Assignment 4
2018101059

Question 1

Takes in array as an input.

Runs three kinds of quicksort,

Normal: two recursive calls one after the other

```
normal_quickSort(arr, low, pi - 1);
normal_quickSort(arr, pi + 1, high);
```

Concurrent: uses fork for both the recursive calls in quicksort. One in parent, other in child.

```
int pid1 = fork();
int pid2;
if (pid1 == 0)
{
    //sort left half array
    concurrent_quickSort(arr, low, pi - 1);
    exit(0);
}
else
{
    pid2 = fork();
    if (pid2 == 0)
    {
        //sort right half array
        concurrent_quickSort(arr, pi + 1, high);
        exit(0);
    }
    else
    {
        //wait for the right and the left half to get sorted
        int status;
        waitpid(pid1, &status, 0);
        waitpid(pid2, &status, 0);
    }
}
```

Threaded: Uses two threads

```
struct arg a1;
a1.l = low;
a1.r = (low + high) / 2;
a1.arr = arr;
pthread_t tid1;
pthread_create(&tid1, NULL, threaded_quickSort, &a1);

struct arg a2;
a2.l = (low + high) / 2 + 1;
a2.r = high;
a2.arr = arr;
pthread_t tid2;
pthread_create(&tid2, NULL, threaded_quickSort, &a2);

pthread_join(tid1, NULL);
pthread_join(tid2, NULL);
```

Partition function same for all.
The code implements quicksort and if array size is 5 or less it switches to insertion sort.
Output is time required for each sort.

Question 2

Global:
First we take in number of chefs, tables, students as input.
Studentcount keeps of track of number of students remaining to eat.

```
int n_tables, m_chefs, k_students;
int studentcount; //0 when all have eaten
```

There are threads for each chef, each table, each student.

```
pthread_t chefthreads[1000];
pthread_t studentthreads[1000];
pthread_t tablethreads[1000];
```

Structs are made for chef/table/student to store required data. An array keeps track of data for each chef/table/student.

```c
typedef struct chefs
{
    int number;   //number in the array
    int prepared; //number of prepared vessels
    pthread_mutex_t lock;
} chefs;

typedef struct tables
{
    int capacity; //number of student it can serve in total
    int serving;  //0 if it has no vessel, 1 if it does
} tables;

typedef struct students
{
    int waiting;      //0 means not arrived, 1 means waiting to eat, 2 means slot assigned, 3 means eate
    int time_arrive;  //when they arrive
    int table_eating; //what table it got a slot at. -1 if no table assigned
    pthread_mutex_t lock;
} students;
```

```c
chefs *cheftrack[1000];
tables *tabletrack[1000];
students *studenttrack[1000];
pthread_mutex_t tablelock[1000];
```

Initialising is done in main.
Threads for the chefs and the students are created in the beginning. Index number of the chef/student is passed to the thread.

For chef thread:

While students are left to be fed, it sets some arbitrary prepare time in range(2,5), prints that it is preparing, then prints when finished. Then it calls the birryani ready function. It waits for biryani ready to return by making all its containers in range(1,10) finished. Keeps repeating until no students left(outer while loop).

```
void *thread_chef(void *arg)
{
    int num = *(int *)arg;
    while (studentcount > 0)
    {
        int w_preparetime = 2 + rand() % 3;

        printf("\x1B[33m"
               "Chef %d is preparing... it will take %d seconds\n"
               "\x1B[0m",
               num, w_preparetime);
        sleep(w_preparetime);

        int r_vessels = 1 + rand() % 10;

        cheftrack[num]->prepared = r_vessels;

        printf("\x1B[33m"
               "Chef %d is done preparing %d vessels,\n"
               "\x1B[0m",
               num, cheftrack[num]->prepared);

        biryani_ready(num);

        while (cheftrack[num]->prepared > 0 && studentcount > 0)
            ;
    }
    // return NULL;
    pthread_exit(NULL);
}
```

For biryani ready:

