Musical Mayhem

Logic

In my logic,I define acoustic stages, electric stages, solo performances where singers can join as seperate resources. There are semaphores representing them. Initially the acoustic, electric , solo performances semaphore is set to number of acoustic stages, electric stages, 0 respectively. Now I conduct a race. Each Perfomer spawns threads that race for each resource that he is allowed to use. For example, a singer spawns three versions of himself and they all race for the three seperate resources. But only one wins, that is acquires the resource first. This version will perform and do all the printing. The other versions leave the scene without doing anything. Now I also have a semaphore representing coordinates, the performers including the singer wait on them in seperate threads to collect t-shirts. **The stages have ids and Singers also collect T-Shirts. BONUS xD**

Code Breakdown with Explanation

Performer Thread

This function is what is called from the main for each performer. This function checks the type of performer, sleeps to simulate arrival and spawns different versions and conducts the race and waits for the winner and others to do their stuff and joins them back and returns.

```
void *PerformerThread(void *performerargs)//function that spawns different
versions of the same performer who race for different resources
{
    int id = ((struct Performer *)performerargs)->id;//get the id
    sleep(performer[id].arrival_time);//sleep till arrival time
    printf(ARRIVAL_STRING,performer[id].name,performer[id].instr);
    //spawn join free acoustic stage thread,join with performer thread,
join electric stage thread according to types
    pthread_mutex_lock(&performer[id].lock);
    char i= performer[id].instr;
    pthread_mutex_unlock(&performer[id].lock);
    pthread_t acousticthread;
    pthread_t electricthread;
    pthread_t joiningthread;
    if(i=='p'||i=='g'){

pthread_create(&acousticthread,NULL,joinFreeAcousticStage,&performer[id]);

pthread_create(&electricthread,NULL,joinFreeElectricStage,&performer[id]);
        pthread_join(acousticthread,NULL);
        pthread_join(electricthread,NULL);
    }
    else if(i=='v'){

pthread_create(&acousticthread,NULL,joinFreeAcousticStage,&performer[id]);
        pthread_join(acousticthread,NULL);
```

```
    }
    else if(i=='b'){

pthread_create(&electricthread,NULL,joinFreeElectricStage,&performer[id]);
        pthread_join(electricthread,NULL);

    }
    else if(i=='s'){

pthread_create(&acousticthread,NULL,joinFreeAcousticStage,&performer[id]);

pthread_create(&electricthread,NULL,joinFreeElectricStage,&performer[id]);
        pthread_create(&joiningthread,NULL,joinPerformer,&performer[id]);
        pthread_join(acousticthread,NULL);
        pthread_join(electricthread,NULL);
        pthread_join(joiningthread,NULL);
    }

    return NULL;
}
```

