OSN Assignment 5

Shreyash Jain

2020101006

Report : Question 2 (with bonus)

- This problem can be broadly broken down into 2 parts, one from the spectators viewpoint and one in terms of the goals in the match.
- So I create `num_spectators` threads which handle the routine of every spectator via the function `spectator_routine` which takes the spectator structure of its respective spectator as an argument.
- Then there is one thread `goals` which handles the goals scored in the match as per the goal scoring chances via the function `goals_routine` which takes the array of structure goalscoring_chance as an argument.
- I have also `num_groups` threads called `groups_exit[]` which handles the exiting of a group in the simulation (part of bonus). It uses `group_exit_routine` which takes its respective groups, group element as an argument.

A. Spectator Routine

This can be further broken down to following sections :

1. Arrival of spectator
2. Spectator waiting for a seat
3. Spectator watching the match
4. Spectator at exit gate

Arrival of spectator

- For this, I make the thread corresponding to its spectator sleep till the time of arrival of the spectator. Once, sleep finishes, we can assume that the spectator has arrived.

```
sleep(s.time_of_arrival);
printf(ANSI_COLOR_RED "t=%ld : %s has reached the stadium\n" ANSI_COLOR_RESET, time_from_start(), s.name);
```

Spectator waiting for a seat

- For this I make use of the function, `sem_timedwait` (refer to man pages) which takes a semaphore and timespec structure as an argument. I used the timespec structure to store the patience limit of the spectator waiting for their seat.

```
    struct timespec ts;
    if (clock_gettime(CLOCK_REALTIME, &ts) == -1)
    {
        perror("clock gettime");
        exit(EXIT_FAILURE);
    }

    ts1.tv_sec += s.patience;
```

- `sem_timedwait` waits for the counter to go above zero till a specified time (set via the timespec struct). If the function returns 0, that means it received `sem_post` before timeout. But if the function returns -1 and `errno == ETIMEDOUT`, that means it wasn't able to receive `sem_post` by the specified time.
- As we know, Home Fan → Home, Neutral seats
- Neutral Fan → Home, Neutral, Away seats
- Away Fan → Away seats

- Since H can access from a pool of H,N seats, N from H,N,A seats and A from only A seats, I created three semaphores :
  - HN for Home fan
  - HNA for Neutral fan
  - A for Away fan

```
sem_init(&HN, 0, Hzone.capacity + Nzone.capacity);
sem_init(&HNA, 0, Hzone.capacity + Nzone.capacity + Azone.capacity);
sem_init(&A, 0, Azone.capacity);
```

- Now for a Home fan,
  - I use sem_timedwait on HN and wait till its patience limit to receive a post on it.
  - If a timeout occurs, I send the fan to the exit gate.
  - If it receives a post before timeout, I lock mutexes H_lock and N_lock as I will now access the capacity of Zones H and N.
  - If both have seats available, I'll chose randomly which seat to occupy.
  - If only one seat is available, I'll choose that one only.
  - After selecting the seat , I increment the value of currently filled seats in that zone and also do `sem_wait` on another semaphore.
    - This is because if I occupy seat H from the pool H and N (available to a Home fan), I also have to decrease the counter for the pool H,N,A as it also has seats for H zone. Thus, I do `sem_wait(&HNA)`
  - Then I unlock the mutexes and move to the next stage.
- Similar steps are done for Neutral and Away fans (only different pool of seats).

Spectator watching the match

- Here I use `pthread_cond_timedwait` which is another function that makes use of struct timespec to set timeout time to `spectating_time` of the spectators. Returns 0 if successful, else `errno`
- I define two condition variables `Agoal_cond` and `Hgoal_cond`, locked by `Agoal_lock` and `Hgoal_lock` for away and home goals respectively.
- For Spectator Type H,

```
pthread_mutex_lock(&Agoal_lock);
int t0 = time_from_start();
while (A_goals < s.enrage_limit && j != ETIMEDOUT)
   j = pthread_cond_timedwait(&Agoal_cond, &Agoal_lock, &ts2); // see if away goal is scored
int t1 = time_from_start();
pthread_mutex_unlock(&Agoal_lock);
```