So thread exits after all students fed, else keeps making biryani and waiting to allocate to some table in biyani ready. Biryani ready polls through the tables looking for an empty table it can give its vessel to. Locks are used to prevent two chefs from supplying the same table. Once it finds a table, it marks it as serving and creates thread for that table.

```c
void biryani_ready(int num)
{
    while (studentcount > 0 && cheftrack[num]->prepared > 0)
    {
        for (int i = 0; i < n_tables; i++)
        {

            if (tabletrack[i]->serving == 0)
            {
                int s = pthread_mutex_trylock(&tablelock[i]);
                if (s == 0)
                {
                    if (tabletrack[i]->serving == 0)
                    {
                        printf("\x1B[32m"
                               "Chef %d gave a vessel to table %d\n"
                               "\x1B[0m",
                               num, i);
                        tabletrack[i]->serving = 1;
                        int *t = malloc(sizeof(int));
                        *t = i;
                        pthread_create(&tablethreads[i], NULL, thread_table, t);
                        cheftrack[num]->prepared--;
                    }
                    pthread_mutex_unlock(&tablelock[i]);
                }
            }
        }
    }
    printf("\x1B[33m"
           "Chef %d exited from biryani ready\n"
           "\x1B[0m",
           num);
    return;
}
```

For table thread:

Created only when it was given a vessel. That vessel has random capacity, which is calculated on the spot here, since all vessels created by chef have random capacity. Then is keeps calling ready to serve until it serves all the portions of biryani it can or all students are served. Then marks itself as not serving anymore so that biryani ready of some chef can find it and give it another vessel, at which time new thread will be created.

```c
void *thread_table(void *arg)
{
    int num = *(int *)arg;
    if (studentcount == 0)
        return NULL;

    pthread_mutex_lock(&tablelock[num]);
    //now table is given a container by robot
    tabletrack[num]->capacity = 25 + rand() % 26;
    pthread_mutex_unlock(&tablelock[num]);

    printf("\x1B[31m"
           "Table %d was just given vessel of capacity %d\n"
           "\x1B[0m",
           num, tabletrack[num]->capacity);

    while (studentcount > 0 && tabletrack[num]->capacity > 0)
    {
        ready_to_serve_table(num);
        printf("\x1B[31m"
               "Table %d just exited from ready to serve it now has capacity %d\n"
               "\x1B[0m",
               num, tabletrack[num]->capacity);
    }

    tabletrack[num]->serving = 0;
    return NULL;
}
```

For ready_serve_table:

It makes random number of slots in range(1,min(10,capacity remaining) available on the table. This function keeps looking for students it can serve. It returns if 1.slots are finished OR 2.no more students waiting. 1 is taken care of by the slots variable. 2 is by flag variable. If it goes through a single iteration and finds that no students are marked as waiting then flag becomes equal to total students and it breaks out of the loop. Here again we use a lock to prevent two tables from serving the same student.

```c
void ready_to_serve_table(int num)
{
    int slots = 1 + rand() % 10;
    if (slots > tabletrack[num]->capacity)
        slots = tabletrack[num]->capacity;
    tabletrack[num]->capacity -= slots;
    printf("\x1B[35m"
           "Table %d made %d slots available\n"
           "\x1B[0m",
           num, slots);

    int flag = 0; //if flag is k_students that means no student is waiting
    while (slots > 0 && flag != k_students)
    {
        flag = 0;
        for (int i = 0; i < k_students; i++)
        {
            if (slots <= 0)
                break;

            else if (studenttrack[i]->waiting == 1)
            {
                int s = pthread_mutex_trylock(&studenttrack[i]->lock);
                if (s == 0)
                {
                    if (studenttrack[i]->waiting == 1)
                    {
                        studenttrack[i]->waiting = 2;
                        studentcount--;
                        pthread_mutex_unlock(&studenttrack[i]->lock);
                        studenttrack[i]->table_eating = num;
                        slots--;
                    }
                }
            }

            else
                flag++;
        }
    }
```

Remaining part of the function is marking all the students as eating since it serves all of them only after it finishes trying to allocate all its slots. Note the slots not allocated will waste the biryani they don’t serve.

```
for (int i = 0; i < k_students; i++)
{
    if (studenttrack[i]->table_eating == num && studenttrack[i]->waiting == 2)
    {
        printf("\x1B[36m"
               "Student %d eating at table %d\n"
               "\x1B[0m",
               i, num);
        studenttrack[i]->waiting = 3;
    }
}
if (slots == 0)
    printf("\x1B[35m"
           "Table %d exited because its slots all used up\n"
           "\x1B[0m",
           num);
else
    printf("\x1B[35m"
           "Table %d exited because no more students are waiting\n"
           "\x1B[0m",
           num);
return;
```