joinFreeElectricStage

The code is commented extensively for clarity This is the function where the electric version of performer attempts to acquire a free electric stage in the race. Here the performer does a timed wait on the electric semaphore. If the wait returns -1 it means the wait timed out. `if(performer[id].status!=LEFT&&performer[id].status==STAGE_NOT_ASSIGNED)` then performer has not left yet and has not been assigned a stage, that means he/ she is impatient and should leave now, so they leave with a print and we keep this performers status as `LEFT`. `if(performer[id].status==LEFT||performer[id].status==STAGE_ASSIGNED)` this means this performer left in another thread or he has been assigned a stage in another thread just leave this thread, this version lost the race. We also increase semaphore because this version is not going to perform. If we proceed further from this point, that means this version won the race, so there must be an electric stage that is free since we acquired the electric semaphore.So we iterate through the array of stages once to get the stage id of the stage that is free and is electric such a stage will always exist if we reach here. This is not busy waiting. We select the stage and update everything appropriately with locks. We increase soloperformances if this performance is a performance where a singer can join. We use mutex for the stage and performer for concurrency. The performance now starts we get tdash, the time of the performance that is between t1, t2 and we sleep to simulate the performance with `sleep()`.After sleep I check if a singer joined me then, I sleep again for 2s because the performance got extended. We reset the stage and move onto collect T-Shirts. For T-Shirt Collection we create a new thread for the main performer and Singer (to implement bonus) so that simulataneous T-Shirt Collection can happen. For this we run a thread with `collectTShirts(void *performerargs)`

```
void *joinFreeElectricStage(void *performerargs) //the function where the
electric version of performer attempts to acquire a free electric stage
{
    int id = ((struct Performer *)performerargs)->id;//get the id
```

```
    //clock for timed wait
    struct timespec ts;
    clock_gettime(CLOCK_REALTIME, &ts);
    ts.tv_sec += t;
    int returnValue = sem_timedwait(&electric, &ts);
    pthread_mutex_lock(&performer[id].lock);// lock this performer,
critical section
    if(returnValue==-1){ // if -1 it means timed out.

if(performer[id].status!=LEFT&&performer[id].status==STAGE_NOT_ASSIGNED){
// if the performer has not left yet and has not been assigned a stage,
that means he/ she is impatient and should leave now
                performer[id].status=LEFT;//The performer will leave, so
update his status

printf(LEAVING_AFTER_TIME_OUT,performer[id].name,performer[id].instr);
//print the statement

        }
        pthread_mutex_unlock(&performer[id].lock);//unlock to avoid
deadlocks
        return NULL;

    }
    if(performer[id].status==LEFT||performer[id].status==STAGE_ASSIGNED){
//if this performer left in another thread or he has been assigned a stage
in another thread just leave this thread, this version lost the race
        sem_post(&electric);//increase the number of electric stages back,
since we decreased it before and now we are leaving
        pthread_mutex_unlock(&performer[id].lock);//unlock the performer to
avoid deadlocks
        return NULL;
    }

    //if we reach this point, that means this version won the race, so
there must be an electric stage that is free since we acquired the electric
semaphore
    //so we iterate through the array of stages once to get the stage id of
the stage that is free and is electric
    //such a stage will always exist if we reach here. this is not busy
waiting
    for(int i=0 ; i< num_of_stages;i++){
        pthread_mutex_lock(&stage[i].lock);//lock for concurrency
        if(stage[i].type==ELECTRIC&&stage[i].state==FREE){ //If this stage
is electric and is free this performer can take this stage
            performer[id].status=STAGE_ASSIGNED; //the performer is
assigned this stage
            performer[id].stageid=i;
            stage[i].performerid=id; //the stage is assigned this performer
            if(performer[id].isSinger){
                stage[i].state=SINGER_CANNOT_JOIN_PERFORMANCE_GOING_ON;
//if this performer is a singer, then another singer cannot join
            }
            else{
```

```
                stage[i].state=SINGER_CAN_JOIN_PERFORMANCE_GOING_ON;
//otherwise another singer can join
                sem_post(&soloPerformancesWhereSingerCanJoin);// so
increase the number of SoloPerformancewheresingerscanjoin semaphore
            }
            pthread_mutex_unlock(&stage[i].lock);//unlock the mutex before
break to avoid deadlocks
            break;

        }
        pthread_mutex_unlock(&stage[i].lock);//unlock in case it didnt
enter if condition
    }
    pthread_mutex_unlock(&performer[id].lock);//unlock the performer
    // the performance now starts we get tdash, the time of the performance
that is between t1, t2 and we sleep to simulate the performance
    int tdash=getRandom(t1,t2);

printf(START_PERFORMANCE_ELECTRIC,performer[id].name,performer[id].instr,pe
rformer[id].stageid,tdash);
    sleep(tdash); //now sleep for some time between t1 and t2 to simulate
performance
    pthread_mutex_lock(&performer[id].lock); //lock performer again for
concurrency
    pthread_mutex_lock(&stage[performer[id].stageid].lock);//lock this
stage again for concurrenyy
    if(performer[id].Singerid!=-1){ //this means a singer joined this
performer during his performace so the performance got extended by 2 secs
        //so we extend by 2s through sleep(2)
        pthread_mutex_unlock(&stage[performer[id].stageid].lock); //unlock
before sleep, so others can access
        pthread_mutex_unlock(&performer[id].lock);
        sleep(2);
        pthread_mutex_lock(&performer[id].lock); //lock again after sleep
for concurrency
        pthread_mutex_lock(&stage[performer[id].stageid].lock);
    }

    if(performer[id].Singerid==-1&&!performer[id].isSinger){ //if no singer
joined me and I am not a singer we should reduce the number of solo
performances
        int returnvalue=sem_trywait(&soloPerformancesWhereSingerCanJoin);
//this will not go into waiting as we increased 1 in the negation of this
and no singer joined us, so it should be atleast 1.
        if(returnvalue==-1){ //if it failed, I am the only hope for the
singer who decremented the semaphore.. He will join me.
            pthread_mutex_unlock(&stage[performer[id].stageid].lock);
//unlock before sleep, so others can access
            pthread_mutex_unlock(&performer[id].lock);
            sleep(2);//let him join.. I will simulate anyways..
            pthread_mutex_lock(&performer[id].lock); //lock again after
sleep for concurrency
            pthread_mutex_lock(&stage[performer[id].stageid].lock);
        }
```

```
    }
    resetStage(performer[id].stageid);//reset the stage, we are leaving
    sem_post(&electric);//increase electric stages because we are leaving
and this stage now becomes free electric stage
    performer[id].status=LEFT; //this performer left

printf(END_PERFORMANCE_ELECTRIC,performer[id].name,performer[id].stageid);
    if(performer[id].Singerid!=-1){ //if a singer is also there with this
perfomer
        pthread_mutex_lock(&performer[performer[id].Singerid].lock); //lock
the singer
        performer[performer[id].Singerid].status=LEFT;//if this stage had
an extra singer then that guy also left

printf(END_PERFORMANCE_ELECTRIC,performer[performer[id].Singerid].name,perf
ormer[id].stageid);

pthread_mutex_unlock(&performer[performer[id].Singerid].lock);//unlock the
singer
    }
    pthread_mutex_unlock(&stage[performer[id].stageid].lock); //unlock
stage
    pthread_mutex_unlock(&performer[id].lock); //unlock performer

    //now collection of tshirts occurs
    if(c==0)goto leave;//if there are no coordinators leave- sedlyf no
tshirts
    pthread_t MainTthread;// thread for main performer
    pthread_t SingerTthread;// thread for joined singer
    if(performer[id].Singerid!=-1){// if there is a singer with the main
performer - BONUS xD
        pthread_create(&MainTthread,NULL,collectTShirts,&performer[id]);
//create a thread where main performer will collect T-Shirt

pthread_create(&SingerTthread,NULL,collectTShirts,&performer[performer[id].
Singerid]);//create a thread where Co performing Singer will collect T-
shirt.
        pthread_join(MainTthread,NULL);// join them
        pthread_join(SingerTthread,NULL);

    }
    else{
        pthread_create(&MainTthread,NULL,collectTShirts,&performer[id]);
//Collect T-Shirts for Main Performer
        pthread_join(MainTthread,NULL);//join
    }

    leave:
    //leave
    return NULL;
}
```

