CS 361: Computer Systems **Lab Session 10 Worksheet**
Fall 2019

Basics

1. In this lab, you will use multithreading to write an elevator simulation. As usual, start by going to `https://classroom.github.com/a/VvMhw7Ca` and checking out your repository.
2. In your repo, you will find four files: hw6.c, hw6.h, main.c, and a Makefile. If you make and then run hw6, you should see a crude simulation of an elevator. Your job in this assignment will be to improve this simulation so that it meets the following criteria:
   (a) Uses all available elevators, rather than just 1.
   (b) Finishes the final case in the "make test" test cases in at most 40 seconds.
3. You will do this by implementing a central elevator controller that responds to the function calls passenger_request and elevator_ready in hw6.c. You should change only code in hw6.c, not main.c or hw6.h

pthread_mutex_t

1. The homework currently uses a single pthread_mutex_t to make sure multiple threads don't try to do dangerous things (like getting on and off the elevator) at the same time.
2. A mutex guarantees that only a single thread will be able to run in protected sections of code at time.
3. When the system call pthread_mutex_lock is called on a pthread_mutex_t, only the first thread to call lock will be allowed to run in any code protected by the mutex. Any other thread that calls pthread_mutex_lock on the same mutex will be blocked until the thread currently in the mutex calls pthread_mutex_unlock.

Elevators and Passengers

1. Each elevator and passenger is represented by a different thread.
2. The thread for main in main.c is below

```
int main(int argc, char** argv) {
   struct timeval before;
   gettimeofday(&before,0);

   scheduler_init();

   pthread_t passenger_t[PASSENGERS];
   for(int i=0;i<PASSENGERS;i++)  {
```

```
        pthread_create(&passenger_t[i],NULL,start_passenger,(void*)i);
    }
    usleep(100000);

    pthread_t elevator_t[ELEVATORS];
    for(int i=0;i<ELEVATORS;i++) {
        pthread_create(&elevator_t[i],NULL,start_elevator,(void*)i);
    }

    /* wait for all trips to complete */
    for(int i=0;i<PASSENGERS;i++)
        pthread_join(passenger_t[i],NULL);
    stop=1;
    for(int i=0;i<ELEVATORS;i++)
        pthread_join(elevator_t[i],NULL);

    struct timeval after;
    gettimeofday(&after,0);

    log(0,"All %d passengers finished their %d trips each.\n",
                                                    PASSENGERS,TRIPS_PER_PASSENGER);
    int ms = (after.tv_sec-before.tv_sec)*1000+(after.tv_usec-before.tv_usec)/1000;
    log(0,"Total time elapsed: %d ms, %d slots\n",ms,ms*1000/DELAY);
}
```

3. As you can see, main creates a bunch of passenger threads and a bunch of elevator threads, and then calls join on each them.

4. We use the following struct to hold state for each passenger

```
static struct Passenger {
        int id;
        int from_floor;
        int to_floor;
        int in_elevator;
        enum {WAITING, ENTERED, EXITED} state;
} passengers[PASSENGERS];
```

5. As you can see, each passenger has a unique id, a floor they are on and a floor they are going to, an int which will be 1 if they are in the elevator and otherwise 0, and a state that says if they are waiting for the elevator, have entered it, or have exited it.

6. Answer the question about this code on Gradescope.

7. Each passenger thread will run the following code

```
void* start_passenger(void *arg) {
```

```
    int passenger=(int)arg;
    struct Passenger *p = &passengers[passenger];
    log(6,"Starting passenger %d\n", passenger);
    p->from_floor = random()%FLOORS;
    p->in_elevator = -1;
    p->id = passenger;
    int trips = TRIPS_PER_PASSENGER;
    while(!stop && trips-- > 0) {
        p->to_floor = random() % FLOORS;
        log(6,"Passenger %d requesting %d->%d\n",
                                            passenger,p->from_floor,p->to_floor);

        struct timeval before;
        gettimeofday(&before,0);
        passengers[passenger].state=WAITING;
        passenger_request(passenger, p->from_floor, p->to_floor,
                                                  passenger_enter, passenger_exit);
        struct timeval after;
        gettimeofday(&after,0);
        int ms = (after.tv_sec - before.tv_sec)*1000 +
                                           (after.tv_usec - before.tv_usec)/1000;
        log(1,"Passenger %d trip duration %d ms, %d slots\n",
                                                  passenger,ms,ms*1000/DELAY);

        p->from_floor = p->to_floor;
        usleep(100);
    }
}
```

8. Each passenger starts on a random floor, and then takes a number of trips on the elevator, each time picking a new random floor to go to. Note that to get the elevator, the passenger calls the passenger_request function - this is code you will write/modify in hw6.c.

9. Answer the question about this code on Gradescope.

10. Now, let's look at elevators. We use the following struct to hold information about each elevator"

```
static struct Elevator {
        int seqno, last_action_seqno; // these two are used to enforce control rules
        int floor;
        int open;
        int passengers;
        int trips;
} elevators[ELEVATORS];
```

11. The first two ints are used to make sure our elevator can't do anything crazy, like jump between floors. Next, we have an int that says which floor the elevator is on, an int which

represents whether or not the door is open, an int that holds the number of passengers the elevator contains, and an int for the number of trips it has made.

12. Each elevator thread runs the following code

```
void* start_elevator(void *arg) {
int elevator = (int)arg;
struct Elevator *e = &elevators[elevator];
e->last_action_seqno = 0;
e->seqno = 1;
e->passengers = 0;
e->trips = 0;
log(6,"Starting elevator %d\n", elevator);

e->floor = 0;//elevator % (FLOORS-1);
while(!stop) {
         e->seqno++;
         elevator_ready(elevator,e->floor,elevator_move_direction,
         elevator_open_door,elevator_close_door);
         sched_yield();
        }
}
```

13. The elevator initializes some variables, and then goes into a loop where it calls elevator_ready, and then calls sched_yield, a system call which causes it to give up the CPU. Elevator_ready is code you will write/modify within hw6.c.

