Example Design Document: Elevator Coordination Problem

This document provides an example of a (mostly) satisfactory initial design for the Elevator coordination problem. This use to be part of Nachos project 1, but has now been replaced by the Boat problem. The design presented here is an actual project submission. It is not entirely correct, but should help to give you an idea of what the TAs want to see in an initial design document. Be sure to read the comments at the end.

Problem Description

NOTE: The description is provided here so that you will know which problems the design is trying to solve. Actual design documents SHOULD NOT include the problem description, since it's already given on the web page.

1. (25%, 300 lines) Using what you have learned from the other sections of Phase #1, you are to implement a controller for a bank of elevators, such as in Soda Hall. Seenachos.machine.ElevatorBank for details on the hardware interface, and nachos.threads.ElevatorController for details on what you are to implement. Create a thread for each elevator; these threads talk to the elevator hardware (through the interface defined in nachos.machine.ElevatorControls) to open and close the elevator doors, move the elevator between floors, etc. The elevator threads should coordinate with each other to make sure that exactly one elevator goes to pick up each set of passengers. For synchronizing the elevator threads, you are free to use semaphores, condition variables, or even the Communicator if you want (though in my experience, using a Communicator only makes things harder). You will also need to use Alarm.waitUntil() to hold open the doors for a fixed amount of time (you should hold them open for timeDoorsOpen ticks).
2. (15%, 100 lines) Since you will need to test out your code before you turn it in, you will also be required to implement elevator riders. See nachos.threads.Rider for details on what you are to implement. This part is considerably easier, because each rider is independent of all other riders, and each rider gets its ownnachos.machine.RiderControls object (as opposed to the elevator controller, which has one controls object for all the elevators, and requires synchronization). Of course, the only communication possible between the elevator riders and the elevator controller is via the physical elevator device. The elevator threads should not share any memory or synchronization variables with the rider threads.

   To test your riders and elevators, you must first call Machine.bank().init(). Then call Machine.bank().addRider() once for each rider. Finally, callMachine.bank().run() to run the simulation (this method returns when the simulation is complete).

   We have provided a GUI to help you to test your elevators and riders together. To enable it, simply call Machine.bank().enableGui() *before*Machine.bank().run().

Sample Solution

Part 1: ElevatorController class

ElevatorController will have an inner class, Elevator, to contain the code and state of each elevator. The state of an Elevator will nclude a task list (tasklist as a linked list), a direction (direction as an int), a destination (destination, an int), and an identifier (id, an int). The state of ElevatorController includes an event queue (LL, as a linked list), a boolean (seenDone), a semaphore (eventWait), and lists to store locks, condition variables, and Elevator objects for each elevator.
ElevatorController uses an interrupt handler to get events from the ElevatorBank event queue. This handler will use eventWait, initialized at zero, which will be incremented every time the handler is called. So, getNextEvent will Proberen in a loop until an event can be returned.

```
interrupt() {
    eventWait.V();
}

getNextEvent() {
    while (1) {
        if ((event == next event from controls) != null) break;
        eventWait.P();
    }
    return event;
}
```

initialize will set interrupt as the interrupt handler for controls, the ElevatorControls. It will also create threads, locks, and condition variables for each elevator and initialize eventWait to 0.

The run method will start running all of the elevator threads, then loop until all queued events are completed and the eventRidersDone has occurred. The queued events are kept in LL and will only consist of Up and DownButtonPressed events. If LL is not empty, run will cycle through the elevators at most once to find an available elevator to fulfill the requests in LL. This is done in a round robin fashion. To check an elevator, its lock is first acquired, then the elevator is checked to see if its doors are closed and itstasklist is empty. If these conditions hold, then the floor request is added to the elevator's tasklist, the elevator's direction is set to up or down (depending on which way the elevator needs to go), and the elevator is woken up (via its condition variable) just before releasing its lock.

```
if (LL.size() != 0) {
    for each elevator starting at i {
        lock[i].acquire();
        if (elev[i].doorsClosedP && elev[i].tasklist.sizeo == 0) {
            elev[i].tasklist.add(LL.removeNextEvent());
            elev[i].direction = either up or down // Up or DownButtonPressed
            cv[i] wake();
            lock[i].release();
            break;
        } else {
            lock[i].release();
        }
    }
}
```

After an event is scheduled or no elevators were found to be available, run calls getNextEvent to retrieve the next event (only if RidersDone has not already arrived). If the new event is the RidersDone event, then seenDone is set to true. If the event is an Up or DownButtonPressed, then it is added to the end of LL, but only if the same event does not already exist in LL. By weeding out duplicate events, only one elevator will be sent to handle each Up or DownButtonPressed per floor. This forces riders to push the elevator button again if an elevator comes and leaves without them.

```
if (event.type == eventUpButtonPressed || event.type == eventDownButtonPressed) {
    for each event e in LL {
        if (e.type == event.type && e.floor == event.floor) {
            break;
        }
    }
    if (did not see duplicate) {
        LL.add(event);
    }
}
```

If the event is a FloorButtonPressed, then the lock of the corresponding elevator is acquired, the request is put on the elevator's tasklist, the elevator is woken up, and then its lock is released.

```
if (event.type == eventFloorButtonPressed) {
    lock[event.floor].acquire();
    elev[event.floor].tasklist.add(event);
    cv[event.floor].wake();
    lock[event.floor].release();
}
```

Finally, if the event is an ElevatorArrived, then the corresponding elevator is woken up.

The constructor of each Elevator object will simply create an empty tasklist (a linked list), and set the initial direction to 0 (the null direction).

