Question 3 Report

Operating Systems

Kushagra Agarwal
2018113012

I used the following logic to implement Q3:

Every performer calls 1 thread/ 2 threads depending upon the stages it can perform on. If it can perform only on acoustic stages then it only calls the acoustic stage thread but if it can perform on both then 2 threads are called(one for acoustic stage acquisition and the other for the electric stage acquisition).

I have not implemented a singer as a separate identity. For me, the singer is a performer who can perform on both stages like the pianist and the guitarist.

Now, both the acoustic and the electric threads are very similar and therefore explanation for one is enough to understand the other.

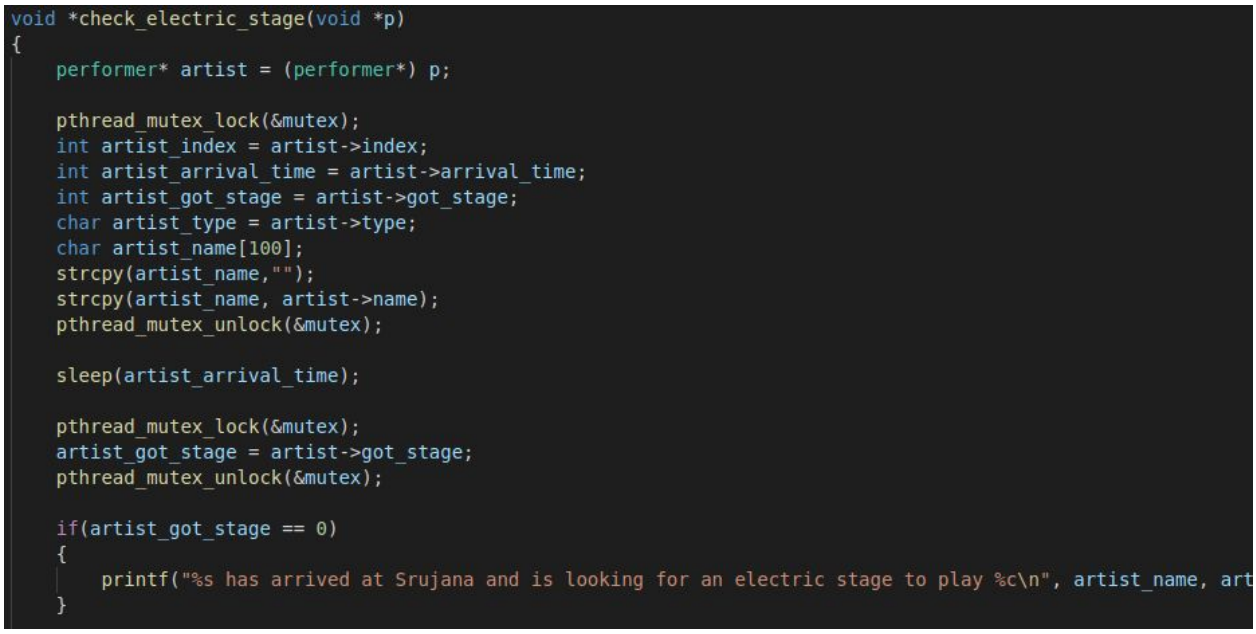
I sent the details about each performer while calling the threads. The details for each performer included are as follows:

```
typedef struct performer
{
    int index;
    char name[100];
    char type;
    int arrival_time;
    int got_stage;

}performer;

performer performers[100];
```

Name stores the name, type stores the music instrument it plays/ sings. Arrival_time stores the time, when it arrived at Srujana and