For student thread:

It sleeps until the student arrives (Arrival time is randomised in beginning of main). It then marks the student as waiting for slot and calls wait_for_slot. Once it is done waiting it goes to student_in_slot so that it can take 2 secs to eat. Then it has finished eating so it returns.

```c
void *thread_student(void *arg)
{
    int num = *(int *)arg;
    sleep(studenttrack[num]->time_arrive);

    pthread_mutex_lock(&studenttrack[num]->lock);
    studenttrack[num]->waiting = 1;
    pthread_mutex_unlock(&studenttrack[num]->lock);

    printf("\x1B[34m"
           "Student %d arrived at mess and waiting for slot\n"
           "\x1B[0m",
           num);
    wait_for_slot(num);
    student_in_slot(num);
    // sleep(2);
    printf("\x1B[34m"
           "Student %d finished eating\n"
           "\x1B[0m",
           num);
    return NULL;
}
```

For wait_for_slot:

It waits for the table to mark it as waiting done slot assigned and served.

```c
void wait_for_slot(int num)
{
    while (studenttrack[num]->waiting != 3)
        ;
}
```

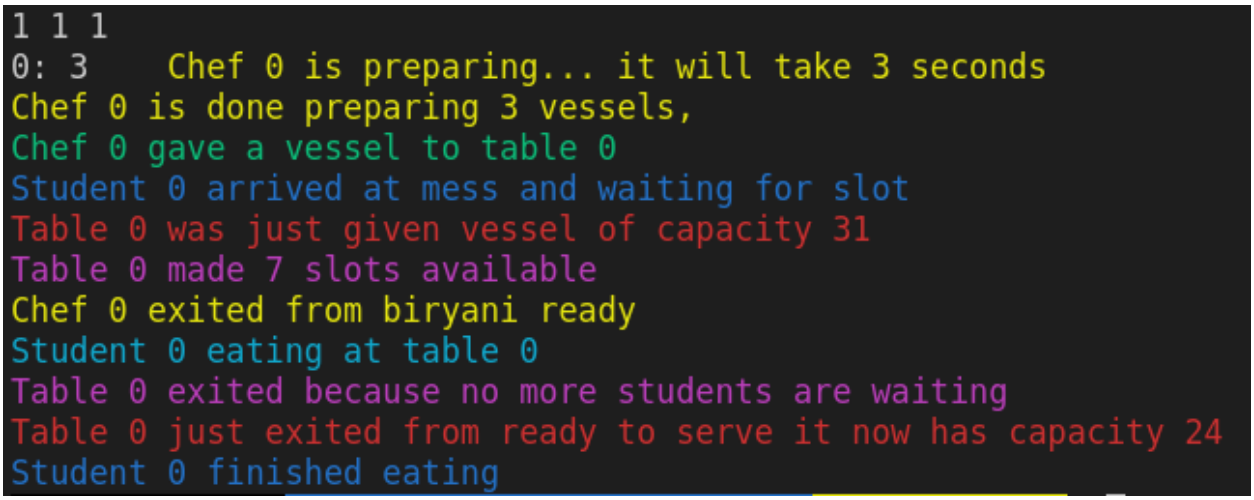
For student_in_slot:

It eats for 2 secs

```c
void student_in_slot(int num)
{
    sleep(2);
}
```

Finally when studentcount becomes 0 then all have been served so all threads return. Then all threads are joined in main, locks are destroyed and program is finished :D.

