_signal(&match_specs[i]-
>opponent_goals);
                    pthread_mutex_unlock(&goals_mutex);
                    break;
```

```
                case 'N':
                    break;
                default:
                    break;
                }
            }
        }
    }
```

it traverse through the spec_list and checks if for any spectator who is a home/away fan and currently watching the match (is_seated> 0), and signals `opponent_goals` in case opponent_goals >= patience goals for that spectator.

6. In the `goal_chance_thread`, sleep for the time in secs until the goal chance is created. Then according to the random result as of whether the chance was converted into a goal or not, we print the corresponding statements and update `goals_away` and `goals_home`. Since these variables are accessed and modified by multiple threads, we use `goal_mutex`.
