Code breakdown for Ober Cab Services

Overview

Create separate threads for all riders and payment servers. Riders continuously check cab availability by looping through the array of cabs. K payment servers check for riders who have completed their rides by waiting on a semaphore.

Run the program

1. Run the command `gcc -pthread -o cabs cabs.c .`
2. Run `./cabs`
3. Input values for number of cabs, riders, and servers to begin the simulation.

Variables

Structs for cabs, riders, and servers

- Cabs:

```
struct Cab
{
    int type;
    int status;
    pthread_mutex_t cab_mutex;
};
```

- Riders

```
struct Rider
{
    int idx;
    int arrivalTime;
    int cabType;
    int maxWaitTime;
    int RideTime;
    int cab_no;
```

```
    int status;
    sem_t payment;
    pthread_t rider_thread_id;
    pthread_mutex_t rider_mutex;
};
```

- Payment Servers

```
struct Server
{
    int idx;
    int status;
    int rider_no;
    pthread_t server_thread_id;
};
```

Arrays

An array of each of the above structures has been created as:

```
Cab Cabs[1000]; Rider riders[1000]; Server servers[1000];
```

Functions

Rider Functions

Rider thread function

- Sleeps for random time to simulate arrival of riders
- Measures time using clock() function to keep track of TIMEOUT condition

```
clock_t time_taken = clock() - t;
double times = ((double)time_taken)/CLOCKS_PER_SEC;
if(times > ((double)rider->maxWaitTime))
{
    time_exceed = 1;
    break;
}
```

- Loops through array of cabs, chooses any free cab if rider wanted premier
- In case of pool, loops through cabs to see if pool is available, if not, goes with a premier

```
for(int i=0; i < no_cabs; i++)
{
    pthread_mutex_lock(&(Cabs[i].cab_mutex));

    if(rider->cabType == 1)
    {
        if(Cabs[i].type == 0)
        {
            ...
        }
    }

    else if(rider->cabType == 2)
    {
        if(Cabs[i].type == 2 && Cabs[i].status == 1)
        {
            ...
        }
    }

    pthread_mutex_unlock(&(Cabs[i].cab_mutex));
}

if(rider->cabType == 2 && flag == 0)
{ .. }
```

- Mutexes are present for each cab so that only one rider tries to access it at a time

```
    pthread_mutex_lock(&(Cabs[i].cab_mutex));
```

- If time has been exceeded, rider exits with a TIMEOUT message
- If cab found, changes cab's status within a mutex, and its number of riders(for pool cabs)
- Sleep for journey time

```
    sleep(rider->RideTime);
```

- Changes state of cab with which it was riding within the cab's mutex

```
pthread_mutex_lock(&(Cabs[no].cab_mutex));
if(Cabs[no].status==1)
{
    Cabs[no].type = 0;
    Cabs[no].status = 0;
}
pthread_mutex_unlock(&(Cabs[no].cab_mutex));
```

- Posts to the riders_ready_to_pay semaphore so that an available payment server can accept the payment

```
rider->status = 1; // completed journey
sem_post(&riders_ready_topay);
```

- Waits for rider payment semaphore

```
sem_wait(&(rider->payment));
```

Server Functions

Server thread function

- Waits on riders_ready_to_pay semaphore in a while(1) loop

```
    while(1)
    {
        sem_wait(&riders_ready_topay);
        server->status = BUSY;
```

- In the critical section, loops through the array of riders to see which one has completed journey
- It reads the rider values by locking the rider mutex so that only one payment server tries to access it at a time

```
for(int i=0; i < no_riders; i++)
```

```
{
    pthread_mutex_lock(&(riders[i].rider_mutex));
    if(riders[i].status == 1)
    {
        ...
        break;
    }

    pthread_mutex_unlock(&(riders[i].rider_mutex));
}
```

- If it finds a rider who has completed a trip (accesses rider info within a mutex), it sleeps for 2 sec to simulate payment time

```
if(server->rider_no)
{
    sleep(2);
    sem_post(&(riders[server->rider_no - 1].payment));
    server->rider_no = 0;
    server->status = FREE;
}
```

- Posts to the rider's payment semaphore so that it can exit from the system

Main

- Takes inputs for number of cabs, riders, and servers
- Initializes the random variables and structs
- Waits for all rider threads to join