SAMPLE OUTPUT:



Question 3:

Global variables:

Input is n cabs,m riders and k servers. There are threads for the riders and servers. Cab is implemented as an array of structs which store its status: 0 means waitState, 1 means onRidePremier, 2 means onRidePoolOne, 3 means onRidePoolFull.

```
typedef struct rider
{
    int time_ride; //how long is the ride
    int time_arrive;  //when booked
    int time_wait;  //how much time will it wait to book
    int type;  //0 means premier,1 means pool
    int number;
    int pay; //1 if payment pending, else 0
} rider;

typedef struct cab
{
    int status; //0 means free, 1 means premier, 2 means poolone, 3 means poolfu

} cab;

int n_cabs, m_riders, k_servers, number;

pthread_t riderthreads[1000];
pthread_t riderthreadsnew[1000];
pthread_t paymentthreads[1000];

int riderflag[1000];
rider *ridertrack[1000];
cab cabtrack[1000];
```

There are two semaphores, one to keep track of number of cabs in state 0 and other used in payment servers. Besides there are 3 locks, lock is used whenever polling over the cab array or changing values to prevent two riders from doing that. lock2 used in a special case of poolcab, lock3 used in payment server.

```
sem_t freecabs, pay;
pthread_mutex_t lock, lock2, lock3;
```

In main:

All the initialising is done. Values in rider struct initialised randomly and printed. Payment and rider threads created. There are two kinds of rider threads: one for pool and one for premier. It sees the type then calls appropriate function for thread.

```
if (ridertrack[i]->type == 0)
    pthread_create(&riderthreads[i], NULL, thread_premier, (void *)(ridertrack[i]));
else
    pthread_create(&riderthreads[i], NULL, thread_pool, (void *)(ridertrack[i]));
```

In premier thread:

Sleeps until rider arrives. Then does a timed wait on the semaphore. If timeout prints error. Else locks to find which cab got available, sleeps for ride duration then marks as ride over. Posts back to the semaphore and marks the cab it used as waitState. Then marks itself as payment pending then exits.

```
struct timespec ts;
if (clock_gettime(CLOCK_REALTIME, &ts) == -1)
    return NULL; //error handliing!!!!!
ts.tv_sec += input->time_wait;

int s;
while ((s = sem_timedwait(&freecabs, &ts)) == -1 && errno == EINTR)
    continue;

if (s == -1)
{
    if (errno == ETIMEDOUT)
    {
        printf("\033[1;31m"
               "TIMEOUT: Unable to find cab for rider %d\n"
               "\033[0m",
               input->number);
        number--;
    }
    else
        perror("sem_timedwait");
}
```

```c
pthread_mutex_lock(&lock);
int i;
for (i = 0; i < n_cabs; i++)
{
    if (cabtrack[i].status == 0)
    {
        cabtrack[i].status = 1;
        break;
    }
}
pthread_mutex_unlock(&lock);

printf("\x1B[33m"
       "Rider %d riding in cab %d\n"
       "\x1B[0m",
       input->number, i);

sleep(input->time_ride); //riding

printf("\x1b[35m"
       "Rider %d exited from cab %d\n"
       "\x1B[0m",
       input->number, i);

ridertrack[input->number]->pay = 1;
sem_post(&pay);

pthread_mutex_lock(&lock);
cabtrack[i].status = 0;
pthread_mutex_unlock(&lock);

sem_post(&freecabs);
```

In pool thread:

Sleeps until rider arrives. Now checks if there is an existing cab in state onRidePoolOne. If yes immediately seats it there.

```
pthread_mutex_lock(&lock);
for (i = 0; i < n_cabs; i++)
{
    if (cabtrack[i].status == 2) //can join an existing cab
    {
        cabtrack[i].status = 3;
        break;
    }
}
```