14. Answer the question about this code on Gradescope.

Code you can change

1. At the top of main.c, there are some global variables (shown below). Right now these are used with our single running elevator. You will want to modify these so each elevator has their own set.

```
pthread_mutex_t lock;

int current_floor;
int direction;
int occupancy;
enum {ELEVATOR_ARRIVED=1, ELEVATOR_OPEN=2, ELEVATOR_CLOSED=3} state;
```

2. These are a mutex used to make sure two threads aren't making changes to a single elevator at the same time, an int representing where the elevator currently is, an int representing which direction it's going, an int with the number of passengers, and the elevators current state.

3. Every passenger thread calls passenger_request, below. This is code in hw6.c which you will want to modify. Let's look at what it does now.

```
void passenger_request(int passenger, int from_floor, int to_floor,
                                    void (*enter)(int, int), void(*exit)(int, int))
{
    // wait for the elevator to arrive at our origin floor, then get in
    int waiting = 1;
    while(waiting) {
        pthread_mutex_lock(&lock);

        if(current_floor == from_floor && state == ELEVATOR_OPEN && occupancy==0) {
                enter(passenger, 0);
                occupancy++;
                waiting=0;
        }

        pthread_mutex_unlock(&lock);
    }

    // wait for the elevator at our destination floor, then get out
    int riding=1;
    while(riding) {
        pthread_mutex_lock(&lock);

        if(current_floor == to_floor && state == ELEVATOR_OPEN) {
                exit(passenger, 0);
                occupancy--;
                riding=0;
        }

        pthread_mutex_unlock(&lock);
    }
}
```

4. passenger_request takes in an int representing a passenger, a floor the passenger is current on (from_floor), a floor the passenger is currently on (to_floor), and pointers to two functions, enter and exit. These functions are defined in main.c, and you will not be able to change them.

5. As you can see, right now the passenger first waits for the elevator. We use a mutex to make sure multiple passengers don't try to get on the elevator at the same time. If the elevator is on our floor, and the door is open, and no other passengers are in it, then we call the enter function, increment the occupancy, and break out of the waiting loop. (Note that every elevator holds at most one passenger.)

6. Next we take the elevator to our floor. Again, we use a lock to make sure multiple passengers do not modify an elevator at the same time. If we reach our floor and the elevator door opens, we call the our exit function, decrement the occupancy, and break out of the elevator riding while loop.

7. The exit and enter function are passed in from main - they are passenger_enter and passenger_exit in main.c. They mostly perform a number of checks to make sure the passengers and elevators are not doing things that should be impossible.

8. Every elevator thread calls elevator ready, in hw6.c. You will want to modify this code as well.

9. Answer the question about this code on Gradescope.

```
void elevator_ready(int elevator, int at_floor,
                          void(*move_direction)(int, int),
                            void(*door_open)(int), void(*door_close)(int)) {
        if(elevator!=0) return;

        pthread_mutex_lock(&lock);
        if(state == ELEVATOR_ARRIVED) {
                door_open(elevator);
                state=ELEVATOR_OPEN;
        }
        else if(state == ELEVATOR_OPEN) {
                door_close(elevator);
                state=ELEVATOR_CLOSED;
        }
        else {
                if(at_floor==0 || at_floor==FLOORS-1)
                        direction*=-1;
                move_direction(elevator,direction);
                current_floor=at_floor+direction;
                state=ELEVATOR_ARRIVED;
        }
        pthread_mutex_unlock(&lock);
}
```

10. elevator_ready takes an int representing the elevator, an int representing its current floor, and pointers to three functions, move_direction, door_open and door_close.

11. Right now we are only using one elevator, so if we are not elevator 0, we don't do anything.

12. Next, we check our current state. If we are ELEVATOR_ARRIVED, we call the door_open function and change our state to ELEVATOR_OPEN

13. If we are ELEVATOR_OPEN, we close our door by calling the door_close button and changing our state to ELEVATOR_CLOSED

14. Otherwise, we move! We check if we are at the ground or top floor, and if we are, we change direction. Then we call the move_direction function and either add or subtract one from our current floor, depending on what direction we are traveling. Next, we set our state to ELEVATOR_ARRIVED. Note that this means we currently stop and open our doors at every floor.

15. The passed in functions are elevator_move_direction, elevator_open_door, elevator_close_door in main.c. Like our other functions, they perform a number of checks to make sure our elevator isn't doing anything it shouldn't, like moving between multiple floors at a turn. They also call usleep, which will give up the CPU for some amount of time. This will allow other passenger and elevator threads to run regularly.

16. Answer the question about this code on Gradescope.

Getting started

1. Start by making sure your passengers have time to get on and off the elevator. You can do this with condition variables (pthread_cond_t), or barriers (pthread_barrier_t). A barrier-based solution seems easier. Use one barrier to make the passenger wait for the door to open, and another to make the elevator wait for the passenger to enter. (We will go over barriers in class this week.)

2. The template solution has a single set of global variables. You'll probably want to have one set per elevator. Define an elevator struct that holds all your necessary state per elevator, and make an array of such structs.

3. Handling multiple elevators should be the last item on your TODO list. An easy way to do it is to randomly decide, for each passenger, which elevator they should use, independent of everything else. Then you can treat each elevator+passengers group separately.

4. You could use an extra set of mutexes around the whole passenger_request() function to make sure only one user's request is handled by each elevator at one time.