Ober Cab Services

Implementation

To begin, each rider and payment server is in it's own thread. There are two variables: `freePoolOnes` (counts the number of free pool cabs which has one passenger in it) and `freeCabs` (counts the number of cabs which are empty).

Whenever a rider arrives, he/she checks if the desired type of cab is available or not. If it is available, the rider takes that cab and begins his/her ride. If no cab is free, it calls `pthread_cond_timedwait`. All of this happens after obtaining a mutex lock.

If the rider doesn't find a cab, he indicates that he is waiting for a cab. The `pthread_cond_timedwait` ensures that the rider doesn't wait for more then `maxWaitTime`.

When some rider is going to end his/her ride, he/she checks the array to see if some rider is waiting for the same type of cab or not. If it finds some rider waiting, it sends a signal to that rider, who gets the lock, and thus the cab and proceeds.

The servers are the most simple. Each thread waits on a semaphore, initialized to 0, in an infinite loop. Whenever a rider wants to pay, he/she calls `sem_post` on that semaphore, so that one server picks it up and finishes the payment for that rider.

Assumptions

Large input is to be avoided, since a large number of threads cause a problem. Also, shared memory is being used in this solution (idk why, but the malloc solution failed ¯_(ツ)_/¯), which is limited.

If `shmget` or `shmat` fail, try running the following command:

```
ipcs -m | grep -E "yog\s+666" | awk '{print $2}' | xargs -n1 ipcrm shm
```

This will clear the type of shared memory created by my application

Code snippets

Rider

Rider checking for available cab:

```
pthread_mutex_lock(&accessCabs);

if((r->cabType == POOL && canGetPool()) ||
        (r->cabType == PREMIER && canGetPremier())) {
    waitingForCab[r->uid] = 0;
    assignCab(r);

    pthread_mutex_unlock(&accessCabs);
} else if((r->cabType == POOL && !canGetPool()) ||
        (r->cabType == PREMIER && !canGetPremier())) {
    waitingForCab[r->uid] = r->cabType;

    int condResult = pthread_cond_timedwait(&condWait[r->uid],
            &accessCabs, getFutureTime(r->waitTime));

    if(condResult) {
        waitingForCab[r->uid] = 0;
        pthread_mutex_unlock(&accessCabs);
        return false;
    }

    waitingForCab[r->uid] = 0;

    assignCab(r);

    pthread_mutex_unlock(&accessCabs);
}
```

Rider handing over the cab to someone else:

```
bool cabWasAssigned = false;
waitingForCab[rider->uid] = 0;

pthread_mutex_lock(&accessCabs);

if(rider->cabType == PREMIER) {
    rider->cab->state = waitState;
    rider->cab->r1 = 0;
    freeCabs++;

    for(int i = 0; i < M; i++) {
        if(waitingForCab[i] == PREMIER || waitingForCab[i] == POOL) {
            pthread_cond_signal(&condWait[i]);
            pthread_mutex_unlock(&accessCabs);
            cabWasAssigned = true;
            printf("Handing over cab %d to %d\n", rider->cab->uid, i);
            break;
        }
    }
} else if(rider->cabType == POOL) {
    int x = checkFree(rider->cab);
```

```
    if(x == 3) {
        rider->cab->state = onRidePoolOne;
        freePoolOnes++;
    } else {
        rider->cab->state = waitState;
        freeCabs++;
        freePoolOnes--;
    }

    if(rider->cab->r1 == rider)
        rider->cab->r1 = 0;
    else if(rider->cab->r2 == rider)
        rider->cab->r2 = 0;

    for(int i = 0; i < M; i++) {
        if(waitingForCab[i] == POOL) {
            pthread_cond_signal(&condWait[i]);
            pthread_mutex_unlock(&accessCabs);
            cabWasAssigned = true;
            printf("Handing over cab %d to %d\n", rider->cab->uid, i);
            break;
        }
    }
}

return cabWasAssigned;
```

Rider paying:

```
sem_post(&paymentServers);
```

Server

Server's code:

```
printf("Server %d initialized\n", ((Server*)s)->uid);

while(1) {
    sem_wait(&paymentServers);

    ...

    sleep(2);
}

printf("Server %d closing down\n", ((Server *)s)->uid);
```