CS 3520—Go Lift Simulator

In this assignment you will write a simulator for an elevator control system. You will build three basic component types, each running in its own goroutine and communicating with other components through channels:

1. A controller. The controller is what actually moves the elevator. It is analagous to the motor and cable assembly in a physical system.
2. A lift. Each lift component simulates a single elevator, typically one of several in a large building.
3. A floor. This represents the bank of elevators and their doors on a single floor, including the call button that a user presses to summon an elevator to that floor.

You will also make use of timers from the `time` package. Its `After` function returns a channel that will deliver a timestamp after a specified amount of time has elapsed. You will treat this almost like another component in your system, except that you will create a new one each time you need to schedule a timer.

The starter kit

I have provided some starter code, including the basic data types that will represent each type of component. You are also given the function signatures for each component.

I recommend building the components in the order they are described here, and writing stubs to test each component in isolation. Do not move past a component until you have tested all of its states and messages.

Here is the starter file:

- lift.go

You may also wish to consult the slides and the Oz code that we covered in lecture:

- Message-passing concurrency slides

The relevant section starts on slide 37 of 54.

The controller

The controller component has two states: the motor is stopped or it is running. It should call `time.After` when it needs to pause while the motor runs. It can then wait for a value to emerge from that channel to indicate that the requested amount of time has elapsed.

Its responses are based on its current state:

1. Motor stopped: the controller can receive a Step message, which tells it the floor that the lift would like to move to. The controller only ever moves a single floor at a time, so this message just lets it know which direction it should move next.
    - If the floor number is less than zero, the controller should shut down.
    - If the lift is already at the desired floor (which the controller tracks in its state), then it does nothing.
    - If the desired floor is down from the current location, it starts the motor running (changing the state), sets the floor to the next floor in the desired direction (the state of the controller always tracks the floor that it will arrive at next, not the floor it just left), and starts a timer for `Pause` duration.
    - If the desired floor is up from the current location, it does the same thing only moving up.
2. Motor running: the controller simply waits for the timer to respond, indicating that the lift has arrived at the next floor. When this happens, it sends an At message with the new floor to its lift.

The lift

The state of the lift component includes the floor the lift is currently at, the schedule of floors to visit, and whether or not the lift is currently moving toward a new floor. Note that the lift tracks whether it is headed toward a destination, while the controller tracks whether or not the motor is currently running to move toward the next adjacent floor.

The lift component always starts by waiting for an incoming message. Since a message could come in on either of its message channels, it must select between them:

```
select {
case destination := <-lift.Call:
    // respond to a Call message
case newFloor := <-lift.At:
    // respond to an At message
}
```

This will pause and wait for the next available message, regardless of which channel it comes from.

You should also implement `scheduleLast`, a simple helper function. It adds the given floor to the end of the given schedule (using `append`), but only if that floor is not already the last floor in the schedule. In either case, it returns the schedule that results.

1. Call message: the call message passes along the number of the floor that the lift should add to its schedule. If that number is less than zero, then the lift should shut down.

   Otherwise, log a message telling the lift number and the number of the floor to which it has been called.

   If the lift is already at the desired floor and it is not moving, then it should immediately send an Arrive message to the floor. Create a boolean channel to send as the message contents, and then wait for a value to be passed back, indicating that the doors are closed and it is safe to move on.

   Otherwise, add the desired floor to the end of the schedule using `scheduleLast`. Then if the lift is not already moving, send a Step message to the controller to move a floor closer to the next floor in the schedule (which may not be the one that just called). Be sure to set Moving to true.

2. At message: log a message indicating the lift number and the floor that it has just arrived at.

   Check if you have arrived at the next floor in your schedule. If so, send an Arrive message to the floor and wait for it to close the doors (you will need to make a boolean channel to do this). Record the floor you have arrived at, and remove it from the schedule (using a slice operation). If the schedule is now empty, set Moving to false, otherwise send the controller a Step message toward the next floor in the schedule and make sure Moving is true.

   If the floor you arrived at is not the next destination in your schedule, simply record the floor you are at and send another Step message to the controller.

The floor

Floors can be in one of three states:

1. Not called: the floor is in an idle state.
2. Doors open: one or more lifts are currently at this floor with their doors open, and the floor is waiting for a timer to expire before closing the doors.
3. Called: a lift has been called, but has not yet arrived.

Each floor should support two message types:

1. Call: a user waiting on a floor calls a lift (any lift—the floor randomly chooses one); the floor should respond by sending a Call message to the lift that it chooses.
2. Arrive: when a lift arrives at a floor and opens its doors, it sends an Arrive message to the floor. The message contains a boolean channel. The floor sends a value across that channel when it closes the doors and the lift is free to move again.

In addition, the floor will receive messages from a timer channel, but this is private to the floor.

Structure the floor’s main loop based on the current state:

1. Not called: if an Arrive message comes in, print a message that a lift has arrived at this floor and opened

its doors, set the state to “doors open”, and start a timer for `Pause` duration (store the timer channel in a local variable) Also, store the acknowledgement channel (the contents of the request message) in a slice. You may have other lifts arrive while the doors are still open, and you must send each of them an acknowledgement.

If a Call message comes in, print a message that the floor is summoning an elevator, and then randomly choose one of the elevators to summon. Send it a Call message and change the state to “called”.

2. Called: If another Call message is received, ignore it.

   If an Arrive message comes in, print a message that a lift has arrived, start a timer, set the state to “doors open”, and store the acknowledgement channel. This is almost identical to the response to the same message in the “not called” state.

3. Doors open: if the timer expires, print a message that the doors are closing on this floor, and send a message to each of the acknowledgement channels, i.e., one for each lift that currently has its doors open. Reset that list (to `nil`) and set the state to “not called”.

   If an Arrive message comes in, simply add the acknowledgement channel to the list of open lifts.

   If a Call message comes in, ignore it. There is already at least one lift at the floor with an open door.

The building

The building component is not really a component: you should not run an infinite loop or create a goroutine. It is merely a convenience function to create all of the other components.

`NewBuilding` takes the number of floors and lifts in the building, and returns a slice of floors and a slice of lifts, all correctly set up and ready to use.

Note that we are using 1-based numbering for lifts and floors, so make the slices big enough that entry zero can be left unused. It is important that you create the slice of floors with the correct size right at the beginning; do not use `append` to grow it as you build the floors. This way, you can hand the complete slice to each lift as you create it, and then fill in the actual values afterward. This is necessary because each lift needs a handle to all of the floors, and each floor needs a handle to all of the lifts.

Create the lifts first. For each lift, create a controller. Then create the lift (giving it a pointer to the controller), and then store a pointer to the lift in the controller object.

Next, create the floors. This is straightforward. As you assign each newly-created floor to the slice, you are implicitly handing that pointer to each of the lifts.

main

An example `main` function is provided. Delete the early return statement to have it generate a short sequence of calls. Write your own tests as well.