Report For Question 2 (Biryani) -Risubh Jain(2018101104)

Structs Used

```c
struct robot
{
   int time_to_prepare;
   int vessels_prepared;
   int capacity_of_vessel;
   pthread_t thread_id;
};

struct table
{
   int current_capacity;
   int current_slots;
   pthread_t thread_id;
};

struct student
{
   pthread_t thread_id;
};

struct id_number
{
   int id;
};
```

Shared Memory Functions

```c
struct robot *shareMemRobot(size_t size)
{
   key_t mem_key = IPC_PRIVATE;
   int shm_id = shmget(mem_key, size, IPC_CREAT | 0666);
   return (struct robot *)shmat(shm_id, NULL, 0);
}

struct table *shareMemTable(size_t size)
{
   key_t mem_key = IPC_PRIVATE;
   int shm_id = shmget(mem_key, size, IPC_CREAT | 0666);
   return (struct table *)shmat(shm_id, NULL, 0);
}

struct student *shareMemStudent(size_t size)
{
   key_t mem_key = IPC_PRIVATE;
   int shm_id = shmget(mem_key, size, IPC_CREAT | 0666);
   return (struct student *)shmat(shm_id, NULL, 0);
}
```

- Functions to allocate shared memory and return pointer to memory segment
- Different function for robot,table,student

Mutex Used

```
pthread_mutex_t studentmutex = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t Robot_mutex[10005];
pthread_mutex_t Table_mutex[10005];
```

- studentmutex - mutex on the count of the number of students left and food capacity of table
- Robot_mutex - separate mutex for each robot to prevent multiple table from accessing the same robot
- Table_mutex - separate mutex for each table to prevent multiple students from accessing the same table

Order of polling

- Table polls robots for food
- Student polls tables for slot

For Robot

```
void *Robot_prepare_food(void *thr)
{
   struct id_number *temp = (struct id_number *)thr;
   int id_number = temp->id;
   while (1)
   {
       Robots[id_number].time_to_prepare = rand() % (4) + 2;
       sleep(Robots[id_number].time_to_prepare);
       biryani_ready(id_number);
   }
   return NULL;
}
```

- Initial function when new robot thread starts
- Sleep for a random time and then call biryani_ready function

```c
void biryani_ready(int id_number)
{
   pthread_mutex_lock(&Robot_mutex[id_number]);
   Robots[id_number].vessels_prepared = rand() % 2 + 1;
   Robots[id_number].capacity_of_vessel = rand() % 5 + 1;
   printf(ROBOT_COLOR "Robot %d prepared %d vessels,each with capacity
%d\n" RESET, id_number, Robots[id_number].vessels_prepared,
Robots[id_number].capacity_of_vessel);
   fflush(stdout);
   pthread_mutex_unlock(&Robot_mutex[id_number]);
   while (Robots[id_number].vessels_prepared > 0)
   {
       pthread_mutex_lock(&Robot_mutex[id_number]);
       pthread_mutex_unlock(&Robot_mutex[id_number]);
   }
   printf(ROBOT_COLOR "Biryani finished for Robot %d\n" RESET, id_number);
   fflush(stdout);
}
```

- Lock mutex so that table is not able to poll the robot
- Assign value to vessels_prepared and capacity_of_vessel and unlock mutex
- While loop to check the vessels_prepared for the particular robot
- Lock and unlock mutex inside while loop to stop while loop when a table is getting refilled by the robot
- After vessels_prepared reaches 0 go back to while loop in parent function

For Table

```c
void *Table_loop(void *thr)
{
   struct id_number *temp = (struct id_number *)thr;
   int id_number = temp->id;

   printf(TABLE_COLOR "Serving table %d is empty and waiting for refill\n" RESET, id_number);
   for (int i = 1; i <= n_of_robots; i = (i + 1) % (n_of_robots + 1) + (i == n_of_robots))
   {
       pthread_mutex_lock(&Robot_mutex[i]);
       if (Robots[i].vessels_prepared > 0 )
       {
           sleep(1);
           Tables[id_number].current_capacity = Robots[i].capacity_of_vessel;
           printf(TABLE_COLOR "Table %d was refilled by robot %d and has %d servings left\n"
RESET, id_number, i, Robots[i].capacity_of_vessel);
           fflush(stdout);
           Robots[i].vessels_prepared--;
       }
       pthread_mutex_unlock(&Robot_mutex[i]);

       while (Tables[id_number].current_capacity > 0)
       {
           pthread_mutex_lock(&studentmutex);
           if (n_of_students == 0)
           {
               return NULL;
           }
           int number_of_slots = min(Tables[id_number].current_capacity, min(rand() % (10) + 1,
n_of_students));
           n_of_students -= number_of_slots;
           pthread_mutex_unlock(&studentmutex);
           sleep(1);
           Tables[id_number].current_slots = number_of_slots;
           printf(TABLE_COLOR "Serving table %d is ready to serve with %d slots\n" RESET,
id_number, Tables[id_number].current_slots);
           ready_to_serve_table(number_of_slots, id_number);
       }
   }
   return NULL;
}
```