Elevator.run, the code that each elevator thread runs, begins by acquiring its own lock. This forces ElevatorController to wait on each elevator's lock, which is only released when an elevator sleeps. Then, the elevator will enter a loop that repeats as long as the thread is alive. First, a loop continues for as long as tasklist is not empty. (If the tasklist becomes empty, direction is set to 0 and the elevator sleeps.) So, while tasklist is not empty, we first remove a task from it. Then, the arrow of the elevator is set according to the current and destination floors (e.g. if destination floor > current floor, set up). Then, we use controls to move the elevator to the desired floor. After this call, the thread should sleep until an ElevatorArrived occurs.

```
while (tasklist.size > 0) {
    task = tasklist.removeNextEvent();
    if (current floor < destination floor) { arrow = dirUp; }
    else if (current floor > destination floor) { arrow = dirDown; }
    controls.moveto(id, task.floor);
    cv[id].sleep(lock[id]);
    ...
```

However, because both ElevatorArrived and FloorButtonPressed can wake up the thread at this point, the elevator must sleep again if the elevator has not yet arrived. If the elevator has arrived (this can be checked by remembering the destination floor and comparing it to the current floor), then the elevator should set its arrow accordingly: ifdirection has been set to either up or down, then the arrow should take on this value; if the elevator is at the top or bottom floor, the arrow should change to down or up,

respectively; otherwise, the arrow should not change. direction should be set to 0 in either case. Then, it should open its doors and set an alarm for a predefined amount of time. When this time has expired, the elevator should close its doors.

```
if (elevator has arrived) {
    if (direction != 0) { controls.setarrow(id, direction); }
        else if (at top floor) { controls.setarrow(id, dirDown); }
        else if (at bottom floor) { controls.setarrow(id, dirUp); }
        direction = 0;
        controls.open(id);
        ThreadedKernel.alarm.waitUntil(doorOpenTime);
        controls.close(id);
    } else {
        cv[id].sleep(lock[id]);
    }
}
```

Finally, because events are needed by both ElevatorController and Elevators, a common data type should be used to represent each. An array of two ints will suffice to hold a floor and a direction.

Part 2: Rider class

A rider needs to access information about its controls, the stops it has to make and which stops it has made. Add the following private variables to Rider:

```
RiderControls controls;
int[] stops;
int numStopsMade;
```

Additionally, we need a way to synchronize RiderEvents coming from the controls. We add a private Semaphore to Rider to do this.

```
Semaphore eventWait;
```

The Rider constructor assigns the passed-in controls and stops to their respective instance variables. It also initializes numStopsMade to 0 and eventWait to a newSemaphore, initially 0. Finally, it sets the interrupt handler for the controls object. The interrupt handler is set in the same way as for the ElevatorControls interrupt handler; please refer to Part 6 for this. The run method will continue to loop as long as numStopsMade is less than stops.length. The loop will monitor events and choose the proper commands to invoke on controls to make the remaining stops in stops. When all stops have been made, controls.finish will be called. Specifically, starting on some floor (and not in an elevator), for each stop in stops, check the floor number against the destination floor to determine which elevator button (up or down) to press. (If you are already on the floor you need to go to, consider yourself as having stopped at the floor). In a while loop, monitor incoming RiderEvents until you come across an event for an elevator that is going in the desired direction, has its doors open and is not full. Enter the elevator and press the button for the floor you need to stop on. When the elevator stops on the desired floor and opens its doors, exit the elevator, finishing this stop.

```
run() {
    e is the current RiderEvent
    i indexes the current stop goal, initially 0
    while numStopsMade < number of stops {
        go to the next stop
        up <- true if getFloor() < stops[i] and false otherwise
```

```
        call pressUpButton() or pressDownButton() depending on 'up'
        keep getting events until e.getDirectionDisplay is the direction desired
        if the elevator has doors open and is not full {
            enterElevator(e.elevator)
        } else {
            try to request another elevator
        }
        immediately pressFloorButton(stops[i])
        wait in a loop until getFloor == stops[i] && elevator doors open
        exitElevator(stops[i])
        numStopsMade <- numStopsMade + i
    }
}
```

Comments on the Design

What this design document gets right:

- **Good use of pseudocode.** Design document submissions SHOULD NOT include verbatim Java code; you will lose points for this. Rather, provide clear and concise pseudocode describing the primary algorithms and any special data structures used in your implementation.
- **Use of text descriptions.** It usually suffices to describe the primary algorithms and data structures of your solution in a short paragraph, then provide more specifics in the pseudocode.
- **Length.** Like the Boat simulation, the Elevator simulator was one of the most complicated portions of Phase 1. In general, the length of a design document section should increase in proportion to the difficulty and length of the solution it describes. Given this relation, each section of your design document (corresponding to a Roman numeral in the project description) should be ABSOLUTELY NO LONGER than the ElevatorController section given above. In all, your design document should be between 2,000 and 4,000 words in length.

What this design document does poorly:

- **TEST CASES!** We need to see that you've actually considered how you're going to test your implementation. Remember, the autograder test cases you see are only a fraction of the tests that will actually be used to grade your project. So unless you want to lose points, be sure to describe (in general terms) how you'll test your solution.
- Here are some sample test cases that would be appropriate for the Elevator scenario:
    1. Instantiate an ElevatorBank with 2 floors and 1 elevator. Create a single rider with a predetermined itinerary of floors to visit (e.g., 1,2,1,2). Use println()statements or the GUI interface to make sure the rider reaches all the floors in the correct order.
    2. Run the previous test with 2 riders, then 2 riders and 2 elevators, then multiple riders, elevators, and floors.
    3. Modify the previous test cases to create a random number of riders, elevators and floors. Run the tests as many times as possible, using assertion checks or debugging output to ensure that the simulation doesn't fail on rare "corner cases."
