Ober Cabs

Jivitesh Jain, 2018101092, UG-2, CSE

Overview

The following is a multithreaded simulation which simulates a cab booking and payment system in which the stakeholders - riders, cabs and payment servers - are modelled as threads. The threads use mutex locks and semaphores to stay in sync and augment each other during the course of the simulation.

```
Enter number of riders, cabs and servers: 10 4 3
Running simulation.
RIDER 1 WAITING FOR PREMIER
RIDER 1 RIDING PREMIER IN CAB 0
RIDER 5 WAITING FOR POOL
RIDER 5 RIDING POOL ALONE IN CAB 1
RIDER 4 WAITING FOR PREMIER
...
RIDER 0 RIDING POOL SHARED WITH RIDER 5 IN CAB 1
RIDER 5 REACHED AND WAITING FOR PAYMENT
RIDER 5 PAYING ON SERVER 2
RIDER 9 WAITING FOR POOL
...
RIDER 8 PAYING ON SERVER 0
RIDER 6 DONE
RIDER 8 DONE
Simulation done.
```

Implementation

The simulation uses threads to model the riders, cabs and servers. Data about each thread is stored is dynamically allocated structs, pointers to which are passed around to share data between threads.

Each rider struct, associated with a rider thread, consists of a mutex and a conditional variable. The riders conditionally wait on their conditonal variables until their state is changed to `riding` by a cab. This avoids busy waiting in the rider threads.

The cabs pick up riders at random, and if they are waiting for a ride and match the type of the cab, assign those riders to themselves. This is done after the cab acquires the mutex belonging to that rider, which protects multiple cab threads from picking the same rider. The cab then sends a conditional signal to the rider, informing them of their new ride. The system preferentially assigns `pool` riders to cabs already carrying a `pool` passenger by means of a global count of the number of `pool` cabs and a mutex to protect it.
The rider thread then waits for its ride-time while the cab, depending on the cab-type either waits or looks for another passenger. When the ride ends, the rider sends a signal to its semaphore, which was initialised to 0 and on which the cab was waiting. This lets the cab know that the ride has come to an end.

Because of the lack of different types, payment servers and modelled somewhat differently. A global semaphore, initialised to 0, represents the number of riders waiting to pay. All server threads wait on this semaphore. A rider who has just finished her ride signals to this semaphore, increasing its count by one. This allows exactly one server thread to proceed, which loops through the list of riders to find the one which just signalled, and processes its payment. The use of a semaphore prevents multiple servers from getting triggered by a single rider and prevents busy waiting.

Data Structures

The following structs store most of the data in this simulation. One such struct is assigned to each thread of the corresponding type.

```
typedef struct Rider {
    int id;
    pthread_t tid;

    int type;
    time_t ride_time;
    time_t wait_time;

    int state;

    struct Cab* cab;

    pthread_mutex_t protect;

    pthread_cond_t cv_cab;
```

```
    sem_t riding;
    sem_t paying;

} Rider;

typedef struct Cab {
    int id;
    pthread_t tid;

    int state;

    struct Rider* rider_a;
    struct Rider* rider_b;

    struct Rider** riders;
    int num_riders;

} Cab;

typedef struct Server {
    int id;
    pthread_t tid;

    int state;

    struct Rider* rider;

    struct Rider** riders;
    int num_riders;
} Server;
```

Arrays of these structs are dynamically allocated, because the heap is shared amongst all the threads.

Apart from these, the global variables described earlier are:

```
pthread_mutex_t num_pool_one_protect;
int num_pool_one;

sem_t sem_rich_riders;
```

The Threads and Functions

Riders

As explained earlier, the rider threads wait on their conditional variables until a cab accepts their request. This requires a change in state of the rider. The state of a rider can be any one of the following:

```
#define RIDER_ST_PREMATURE (int)0
#define RIDER_ST_READY (int)1
#define RIDER_ST_RIDING (int)2
#define RIDER_ST_REACHED (int)3
#define RIDER_ST_PAYING (int)4
#define RIDER_ST_DONE (int)5
```

This state of the rider is read and modified by the rider itself, the cab threads and the server threads, and hence is protected by a mutex, one for each rider.