If no, it needs to check for either cab freed, or some cab with state onRidePoolOne, in wait time of rider. At this point the main thread checks for onRidePoolOne cab by polling (Here a lock2 exists since we don't want more than one cab polling). It also creates another thread to do a timed wait on the semaphore. A riderflag is set 0 initially which means looking for both conditions. If -2 means timeout. Then it will return. Other thread prints the timeout message. If -1 means it found onRidePoolOne. If 1 then it found a free cab within time. Then status of cabs is appropriately updated and code continues.

```c
pthread_create(&riderthreadsnew[i], NULL, thread_checker, (void *)(input));

pthread_mutex_lock(&lock2);
while (riderflag[input->number] == 0)
{
    for (i = 0; i < n_cabs; i++)
    {
        if (cabtrack[i].status == 2)
        {
            pthread_mutex_lock(&lock);
            riderflag[input->number] = -1; //can join an existing cab
            cabtrack[i].status = 3;
            break;
        }
    }
}

pthread_mutex_unlock(&lock2);

if (riderflag[input->number] != -1)
    pthread_mutex_lock(&lock);

if (riderflag[input->number] == -2) //means timeout
{
    pthread_mutex_unlock(&lock);
    pthread_join(riderthreadsnew[input->number], NULL);
    return NULL;
}
else if (riderflag[input->number] == 1) //means found a free cab
{
    for (i = 0; i < n_cabs; i++)
    {
        if (cabtrack[i].status == 0)
        {
            cabtrack[i].status = 2; //can join an existing cab
            break;
        }
    }
}
```

It prints starts riding, sleeps for ride time, then marks itself as ride finished. Also marks itself as payment pending. Cab status is updated while it is exiting to appropriate value then exits.

```
ridertrack[input->number]->pay = 1;
sem_post(&pay);
pthread_mutex_lock(&lock);
if (cabtrack[i].status == 3)
{
    cabtrack[i].status = 2;
}
else if (cabtrack[i].status == 2)
{
    sem_post(&freecabs);
    cabtrack[i].status = 0;
}
```

In checker thread:

This does a timed wait on the semaphore for pool riders. Created in case described above. Sem_timedwait return value s is 0 if semaphore decremented successfully, else 1 if timeout. However if riderflag is -1 it was already allocated a seat of type onRidePoolOne. The posts back to the semaphore func returned 0(since unecessarily decremented sem value). If riderflag is not -1, it checks s value. If s=-1 then timeout, it will print the error and mark riderflag as -2. Else it marks riderflag as 1 ie found a free cab to sit in. Then returns.

```c
struct timespec ts;
if (clock_gettime(CLOCK_REALTIME, &ts) == -1)
    return NULL; //error handliing!!!!!
ts.tv_sec += input->time_wait;

int s;

while ((s = sem_timedwait(&freecabs, &ts)) == -1 && errno == EINTR)
    continue;

if (riderflag[input->number] == -1) //means found an empty seat
{
    if (s != -1)
        sem_post(&freecabs);
    return NULL;
}

if (s == -1)
{
    riderflag[input->number] = -2; //means timeout
    if (errno == ETIMEDOUT)
    {
        printf("\033[1;31m"
               "TIMEOUT: Unable to find cab for rider %d\n"
               "\033[0m",
               input->number);
        number--;
    }
    else
        perror("sem_timedwait");
}
else
{
    riderflag[input->number] = 1;
}
return NULL;
```

For servers:

Using a global semaphore pay.Then all servers are waiting on that semaphore. As soon as a rider finishes their ride, it will do a sem post and that will cause one of the servers to be notified (since semaphore got incremented). That uses lock3 and checks all the cabs once to see which needs to pay. Then it sleeps for 2 secs which is time required to pay, then

completes payment.Global number variable is decremented either if a rider finishes paying or times out.

```
while (number > 0)
{
    sem_wait(&pay);
    sleep(2);

    pthread_mutex_lock(&lock3);
    for (int i = 0; i < m_riders; i++)
    {
        if (ridertrack[i]->pay == 1)
        {
            printf("\x1B[32m"
                   "Payment complete for rider %d from server %d\n"
                   "\x1B[0m",
                   i, num);
            number--;
            ridertrack[i]->pay = 0;
            break;
        }
    }
    pthread_mutex_unlock(&lock3);
}
```

Finally when number becomes 0 then all have been served so all threads return. A sem post is done k times to free any servers waiting to perform their payment. Then all threads are joined in main, locks and semaphores are destroyed and program is finished :D.

SAMPLE OUTPUT:

```
1 2 1
Number=0 Type=1 Arrive=3 Ride=5 Wait=4
Number=1 Type=0 Arrive=1 Ride=5 Wait=5
Rider 1 riding in cab 0
Rider 1 exited from cab 0
Rider 0 riding in cab 0
Payment complete for rider 1 from server 0
Rider 0 exited from cab 0
Payment complete for rider 0 from server 0
```