joinPerformer

The code is commented extensively for clarity This version only exists only for the singer type. This is the racer who tries to join an already existing performer. Here the performer does a timed wait on the soloperfomanceswheresingerscanjoin semaphore. If the wait returns -1 it means the wait timed out. `if(performer[id].status!=LEFT&&performer[id].status==STAGE_NOT_ASSIGNED)` then performer has not left yet and has not been assigned a stage, that means he/ she is impatient and should leave now, so they leave with a print and we keep this performers status as `LEFT`. `if(performer[id].status==LEFT||performer[id].status==STAGE_ASSIGNED)` this means this performer left in another thread or he has been assigned a stage in another thread just leave this thread, this version lost the race. We also increase semaphore because this version is not going to perform. If we proceed further from this point, that means this version won the race, so there must be a stage where there singer can join since we acquired the semaphore. So we go through the array once and choose stage and update everything appropriately and leave.

```
void *joinPerformer(void *performerargs){
    int id = ((struct Performer *)performerargs)->id;//get the id to be
used on arrays
    //clock for timed wait
    struct timespec ts;
    clock_gettime(CLOCK_REALTIME, &ts);
    ts.tv_sec += t;
    int returnValue = sem_timedwait(&soloPerformancesWhereSingerCanJoin,
&ts);
    pthread_mutex_lock(&performer[id].lock);// lock this performer,
critical section
    if(returnValue==-1){ // if -1 it means timed out.

if(performer[id].status!=LEFT&&performer[id].status==STAGE_NOT_ASSIGNED){
// if the performer has not left yet and has not been assigned a stage,
that means he/ she is impatient and should leave now
                performer[id].status=LEFT;//The performer will leave

printf(LEAVING_AFTER_TIME_OUT,performer[id].name,performer[id].instr);

        }
        pthread_mutex_unlock(&performer[id].lock);//unlock
        return NULL;

    }
    if(performer[id].status==LEFT||performer[id].status==STAGE_ASSIGNED){
//if this performer left in another thread or he has been assigned a stage
in another thread just leave this thread, he lost the race
        sem_post(&soloPerformancesWhereSingerCanJoin);//increase the number
of joinable stages back, since we decreased it before and we are leaving it
        pthread_mutex_unlock(&performer[id].lock);//unlock the performer to
avoid deadlocks
        return NULL;
    }
    for(int i=0; i<num_of_stages;i++){
        pthread_mutex_lock(&stage[i].lock); //lock this stage to avoid
```

```
concurrency issues
        if(stage[i].state==SINGER_CAN_JOIN_PERFORMANCE_GOING_ON){ // A
singer can join this stage
            //updating this performer
            performer[id].stageid=i;
            performer[id].status=STAGE_ASSIGNED;
            //update this stage
            stage[i].state=SINGER_CANNOT_JOIN_PERFORMANCE_GOING_ON;
            //update the performer at stage
            performer[stage[i].performerid].Singerid=id;


printf(SINGER_JOINING,performer[id].name,performer[stage[i].performerid].na
me);
            pthread_mutex_unlock(&stage[i].lock); //unlock the stage to
avoid deadlocks
            break;

        }
        pthread_mutex_unlock(&stage[i].lock);// unlock the stage if the if
statement didn't execute
    }
    pthread_mutex_unlock(&performer[id].lock);//unlock the performer
    //leave
    return NULL;
}
```

