Question 3:

Ober Cab Service

Structures Used

Riders Structure:
The riders struct stores the rider number, what kind of cab he wants(0 for premier, 1 for pool) , the maximum time he is willing to wait, the ride_time of the cab and how long he takes to book the cab from start of program. He also stores payment status in which 0 is ride has not been completed yet, 1 is the payment is processing, 2 is the payment is complete.

Struct cab:
The cab structure stores index and state of the cab as defined in the question.
State 0 = waitState
State 1 = onRidePremier
State 2 = onRidePoolFull
State 3 = onRidePoolOne

Assumption : Each cab can be premier or pool based on availability as the question says the “ride” is premier or pool and I am considering a ride as one car ride and not the definition of the ride being the car itself.

Struct servers:
Servers store index and the status(0 if not in use, 1 if in use)

```c
struct riders{
     int index;
     int cab_type;
     int max_wait_time;
     int ride_time;
     int arrival_time;
     int status;
} * riders;

struct cab{
     int index;
     int state;
} * cab;

struct servers{
     int index;
     int status;
} * servers;
```

```c
while((s = sem_timedwait(&cabs_sem, &ts)) == -1 && errno == EINTR){
     continue;
}
if (s == -1){
     if (errno == ETIMEDOUT){
          printf("Rider %d could not find a cab\n",riders[index].index
     }
     else{
          perror("sem_timedwait");
     }
}
else{
     int riding=-1;
     for(int i=0; i<n; i++){
          if(pthread_mutex_trylock(&mutex_cabs[i])){
               continue;
          }
          if(cab[i].state==0){
               cab[i].state=1;
               riding=i;
               printf("Rider %d is riding in cab %d with state %d\n",r
               pthread_mutex_unlock(&mutex_cabs[i]);
               break;
          }
          pthread_mutex_unlock(&mutex_cabs[i]);
     }
```

Premier Rides

Riders are not allowed into the critical section if there are more number of riders looking than free cabs available. We perform a timed wait to see if the rider times out, or the semaphore unlocks before it. If the semaphore allows the the thread to pass, since a “post” call has been called, there has to be a cab free which I find by iterating through the cabs.
Mutex’s are used for mutual exclusion for each cab.
If the timed wait times out, we print that the rider couldn’t find a cab and return.

Pool Rides

Initially, a pool rider searches if there are cabs available which are already in the State 3 so he can get on immediately. If not, 2 threads are created, one to check on new cabs and one to check for free slots with cabs in state 3. 2 Threads are used so we can accomplish this task simultaneously and by using global variables can recognise which came first and act accordingly. Thread to check for cabs in state 3 is in diagram 2 and the thread to check for new cab is found in diagram 3 .

```
for(j=0;j<n; j++){
     if(pthread_mutex_trylock(&mutex_cabs[j])){
          continue;
     }
     if(cab[j].state==3){
          cab[j].state=2;
          riding=j;
          printf("Rider %d is riding in cab %d with state
          pthread_mutex_unlock(&mutex_cabs[j]);
          break;
     }
     pthread_mutex_unlock(&mutex_cabs[j]);
}
```

```
while((s = sem_timedwait(&cabs_sem, &ts)) == -1 && errno == EINTR){
     continue;
}
int flag=0;
if(riderflag[index]==1){//found onepool
     flag=1;
     if(s!=-1){
          sem_post(&cabs_sem);
          return NULL;
     }
}
if (s != -1){
     riderflag[index]=2;//found cab
}
else{
     riderflag[index]=-1;
     if (errno == ETIMEDOUT){//found nothing
          printf("Rider %d could not find a cab\n",riders[index].index);
     }
     else{
          perror("sem_timedwait");
     }
}
```

```
for(j=0;j<n; j++){
     if(pthread_mutex_trylock(&mutex_cabs[j])){
          continue;
     }
     if(cab[j].state==3){
          cab[j].state=2;
          riding=j;
          printf("Rider %d is riding in cab %d with state
          pthread_mutex_unlock(&mutex_cabs[j]);
          break;
     }
     pthread_mutex_unlock(&mutex_cabs[j]);
}
```

```
servers[index].status=0;
printf("Server %d is now online\n",servers[index].index);
while(1){
     servers[index].status=0;
     sem_wait(&server_sem);
     printf("Server %d has detected a user who is done with
     servers[index].status=1;
     for(int i=0; i<m ;i++){
          pthread_mutex_lock(&mutex_riders[i]);
          if(riders[i].status==1){
               printf("Server %d is accepting the payment of
               sleep(2);
               riders[i].status=2;
               printf("Server %d has accepted the payment of
               pthread_mutex_unlock(&mutex_riders[i]);
               break;
          }
          pthread_mutex_unlock(&mutex_riders[i]);
     }
}
```

Servers

A server initialises with status 0 and then waits on a semaphore which initially allows 0 threads.
Once a rider completes his trip, he changes his status to ready for payment and uses a sem_post which allows a thread through to accept payment. It goes through the list of riders, finds the rider and then takes 2 seconds to process payment and goes back to wait state. This avoid busy waiting.