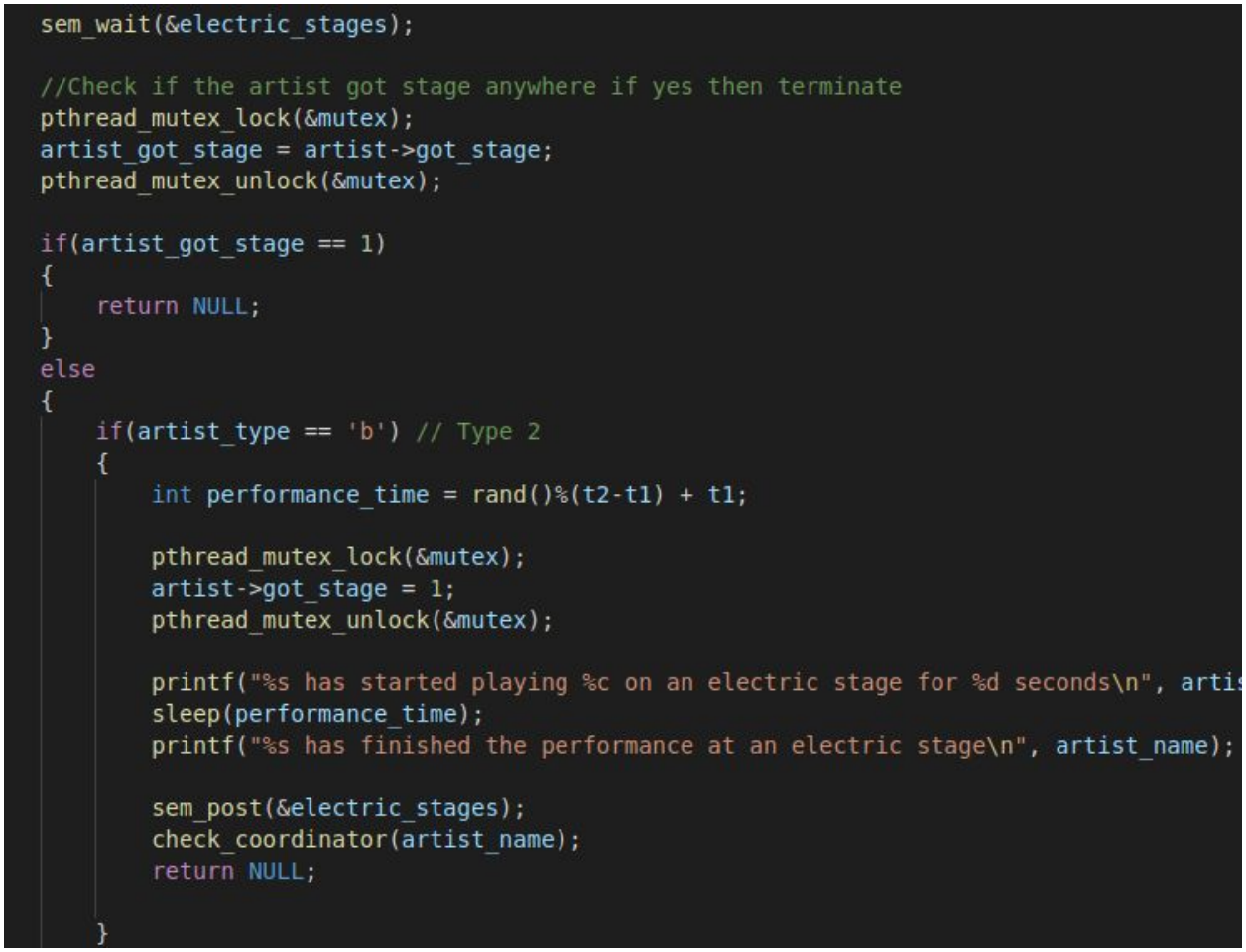
got_stage, is a flag which lets us know if the performer got a stage yet or not. This is particularly helpful when for example a performer who can play at both stages is waiting for the acoustic stage to be empty while it has already performed at the electric stage. Therefore while waiting for the stage, the thread checks if the performer got a stage anywhere. If it did then the wait is halted and NULL is returned.