joinFreeAcousticStage

The code is commented extensively for clarity Same logic and code as `void *joinFreeElectricStage(void *performerargs)` except that in `void *joinFreeAcousticStage(void *performerargs)` we wait on accoustic semaphore and we check for acoustic stage.

T-Shirt Collection

This is where the T-Shirt Collection happens. We wait on the coordinator semaphore, print , sleep for 2s to simulate collection time, and the increase it back. This thread is created by the other threads for T-Shirt Collection.

```
void *collectTShirts(void *performerargs){
    int id = ((struct Performer *)performerargs)->id;//get the id to be
used on arrays
    sem_wait(&coordinator); // wait for a coordinator
    printf(T_SHIRT_COLLECTION,performer[id].name);
    sleep(2); //simulate collection time
    sem_post(&coordinator); // increase coordinator back
    return NULL;
}
```

States of Entities

```
// States of Stage
#define FREE 0
#define SINGER_CAN_JOIN_PERFORMANCE_GOING_ON 1
#define SINGER_CANNOT_JOIN_PERFORMANCE_GOING_ON 2


//State of Performers
#define STAGE_NOT_ASSIGNED 0
#define STAGE_ASSIGNED 1
#define LEFT 2

//Types of Stages
#define ACOUSTIC 0
#define ELECTRIC 1
```

Global Variables and Arrays

```
//Inputs
int k,a,e,c,t1,t2,t;
char instr;
char name[1000];
int ar;
int num_of_stages;
int num_of_performers_who_left=0;

//semaphores
sem_t acoustic; //this is the semaphore representing empty acoustic stages
sem_t electric; //this is the semaphore representing empty electric stages
sem_t coordinator; //this is the semaphore representing coordinators
sem_t soloPerformancesWhereSingerCanJoin;//this is the semaphore
representing solo performaces, where a singer can join
pthread_mutex_t global_lock;//lock for global variables

struct Stage
{
    int id; //stores id of stage
    int state; //stores state of Stage
    int type; //type of stage
    int performerid;
    pthread_mutex_t lock; //locks this stage for concurrency issues

};
struct Stage *stage; //array of stages


struct Performer
{
    int id;
```

```
    char name[1000];
    char instr; //the instrument
    int arrival_time;
    int status;
    int isSinger; //is this performer a singer or not
    int Singerid; //the id of singer who joined the performance
    int stageid;
    pthread_mutex_t lock; //locks this performer for concurrency
};
struct Performer *performer; //array of performers
```

Helper Functions

```
//gets a random number including lower and upper bounds
int getRandom(int lb, int ub)
//resets the stage
void resetStage(int id)
```

Main

It initialises, takes inputs, launches threads and joins them back. Then it moves on to cleaning up and exits.