The rider thread executes the following function:

```
void* rider_run(void* args) {
    Rider* self = (Rider*)args;
    self->tid = pthread_self();

    pthread_mutex_t* protect_ptr = &(self->protect);
    pthread_cond_t* cv_cab_ptr = &(self->cv_cab);

    int premature_time = rand() % 10;
    sleep(premature_time);

    if (self->type == RIDER_TYPE_PREMIER) {
        printf(ANSI_RED "RIDER %d WAITING FOR PREMIER\n" ANSI_DEFAULT, self->i
```

```
d);
    } else {
        printf(ANSI_RED "RIDER %d WAITING FOR POOL\n" ANSI_DEFAULT, self->id);
    }
...
```

The rider thread waits for a random time before requesting a ride (essentially changing its state from `RIDER_ST_PREMATURE` to `RIDER_ST_READY`). This simulates riders arriving at random times.

```
...
    pthread_mutex_lock(protect_ptr);

    self->state = RIDER_ST_READY;

    struct timespec timer;
    clock_gettime(CLOCK_REALTIME, &timer);
    timer.tv_sec = timer.tv_sec + self->wait_time;

    while (true) {

        if (self->state != RIDER_ST_READY) {
            break;
        } else {
            if (pthread_cond_timedwait(cv_cab_ptr, protect_ptr, &timer) == ETIM
EDOUT){
                self->state = RIDER_ST_DONE;
                sem_post(&(self->riding));
                sem_post(&(self->paying));
                printf(ANSI_RED_BOLD "RIDER %d TIMED OUT\n" ANSI_DEFAULT, self-
>id);
                pthread_mutex_unlock(protect_ptr);
                return NULL;
            }
        }
    }
...
```

The code snippet above causes the rider thread to conditionally wait until some cab changes its state. The rider however, will leave the system after its maximum wait time is exceeded. This is achieved using the `pthread_cond_timedwait()` function.

```
...
    pthread_mutex_unlock(protect_ptr);

    // NOW STATE IS RIDING
    if (self->type == RIDER_TYPE_PREMIER) {
        printf(ANSI_YELLOW "RIDER %d RIDING PREMIER IN CAB %d\n" ANSI_DEFAULT,
 self->id, self->cab->id);
    } else {
        if (self->cab->state == CAB_ST_POOL_ONE) {
            printf(ANSI_YELLOW "RIDER %d RIDING POOL ALONE IN CAB %d\n" ANSI_DE
FAULT, self->id, self->cab->id);
        } else {
            printf(ANSI_YELLOW "RIDER %d RIDING POOL SHARED WITH RIDER %d IN CA
B %d\n" ANSI_DEFAULT, self->id, (self->cab->rider_a->id != self->id ? self->cab
->rider_a->id : self->cab->rider_b->id), self->cab->id);
        }
    }

    sleep(self->ride_time);
    sem_post(&(self->riding));

    // NOW STATE IS WAITING FOR PAYMENT
    pthread_mutex_lock(protect_ptr);
    self->state = RIDER_ST_REACHED;
    self->cab = NULL;
    pthread_mutex_unlock(protect_ptr);

    printf(ANSI_CYAN "RIDER %d REACHED AND WAITING FOR PAYMENT\n" ANSI_DEFAULT,
self->id);

    sem_post(&sem_rich_riders);
    // SOME SERVER CATCHES THIS AND MAKES IT'S STATE PAYING
```

```
    sem_wait(&(self->paying));
    // DONE
    pthread_mutex_lock(protect_ptr);
    self->state = RIDER_ST_DONE;
    pthread_mutex_unlock(protect_ptr);

    return NULL;
}
```

The code-snippet above handles the rest of the lifecycle of the rider. It waits for its ride to complete, signals the cab of the same and then posts on the semaphore `num_rich_riders` *(pardon the name!)* to allow one of the servers to process its payment, as explained earlier.

Cabs

A cab can be in any one of the following states:

```
#define CAB_ST_EMPTY (int)0
#define CAB_ST_PREMIER (int)1
#define CAB_ST_POOL_ONE (int)2
#define CAB_ST_POOL_TWO (int)3
```