The explanation for electric thread starting:



```
void *check_electric_stage(void *p)
{
    performer* artist = (performer*) p;

    pthread_mutex_lock(&mutex);
    int artist_index = artist->index;
    int artist_arrival_time = artist->arrival_time;
    int artist_got_stage = artist->got_stage;
    char artist_type = artist->type;
    char artist_name[100];
    strcpy(artist_name,"");
    strcpy(artist_name, artist->name);
    pthread_mutex_unlock(&mutex);

    sleep(artist_arrival_time);

    pthread_mutex_lock(&mutex);
    artist_got_stage = artist->got_stage;
    pthread_mutex_unlock(&mutex);

    if(artist_got_stage == 0)
    {
        printf("%s has arrived at Srujana and is looking for an electric stage to play %c\n", artist_name, art
    }
```

At first, I lock the mutex to access the values stored in the performer struct. This allows the critical sections to be secured from data manipulation attacks. Now the thread sleeps for (arrival_time) number of seconds to imitate the performer arriving at Srujana at arrival_time. I check if the artist got a stage. If it has not then we say that the artist has arrived at Srujana and is waiting for allocation to an electric stage.



```
sem_wait(&electric_stages);

//Check if the artist got stage anywhere if yes then terminate
pthread_mutex_lock(&mutex);
artist_got_stage = artist->got_stage;
pthread_mutex_unlock(&mutex);

if(artist_got_stage == 1)
{
    return NULL;
}
else
{
    if(artist_type == 'b') // Type 2
    {
        int performance_time = rand()%(t2-t1) + t1;

        pthread_mutex_lock(&mutex);
        artist->got_stage = 1;
        pthread_mutex_unlock(&mutex);

        printf("%s has started playing %c on an electric stage for %d seconds\n", artis
        sleep(performance_time);
        printf("%s has finished the performance at an electric stage\n", artist_name);

        sem_post(&electric_stages);
        check_coordinator(artist_name);
        return NULL;

    }
```

As can be seen from above, I wait for the electric_stages semaphore. This is like waiting for a stage to be empty so that our performer can enter the stage and play. We again check if the artist has already gotten the stage somewhere else. If yes, exit and return NULL. But if not then we check for the artist type. If let's say the artist type is a bassist (b), then we first calculate the performance time subjection to the constraints (time>=t1 and time<=t2) using rand() function. We also update the got_stage variable for our performer to 1 as it has gotten a stage. Then we print that the performance is starting. We start the performance

using the sleep(performance time) function. Once the performance ends, we increment the value of the semaphore electric_stage to allow any other waiting singer to come and perform. It then calls the check_coordinator() function to go and collect the T-shirt after having performed. Once it returns from that function it also returns from the current function using return NULL.

```
void *check_coordinator(char name[])
{

    sem_wait(&coordinators);

    printf("%s tshirt collection process begins\n", name);
    sleep(2);
    printf("%s says Good-bye!!!\n", name);
    sem_post(&coordinators);

    return NULL;

}
```

In the check_coordinator function, we first wait for the semaphore coordinator, this is like waiting for a coordinator to be free to give us the T-shirt. Once a coordinator is free, we print that the T-shirt collection process has started and wait for 2 seconds (using sleep(2)) to give the coordinator time to find the right-sized T-shirt for us. Then goodbye is printed and the performer leaves, and increments(signals) the coordinator semaphore again.

```c
struct timespec ts;
if (clock_gettime(CLOCK_REALTIME, &ts) == -1)
{
    exit(0);
}
ts.tv_sec += t;

int s = sem_timedwait(&electric_stages, &ts);
if (s == -1)
{
    if (errno == ETIMEDOUT)
    {
        printf("%s left in frustation after %d seconds due to a long waiting time\n", artist_name, t);
        return NULL;
    }

}
```

I used the sem_timedwait function to implement maximum time to wait for the threads. If the sem_timedwait returns -1 which it can either due to excess waiting or sys eros then, I check if it is caused by the excess waiting. If this is the case then I exit saying that the performer left as he got frustrated.
