# Ober Services

Approach:
There are M cabs, N riders and K servers. We must create a pipeline such that there are no deadlocks or errors and every rider must get a ride or he leaves after a max waiting time.
In the main function I initialise the cabs, riders and servers in separate threads.
In cabs_init and server_init, using a random value generator I give each cab/server its attributes.
In riders_init I do the same thing as in cabs_init and servers_init and after that I pass each new rider through the book_ride function.
In the book_ride function the rider searches every cab to see if it's available. If a cab is available, we check if the cab accepts the request in the accept_ride function.
In the accept_ride function the available seats decrease as it is taken by the new rider.
Then the rider rides for the ride time. At the end the rider moves to the end_ride function where the number of seats will increase by 1 as the rider exits the cab.
After the ride the rider has to pay through some payment server. To do this it enters the make_payment function. In the make_payment function the rider searches through all the servers to find a slot. Once it finds a slot it waits for the amount of time required to process the request and then the transaction is successful and the rider exits.

Implementation:

```
struct cab{
        int id;
        int type;
        int num_slots;
}caabs[1000];

struct rider{
        int id;
        int pref;
        int max_wait_time;
        int ride_time;
}riders[1000];

struct server{
        int id;
        int empty;
}servers[1000];
```

Cabs, riders and servers maintain a list of their respective objects and each such entity is initialised with its attributes.

```
void accept_ride(int rider_id, int cab_id){
        caabs[cab_id].num_slots--;
```

```
        printf("Rider %d is travelling in cab %d. %d seats left\n",rider_id, cab_id,
caabs[cab_id].num_slots);
        return;
}

void end_ride(int rider_id, int cab_id){
        int check = 0;
        while(!check){
        if(!pthread_mutex_trylock(&mutex_cabs[cab_id])){
        caabs[cab_id].num_slots++;
        printf("Rider %d has reached his location in cab %d\n", rider_id, cab_id);
        return;
        }
        }
        return;
}

int book_ride(int id){
        int check = 0;
        long double start = time(NULL);
        while(!check){
        int j = M;
        while(j--){
        long double curr = time(NULL);
        if((int)(curr-start)>riders[id].max_wait_time){
                printf("Rider %d has left at time %Lfs: Time Error\n", riders[id].id, curr-start);
                return 0;
        }
        if(!pthread_mutex_trylock(&mutex_cabs[M-j]) && riders[id].pref == caabs[M-j].type &&
caabs[M-j].num_slots){
                accept_ride(id, M-j);
                pthread_mutex_unlock(&mutex_cabs[M-j]);
                sleep(riders[id].ride_time);
                end_ride(id, M-j);
                pthread_mutex_unlock(&mutex_cabs[M-j]);
                return 1;
        }
        pthread_mutex_unlock(&mutex_cabs[M-j]);

        }
        }
        return 0;
}
```

This snippet is about finding the cab. It starts with the book_ride function where the rider moves through each available cab until it finds one which has an empty slot, is of the preferred type and its not being used at the moment. If it doesn't find a cab in the max_waiting time it exits the function and returns 0. On finding such a cab, the rider enters the accept_cab function where the cab loses an available seat due to the new rider. The rider then rides for the set amount of time at the end of which it reaches its location. The rider enters the end_ride function which simply adds an extra slot to the available seats of the cab. When it exits book_ride after finishing the ride the function returns 1.

```
void make_payment(int id){
        int check = 0;
        while(!check){
        int j = K;
        while(j--){
        if(!pthread_mutex_trylock(&mutex_servers[K-j]) && servers[K-j].empty){
                printf("Rider %d using server %d to make payment\n", id, servers[K-j].id);
                servers[K-j].empty = 0;
                pthread_mutex_unlock(&mutex_servers[K-j]);
                sleep(2);
                pthread_mutex_lock(&mutex_servers[K-j]);
                printf("Rider %d has paid through server %d\n",id, servers[K-j].id);
                servers[K-j].empty = 1;
                pthread_mutex_unlock(&mutex_servers[K-j]);
                check = 1;
                return;
        }
        pthread_mutex_unlock(&mutex_servers[K-j]);
        }
        }
        return;
}
```

In this snippet the rider tries to make a payment to the cab through a payment server. The rider goes  through each server and checks if its being used. If not it uses that server to send a request to pay. The request takes 2 seconds to process after which the payment is done and the rider exits from the scene.

```
void *cab_init(void *args){
        int id = *(int *)args;
        caabs[id].type = generator(0,1);
        if(caabs[id].type)
        caabs[id].num_slots = 2;
        else
```

```
        caabs[id].num_slots = 1;
        printf("Cab of id %d of type %d is available\n",id, caabs[id].type);
}

void *server_init(void *args){
        int id = *(int *)args;
        servers[id].empty = 1;
}
```

This is just initialisation of a cab/server.

```
void *rider_init(void *args){
        int id = *(int *)args;
        riders[id].max_wait_time = generator(5,10);
        riders[id].pref = generator(0,1);
        riders[id].ride_time = generator(5,15);
        printf("Rider %d has arrived. It has a ride time of %d, will travel only by car type %d and
will not wait for more than %ds\n", id, riders[id].ride_time, riders[id].pref,
riders[id].max_wait_time);
        if(book_ride(id))
        make_payment(id);
        return NULL;
}
```

This function initializes a rider. It books a ride using the book_ride function. If the rider gets the ride it makes the payment using the make_payment function.

```
        int i = M;
        while(i--){
        caabs[M-i].id = M-i;
        usleep(100);
        pthread_create(&cabtids[M-i], NULL, cab_init, (void *)&caabs[M-i].id);
        }
        i = N;
        while(i--){
        riders[N-i].id = N-i;
        usleep(100);
        pthread_create(&ridertids[N-i], NULL, rider_init, (void *)&riders[N-i].id);
        }
        i = K;
        while(i--){
        servers[K-i].id = K-i;
        usleep(100);
        pthread_create(&servertids[K-i], NULL, server_init, (void *)&servers[K-i].id);
```

```
        }
        i = K;
        while(i--)
        pthread_join(ridertids[K-i], NULL);
        printf("Simulation ended\n");
```

This part in main just creates threads, each initializing an object. It ends after all the rider threads terminate i.e. each rider is taken care of.