Football Match Simulation

- In the problem statement , we'd to simulate a football match , between 2 teams `Home` and `Away` using threads and related concepts.

Main Logic Flow of the program

- We first introduce our entities to the playground , by creating threads for the goals and the people coming in the struct.
- For maintaining the seat count of the stand , I have used `semaphores`
  - **H** : for home stand ,
  - **A** for away stand and
  - **N** for neutral stand fans
- I have simulated goals and people incoming using `threads`

Working of Goals

- **Goals** : Simulated the goals as threads , created the goal object as :

```
struct goals
{
    char team[2];     // "H" or "A"
    int time_elapsed; // time from start after which chance is created
    float chance;     // probability for conversion to a goal
    pthread_t g_thread;
};
```

- The total goals scored by Home team was stored as `goals_home` associated with a mutex lock `h_goals` , and corresponding variables `goals_away` and `a_goals` for the away team. Locks were created so as to ensure consistency while updating the goals amongst the many goal threads created by us.
- As a goal thread came to life , I calculated random probability using the random number generator as : (float)rand()/RAND_MAX and compared it with chance if

`chance >= random_probability` , goal was scored and the variables were updated for the team corresponding to the `team` of the goal entity and corresponding messages were printed .

Working of People

Structure of people object

- **Groups** groups of people , basically treated as a **resource** , were creates as :

```
struct group
{
    int id;                       // store the group number
    int num_people;               // number of people in a group
    struct person people[MAX_LEN]; //people details
```

```
    int exit_count;               // store the number of people exiting the gate
    pthread_cond_t wait_at_gate;  // for people waiting at exit gate ( waiting for their friends )
};
```

- **People** were simulated using threads :

```
struct person
{
    char name[MAX_LEN]; // name of the person
    int reach_time;     // time taken to reach the stadium
    char zone[2];       // "H" , "N" , "A"    // the side
    int patience;       // patience
    int num_goals;      // number of goals , before it leaves  the game

    int status;              // status , could be one of those #defines ( evident from their name what they mean )
                             // depict current status of the thread
    int group_id;            // group id of the people object
    int id;                  // id [0 - max_people_in group -1 ]
    int counter;             // counter variable  ( usage explained in README ) - check people object part
    int zone_allocated;      // the zone allocated
                             // if person is from Home team it could be HOME or NEUTRAL
                             // if A then could only be AWAY
                             // if N could be any of AWAY , NEUTRAL , or HOME
    struct timespec *t;      // for implementing the thread timed wait
    bool is_allocated;       // boolean* expression , so that the person isnt allocated multiple seats simultaneously
    pthread_t p_thread;      // thread object
    pthread_mutex_t mutex;   // lock for critical section
    pthread_cond_t cond_var; // cond_var for avoiding busy sleeping while ,
                             //1. thread is looking for seat
                             // 2. thread is watching match
};
```

Below are the states and other definitions for making the code easy to read and write 😸

```
// possible states for the status variable
#define WAITING 0  // waiting for seat to be allocated
#define SEATED 1 // if watching the match
#define AT_EXIT_GATE 2 // if waiting at the exit gate
#define EXITS 3 // if exits the simulation
#define BAD_PERFORMANCE 4 // if exits due to bad performance

// used to depict the zone variable
// preffered this way , as its easy to compare integers than strings in C xD :)
#define HOME 10
#define AWAY 20
#define NEUTRAL 30
```

- For the people thread , the work could be divided into 3 sub- parts :
    1. **Looking for Seat** :
        a. For this part , depending on the group a person belongs (Home , Away or Neutral) and corresponding allowed seats i.e. Home person could only be allotted seats in Home or Neutral

section , and Away person could only given seats in Away stands and similar for neutral which could be seated anywhere . **I create multiple threads** depending on allowed seats

b. The main thread (the person thread) goes to sleep using `pthread_cond_wait` on the conditional variable `cond_var` described before , after creating the corresponding threads u.

c. The multiple threads created , waited for the seats to be allocated (I used `sem_timedwait` for this , so as the person doesnt loses his **patience** before its allocated the seat) , and signal the waiting thread sleeping on the conditional variable , using `pthread_cond_signal`

d. **Counter** : This has the value equal to the number of possible areas , the person could be seated , say

   i. for home fan counter = 2 (H or N)

   ii. for away fan counter = 1 (only A)

   iii. and for Neutral fan , counter = 3 (H or A or N)

   Now the main person thread goes into wait condition like :

```
while (spectator->counter > 0 && spectator->is_allocated == 0)
{
  pthread_cond_wait(&spectator->cond_var, &spectator->mutex);
}
```

- After a the multiple threads , we created , finish their timed value (exceed the patience mark) we signal the thread , after decrementing the **counter** once , the counter is 0 , it means all the threads have completed their part as seat was not allocated to the person and we can continue with further actions.
- However if , one of the thread is able to find seat for the person , it signals the waiting thread after initializing the the variable `is_allocated` to 1 , and the person gets the seat allocated. and exits the wait to further continue watching the match.
- **Case Handling** : Multiple seats being allocated to a single person ,
  - if one of the multiple thread is able to find the seat for a user , it acquires the lock of the person **mutex** and converts the variable **is_allocated** to 1 , which is checked in other threads , and of the other threads also allocates the seat to the person (by decrementing the semaphore for the corresponding seat) , we increment the semaphore using `sem_post` as shown in the code below :

```
int ret = sem_timedwait(&N, spectator->t);

    if (ret == -1)
    {
        pthread_mutex_lock(&spectator->mutex);
        spectator->counter--;
        pthread_mutex_unlock(&spectator->mutex);
        pthread_cond_signal(&spectator->cond_var);
        sem_getvalue(&N, &x);
        return NULL;
    }
    // if the person was already allocated seat by some other thread
```

```
    pthread_mutex_lock(&spectator->mutex);
    if (!spectator->is_allocated)
    {
        spectator->is_allocated = true;
        spectator->status = SEATED;
        spectator->zone_allocated = NEUTRAL;
        pthread_mutex_unlock(&spectator->mutex);
        pthread_cond_signal(&spectator->cond_var);
        sem_getvalue(&N, &x);
        return NULL;
    }
    else
        sem_post(&N); // if already allocated , then dont allot the seat
    pthread_mutex_unlock(&spectator->mutex);
```

  - **The case when no seat is allocated** if none of the threads are able to give the seats , the counter value if decremented and the thread is signaled 👍 accordingly.

- **Seat Allocated** ,after the seat is allocated the person goes onto timed sleep for time duration given by `X` using `pthread_cond_timedwait` on the conditional variable `cond_var` defined above and exits the simulation.

- **Goal being greater than Performance benchmark** : I made a separate thread that runs all through the simulation `signal_thread` that compared the status of the person and the number of goals to the benchmark set by the person. and signals the thread waiting while watching the match (the signal is only sent to threads with status as `Watching` or `Waiting` . Signal is sent after changing the status to `BAD_PERFORMANCE` .