- I wait for a signal on the condition variable from thread `goals`, and check in a while loop if it has timedout or the goals by opponent team is still less that enrage_limit of the spectator.
- Once it breaks out of the loop, I also take an account of the time it stayed in the loop and see if it is greater than the timeout limit.
  - This is done because at times I will get a signal on my cond_variable but still it might not break out of the loop if goals are less than enrage limit of the spectator. Still a timeout, just not shown by j == ETIMEDOUT

```
if ((t1 - t0) >= spectating_time || j == ETIMEDOUT)
{
  // Timed out
  // stands up from his/her seat
  s.is_seated = 0;
  printf(ANSI_COLOR_GREEN "t=%ld : %s watched the match for %d seconds and is leaving\n" ANSI_COLOR_RESET, time_from_start(), s.na
}
else
```

```
{
  // Gets Enraged
  printf(ANSI_COLOR_GREEN "t=%ld : %s is leaving due to bad performance of his team\n" ANSI_COLOR_RESET, time_from_start(), s.name
  s.is_seated = 0;
}
```

- Similarly it is done for Away spectator. But for a neutral spectator (no enrage limit), I just sleep till its spectating time and send him to exit gate.
- For vacating his/her seat,

```
if (s.seated_zone == 'H')
{   // is part of pools HN and HNA
    pthread_mutex_lock(&H_lock);
    Hzone.currently_filled--;
    pthread_mutex_unlock(&H_lock);
    sem_post(&HN);
    sem_post(&HNA);
}
else if (s.seated_zone == 'N')
{   // is part of pools HN and HNA
    pthread_mutex_lock(&N_lock);
    Nzone.currently_filled--;
    pthread_mutex_unlock(&N_lock);
    sem_post(&HN);
    sem_post(&HNA);
}
else
{   // is part of pool A
    pthread_mutex_lock(&A_lock);
    Azone.currently_filled--;
    pthread_mutex_unlock(&A_lock);
    sem_post(&A);
    sem_post(&HNA);
}
```

Spectator at exit gate

- I make use of mutex locks `group_lock[s.group_id]` for each group of spectators. And also conditional variables `group_cond[s.group_id]` to signal an update in the spectators at exit for a particular group to the `groups_exit` thread.

```
// Exit the stadium
s.is_at_exit = 1;
printf(ANSI_COLOR_BLUE "t=%ld : %s is waiting for their friends at the exit\n" ANSI_COLOR_RESET, time_from_start(), s.name);
pthread_mutex_lock(&group_lock[s.group_id]);
group_members_at_exit[s.group_id]++;
// Signals the exit_routine
pthread_cond_broadcast(&group_cond[s.group_id]);
pthread_mutex_unlock(&group_lock[s.group_id]);
```

B. Goals Routine

- In this I iterate through the goalscoring_chance array and sleep till the time for the next goalscoring chance has come.

```
int current_time = time_from_start();
int time_to_wait = g[i].time_since_start - current_time;
if (time_to_wait > 0)
    sleep(time_to_wait);
```

- Now when the chance has come, I generate a random float number between 0 and 1 (inclusive) and use it as probability. If it is less than or equal to the probability of goal to occur, a goal is given.
- I check the team which has the chance and give it a goal if it satisfies the probability. Then I also broadcast a signal to its respective condition variable.

```
if (probability <= g[i].probability)
{
    if (g[i].type == 'H')
    {
        pthread_mutex_lock(&Hgoal_lock);
        H_goals++;
        pthread_cond_broadcast(&Hgoal_cond);
        pthread_mutex_unlock(&Hgoal_lock);
    }
    else
    {
        pthread_mutex_lock(&Agoal_lock);
        A_goals++;
        pthread_cond_broadcast(&Agoal_cond);
        pthread_mutex_unlock(&Agoal_lock);
    }
    printf("t=%ld : Team %c has scored their %s goal\n", time_from_start(), g[i].type, key);
}
else
{
    printf("t=%ld : Team %c missed the chance to score their %s goal\n", time_from_start(), g[i].type, key);
}
```

- I keep iterating till all chances are over.

C. Group Exit Routine

- I make use of mutex locks and conditional variables for each group to check and wait for all the members of a particular group to reach the exit gate.
- Once all group members are at exit, I break out of the loop and make the group leave for dinner.

```
// Wait for all spectators of a group to leave
pthread_mutex_lock(&group_lock[g->group_id]);
while (group_members_at_exit[g->group_id] < g->group_size)
    pthread_cond_wait(&group_cond[g->group_id], &group_lock[g->group_id]);
pthread_mutex_unlock(&group_lock[g->group_id]);

// Now the time has come
printf(ANSI_COLOR_YELLOW "t=%ld : Group %d is leaving for dinner\n" ANSI_COLOR_RESET, time_from_start(), (g->group_id + 1));
```

Main function

- In the main function I take inputs, initialize threads, locks, condition variables and semaphores.
- I create threads and join them.
- I also free the pointers (used for passing args to threads) allocated via malloc.