Automating Biryani Serving

Approach:

There are m chefs, n tables and k children. This program aims to automate the process of giving biryani to the students such that everyone gets a portion.

In the main function I initialise the chefs, tables and students, in new threads, using their respective init functions.

In chef_init, using a random number generator, I initialise a chef with r number of vessels produced, each serving p people, and in time w. Once produced we go through all the tables to check if it's available to deposit a vessel by calling biryani_ready(). We do this till all our vessels are exhausted.

In tables_init, we initialise a table with an Id and the number of slots in which it serves. After creating the table element, it goes into the ready_to_serve function to show to the chefs that it can be used to serve biryani.

In student_init, a student is initialised with an index. Then it goes into wait_for_slot function, in order to get a slot. Once it gets an open slot, it goes into the in_slot function, where it waits until all the slots of the table are filled and then eats biryani. We show the eating of biryani by decrementing the number of students, number of slots and number of available portions.

The main reason that the code runs without deadlock is the use of mutex. The mutex locks on a memory stops other items from accessing that piece of memory thus removing any incoherency in values. Therefore whenever an thread wishes to access some memory it checks if it has been locked. If not, it locks it and accesses the memory.And unlocks it after use for other threads to use that memory.

Implementation:

```
struct robot_chef{
    int index;
    int preptime;
    int num_vessels;
    int capacity;
} * chefs;
```

```
struct serving_table{
    int index;
    int serving_container;
    int slots;
} * tables;
struct students{
    int index;
    int status;
} * students;
```

Structs for storing data of each instance of thread created.

```
pthread_mutex_t *mutex_for_chefs,*mutex_for_tables;
```

Create for mutex locks for each thread of chefs and tables.

Call biryani_ready by chef number = index

```
void biryani_ready(int index){
    while(chefs[index].num_vessels){
        for(int j=0; j<n; j++){
            if(chefs[index].num_vessels <= 0){
                break;
            }
            if(pthread_mutex_trylock(&mutex_for_tables[j])){
                continue;
```

```
            }
```

Trylock to check if table is engaged by another chef or not.

If yes, next table,

Else, locks and proceeds.

```
            if(tables[j].serving_container==0){
                printf("Chef %d served table %d with capacity
%d\n",chefs[index].index,tables[j].index,chefs[index].capacity);
                chefs[index].num_vessels-=1;
                tables[j].serving_container=chefs[index].capacity;
            }
```

If table is empty,serves biryani.

```
            pthread_mutex_unlock(&mutex_for_tables[j]);
        }
    }
}

void student_in_slot(int j){
    tables[j].serving_container-=1;
    tables[j].slots-=1;
    pthread_mutex_unlock(&mutex_for_tables[j]);
    studentsremain--;
    while(tables[j].slots!=0 && studentsremain!=0){
    }
```

Wait for all slots to be filled.

}

Student_in_slot assigns a student to a slot on the table j and waits for all slots to be filled or all students to be seated before allowing student to eat.

```
void ready_to_serve_table(int index){
    int flag = 0;
    while(flag == 0){
        if(pthread_mutex_trylock(&mutex_for_tables[index])){
            continue;
        }
        if(studentsremain==0)    flag = 1;
        if(tables[index].slots==0)      flag = 1;
        pthread_mutex_unlock(&mutex_for_tables[index]);
    }
}
```

Makes table number = index ready to be provided biryani by the chef.

```
void * chef_init(void * arg){
    int *index_pointer = (int*) arg;
    int index = *index_pointer;
    int flag3 = 10;
    while(flag3 == 10){
        chefs[index].num_vessels=1+(rand()%10);
```

```
        chefs[index].capacity=25+(rand()%26);

        chefs[index].preptime=2+(rand()%4);

        sleep(chefs[index].preptime);

        printf("Chef %d has prepared %d vessels which feed %d people in %d
seconds\n",chefs[index].index,chefs[index].num_vessels,chefs[index].capacity,chefs[ind
ex].preptime);



        biryani_ready(index);

        if(studentsremain==0)    flag3 = -1;

    }
```

Initialize chef with random values and also proceed to serving biryani till required.

```
    return NULL;

}

void * table_init(void * arg){

    int *index_pointer = (int*) arg;

    int index = *index_pointer;


    while(1){

        if(studentsremain==0){

            break;

        }

        if(tables[index].serving_container>0){
```

```
            tables[index].slots=1+(rand()%10);

            if(tables[index].serving_container < tables[index].slots)   tables[index].slots =
tables[index].serving_container;
```

Min of randomized value and remaining students to be served for available slots.

```
            printf("Table %d serving with %d slots and has capacity
%d\n",tables[index].index,tables[index].slots,tables[index].serving_container);

            fflush(stdout);

            ready_to_serve_table(index);

            printf("Table %d has finished its container\n",index);

        }

    }


    return NULL;

}


void * wait_for_slot(void * arg){

    int *index_pointer = (int*) arg;

    int index = *index_pointer;

    int flag2 = 10;

    printf("Student %d is waiting to be allocated a slot on the serving table\n",index);

    while(flag2 == 10){

        for(int j=0; j<n; j++){
```

```
            if(pthread_mutex_trylock(&mutex_for_tables[j])){
                continue;
            }
            if(tables[j].slots>0){
                printf("Student %d is going to eat at
%d\n",students[index].index,tables[j].index);
                fflush(stdout);
                student_in_slot(j);
                flag2 = -1;
                printf("Student %d has finished eating at
%d\n",students[index].index,tables[j].index);
                fflush(stdout);
                break;
            }
            pthread_mutex_unlock(&mutex_for_tables[j]);
        }
    }

    return NULL;
}
```

Driver function for a student, i.e. lifetime of a student. Gets assigned a table, eats and then leaves.

```
int main(void){
    srand(time(NULL));
```

Seed the current time as the seed for rand()

```
printf("Enter the number of Robot chefs, Serving Tables and Students: \n");
scanf("%d %d %d",&m,&n,&k);
pthread_t timid[m],tabletimid[n],studenttimid[k];
```

To store the ids for recognizing each thread of each type.

```
mutex_for_tables=(pthread_mutex_t *)malloc((n)*sizeof(pthread_mutex_t));
studentsremain=k;
chefs=(struct robot_chef*)malloc(sizeof(struct robot_chef)*(m));
mutex_for_chefs=(pthread_mutex_t *)malloc((m)*sizeof(pthread_mutex_t));
```

```
for(int i=0; i<m; i++){
    chefs[i].index=i;
    pthread_create(&timid[i],NULL,chef_init,(void *)&chefs[i].index);
    usleep(100);
}
sleep(1);
tables=(struct serving_table*)malloc(sizeof(struct serving_table)*(n));
for(int i=0; i<n; i++){
    tables[i].index=i;
    pthread_create(&tabletimid[i],NULL,table_init,(void *)&tables[i].index);
```

```
        usleep(100);
    }
    sleep(1);
    students=(struct students*)malloc(sizeof(struct students)*(k));
    for(int i=0; i<k; i++){
        students[i].index=i;
        printf("Student %d has arrived\n",i);
        pthread_create(&studenttimid[i],NULL,wait_for_slot,(void *)&students[i].index);
        usleep(100);
    }
Create threads for chefs,tables and students.
    for(int i=0; i<k; i++){
        pthread_join(studenttimid[i],NULL);
    }
Wait for all students to finish eating biryani.
    printf("Simulation Over\n");
    return 0;
}
```