The function executed by the cab threads is described below.

```
void* cab_run(void* args) {
    Cab* self = (Cab*)args;
    self->tid = pthread_self();

    while (true) {
        if (self->state == CAB_ST_EMPTY) {
            int i = rand() % self->num_riders;
            Rider* rider = self->riders[i];
            pthread_mutex_lock(&(rider->protect));

            if (rider->state != RIDER_ST_READY) {
                pthread_cond_signal(&(rider->cv_cab));
```

```
                pthread_mutex_unlock(&(rider->protect));
                continue;
            }

            if (rider->type == RIDER_TYPE_PREMIER) {
                if (pthread_cond_signal(&(rider->cv_cab)) == 0) {

                    self->state = CAB_ST_PREMIER;
                    self->rider_a = rider;

                    rider->state = RIDER_ST_RIDING;
                    rider->cab = self;

                    pthread_mutex_unlock(&(rider->protect));

                    sem_wait(&(rider->riding));

                    self->state = CAB_ST_EMPTY;
                    self->rider_a = NULL;
                    continue;
                } else {
                    pthread_mutex_unlock(&(rider->protect));
                    continue;
                }


            } else if (rider->type == RIDER_TYPE_POOL) { // HAS TO BE ONE OF TH
ESE TWO TYPES
                pthread_mutex_lock(&num_pool_one_protect);
                if (num_pool_one > 0) {
                    pthread_mutex_unlock(&num_pool_one_protect);

                    pthread_cond_signal(&(rider->cv_cab));
                    pthread_mutex_unlock(&(rider->protect));

                    continue;
                } else {
```

```
                    if (pthread_cond_signal(&(rider->cv_cab)) == 0) {
                        num_pool_one++;
                        self->state = CAB_ST_POOL_ONE;
                        pthread_mutex_unlock(&num_pool_one_protect);

                        self->rider_a = rider;

                        rider->state = RIDER_ST_RIDING;
                        rider->cab = self;

                        pthread_mutex_unlock(&(rider->protect));

                        continue;
                    } else {
                        pthread_mutex_unlock(&num_pool_one_protect);
                        pthread_mutex_unlock(&(rider->protect));
                    }
                }
            }
        } else if (self->state == CAB_ST_POOL_ONE) {
            if (sem_trywait(&(self->rider_a->riding)) == 0) {
                pthread_mutex_lock(&num_pool_one_protect);
                num_pool_one--;
                self->state = CAB_ST_EMPTY;
                self->rider_a = NULL;
                pthread_mutex_unlock(&num_pool_one_protect);
                continue;
            }

            int i = rand() % self->num_riders;
            Rider* rider = self->riders[i];
            pthread_mutex_lock(&(rider->protect));

            if (rider->state != RIDER_ST_READY) {
                pthread_cond_signal(&(rider->cv_cab));
                pthread_mutex_unlock(&(rider->protect));  // TODO: requires con
d_signal
```

```
                continue;
            }

            if (rider->type != RIDER_TYPE_POOL) {
                pthread_cond_signal(&(rider->cv_cab));
                pthread_mutex_unlock(&(rider->protect));
                continue;
            }

            if (pthread_cond_signal(&(rider->cv_cab)) == 0) {
                pthread_mutex_lock(&num_pool_one_protect);
                num_pool_one--;
                self->state = CAB_ST_POOL_TWO;
                pthread_mutex_unlock(&num_pool_one_protect);

                self->rider_b = rider;

                rider->state = RIDER_ST_RIDING;
                rider->cab = self;
            }

            pthread_mutex_unlock(&(rider->protect));
            continue;

        } else if (self->state == CAB_ST_POOL_TWO) {
            if (sem_trywait(&(self->rider_a->riding)) == 0) {
                pthread_mutex_lock(&num_pool_one_protect);
                num_pool_one++;
                self->state = CAB_ST_POOL_ONE;
                self->rider_a = self->rider_b;
                self->rider_b = NULL;
                pthread_mutex_unlock(&num_pool_one_protect);
                continue;
            }
            if (sem_trywait(&(self->rider_b->riding)) == 0) {
                pthread_mutex_lock(&num_pool_one_protect);
                num_pool_one++;
```

```
                self->state = CAB_ST_POOL_ONE;
                self->rider_b = NULL;
                pthread_mutex_unlock(&num_pool_one_protect);
                continue;
            }
        }
    }
}
```

Depending on the type of the request and its own type, the cabs either reject or accept the randomly picked up rider if it was in the `RIDER_ST_READY` state. Then the cab chooses its next state and action appropriately.

Payment Servers

Servers wait on the `num_rich_riders` semaphore until a rider signals the semaphore, indicating that it is ready to pay.

```
void* server_run(void* args) {
    Server* self = (Server*)args;

    while(true) {

        sem_wait(&sem_rich_riders);
        self->state = SERVER_ST_BUSY;
...
```

The server then iterates through the array of `Rider` structs to find a rider that is ready to pay. Exactly as many riders as the number of servers looking for them are guaranteed to exist, because of the use of semaphores.
The server then changes the state of the rider, waits for `PAYMENT_TIME` and then signals the waiting rider to know that its payment has been successful.

```
...
        for (int i = 0; i < self->num_riders; i++) {

            pthread_mutex_lock(&(self->riders[i]->protect));
```

```
            if (self->riders[i]->state == RIDER_ST_REACHED) {

                self->rider = self->riders[i];
                self->rider->state = RIDER_ST_PAYING;

                pthread_mutex_unlock(&(self->rider->protect));
                break;

            } else {
                pthread_mutex_unlock(&(self->riders[i]->protect));
            }
        }

        if (self->rider == NULL) {
            continue;
        }
        printf(ANSI_CYAN "RIDER %d PAYING ON SERVER %d\n" ANSI_DEFAULT, self->r
ider->id, self->id);

        sleep(PAYMENT_TIME);
        sem_post(&(self->rider->paying));

        printf(ANSI_GREEN "RIDER %d DONE\n" ANSI_DEFAULT, self->rider->id);

        self->state = SERVER_ST_FREE;
        self->rider = NULL;
    }
}
```

Conclusion

This simulation demonstrates synchronisation between threads through the use of mutexes and semaphores to effectively model a cab booking system.