- A table polls every possible robot one by one
- Try to lock the mutex for particular robot
- If lock is available lock it and check vessel_prepared for robot .If vessel are available, load food into table and reduce the number of vessels by 1 and unlock mutex
- If lock is not available go to next robot (cannot enter second while loop as table has no food)
- If table gets food from the robot go to the while loop
- In the while loop lock studentmutex before calculating number of slots for the table so that values of n_of_students and current capacity of table is not changed improperly
- Calculate number of slots as min(CapacityofTable,rand(1-10),n_of_students);
- Assign slots to table and call ready_to_serve function

```
void ready_to_serve_table(int number_of_slots, int id_number)
{
   while (Tables[id_number].current_slots != 0)
   {
       pthread_mutex_lock(&Table_mutex[id_number]);
       pthread_mutex_unlock(&Table_mutex[id_number]);
   }
   pthread_mutex_lock(&studentmutex);
   printf(TABLE_COLOR "Table %d finished serving %d slots\n" RESET,
id_number, number_of_slots);
   Tables[id_number].current_capacity -= number_of_slots;
   if (Tables[id_number].current_capacity == 0)
   {
       printf(TABLE_COLOR "Serving table %d is empty and waiting for
refill\n" RESET, id_number);
   }
   pthread_mutex_unlock(&studentmutex);
}
```

- Slots have been assigned to table now
- While loop to check if slots become 0 . Mutex lock to stop the loop when some student is polling the table
- When slots become 0 we have to reduce capacity of container on the table
- studentmutex is locked to prevent value of current_capacity from changing inappropiately
- Unlock mutex and return to parent function

For Student

```
void *Student_loop(void *thr)
{
   struct id_number *temp = (struct id_number *)thr;
   int id_number = temp->id;
   sleep(rand() % 10 + 4);
   printf(STUDENT_COLOR "Student %d has arrived\n" RESET, id_number);
   int table_alloted = wait_for_slot(id_number);
   student_in_slot(id_number, table_alloted);
   pthread_mutex_unlock(&Table_mutex[table_alloted]);
   return NULL;
}
```

- Sleep for random time so that all students don't come at once
- After student has arrived student , it calls wait_for_slot which returns value of assigned table
- Student goes to table and calls student_in_slot
- Unlock table mutex which was locked when wait_for_slot was called

```
void student_in_slot(int student_id, int table_id)
{
   Tables[table_id].current_slots--;
   printf(STUDENT_COLOR "Student %d has been assigned a slot at table 
%d\n" RESET, student_id, table_id);
}
```

- Reduce current_slots for chosen table

```c
int wait_for_slot(int id_number)
{
   int i = 1;
   printf(STUDENT_COLOR "Student %d is waiting for slot\n" RESET,
id_number);
   while (1){
       for (; i <= n_of_tables;){
           if (pthread_mutex_trylock(&Table_mutex[i])){
               i = rand() % n_of_tables + 1,sleep(1);
               break;
           }
           if (Tables[i].current_slots == 0){
               i = rand() % n_of_tables + 1,sleep(1);
               pthread_mutex_unlock(&Table_mutex[i]);
               break;
           }
           return i;
       }
   }
}
```

- Enter while loop which exits when a slot is found at any table
- For loop which goes through tables randomly
- In for loop try to acquire lock and if acquired check if current_slot at table are 0
- If slots are 0 , unlock mutex for this table and go to different table else return id_number of this table
- If lock was not acquired i.e another student is polling this table go to a different table

In Main

```
for (int i = 0; i < 10005; i++)
       pthread_mutex_init(&Robot_mutex[i], NULL),
pthread_mutex_init(&Table_mutex[i], NULL);
   srand(time(NULL));
   printf("Enter number of robots / tables / students\n");
   scanf("%d %d %d", &n_of_robots, &n_of_tables, &n_of_students);
   Robots = shareMemRobot((n_of_robots + 1) * sizeof(struct robot));
   Tables = shareMemTable((n_of_tables + 1) * sizeof(struct table));
   Students = shareMemStudent((n_of_students + 1) * sizeof(struct
student));
```

- Take input for number of robots / tables / students;
- Create shareMemory for each robot / table / student
- Initialize Robot_mutex and Table_mutex

```
for (int i = 1; i <= n_of_robots;)
   {
       struct id_number temp;
       temp.id = i;
       pthread_create(&Robots[i].thread_id, NULL, (void
*)Robot_prepare_food, &temp);
       i++;
       usleep(20);
   }

   int temp_n_of_students = n_of_students;
   for (int i = 1; i <= n_of_tables;)
   {
       Tables[i].current_capacity = 0;
       struct id_number temp;
       temp.id = i;
       pthread_create(&Tables[i].thread_id, NULL, (void *)Table_loop,
&temp);
       i++;
       usleep(20);
   }
for (int iiii = 1; iiii <= n_of_students;)
   {
       struct id_number temp;
       temp.id = iiii;
       pthread_create(&Students[iiii].thread_id, NULL, (void
*)Student_loop, &temp);
       iiii++;
       usleep(30000);
   }
```

- Initialize threads for robots/tables/students
- Usleep to prevent threads from getting random values

```c
for (int i = 1; i <= temp_n_of_students; i++)
   {
       pthread_join(Students[i].thread_id, NULL);
       printf("Student %d finished eating\n", i);
       fflush(stdout);
   }
   printf("Simulation Over\n");
   sleep(15);
```

- Loop to join all student thread
- After all have joined Simulation is over
- Sleep(15) to print any extra statement