

Elevator Simulation Project

Task overview

Your task is to develop and submit multiple software components relating the operation of an **elevator system** within a multi-storey building. These components will communicate with each other over TCP-IP and POSIX shared memory.

These components include:

- The [car](#), which controls the functionality of a single elevator car. The elevator system may consist of multiple cars.
- The [control system](#) (also known as the **controller**), which is in charge of scheduling all elevators in the network. The elevator system will consist of a single **control system**.
- The [call pad](#), which simulates the devices on each floor that users press to call the elevator. In this simulation, the call pad is simply a command-line program that will be invoked to simulate calls to the elevators.
- The [internal controls](#), which simulates the buttons (and maintenance controls) within an elevator car and are used to allow users to open/close the car doors, hit the emergency stop button as well as put the elevator car into **service mode** (which can then be used to operate the car). In this simulation the internal controls are

represented by another command-line program that will be invoked to simulate the press of a button inside the car.

- The [safety system](#), of which there is one per car, monitors the internal conditions of the elevator and will put the elevator into **emergency mode** if those conditions are met. **This is a safety-critical component and must be programmed at a higher standard, in accordance with MISRA C guidelines.**

Each component will be a separate C program with the following names:

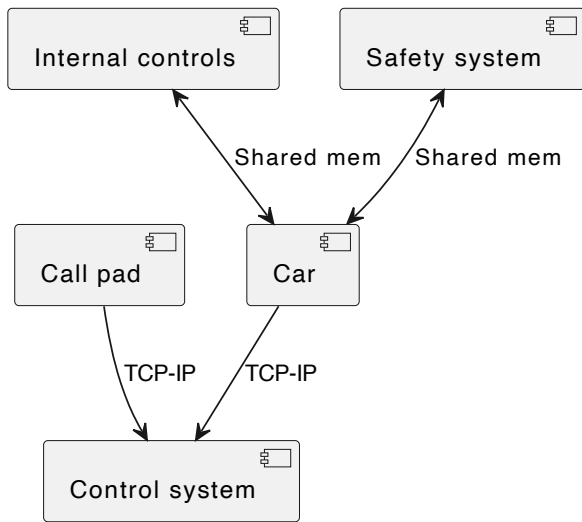
- `car`
- `controller`
- `call`
- `internal`
- `safety`

Scenario

The scenario used for this assignment is a simplified setup with the following rules:

- Floors are labelled, either with a number (for regular levels) or with a B followed by a number (for basement levels.) Floor numbers increase as you go higher, basement level numbers increase as you go lower, and B1 is connected directly to 1. So in a building that goes from B5 to 10, the floors are: B5 B4 B3 B2 B1 1 2 3 4 5 6 7 8 9 10. Floor numbers go between B99 and 999.
- Each elevator car occupies its own elevator shaft and is the only car in that shaft. Elevator cars naturally can't move between shafts.
- All elevators on a particular floor are located in the same area and there is 1 call pad per floor.
- The call pads use the **destination dispatch** system - that is, the user presses the number on the call pad indicating which level they want to go to. The call pad then tells the user which elevator has been dispatched (so they know which one to wait by).
- Certain elevator cars may be configured to stop at different floors. For example, car A might be configured to go between floors 1 and 10 while car B might be configured to go between floors 1 and 20. A user asking for floor 5 could get either car, depending on what is available, but a user asking for floor 15 will always get car B.

Connectivity



Shared memory

Each car has its own shared memory segment - this will be named `/carA`, `/carB`, `/carC` etc. depending on the elevator's name. The shared memory segment will be used to store information about the car. The segment will contain a POSIX threads **mutex** and **condition variable** (both of which must be **pshared** as multiple processes will be using them), which will be used to protect the structure and indicate to components that something has changed about its contents.

The internal controls will use the shared memory segment to tell the car that the passengers have requested that the doors be opened/closed, and also to control the elevator when it is in service mode.

The safety system will watch the internal state of the car, including the pressing of an emergency button, and react to set the car into emergency mode.

The shared memory segment has the following structure:

```

typedef struct {
    pthread_mutex_t mutex;           // Locked while accessing struct contents
    pthread_cond_t cond;             // Signalled when the contents change
    char current_floor[4];           // C string in the range B99-B1 and 1-999
    char destination_floor[4];       // Same format as above
    char status[8];                  // C string indicating the elevator's status
    uint8_t open_button;             // 1 if open doors button is pressed, else 0
    uint8_t close_button;           // 1 if close doors button is pressed, else 0
    uint8_t door_obstruction;        // 1 if obstruction detected, else 0
    uint8_t overload;                // 1 if overload detected
    uint8_t emergency_stop;          // 1 if stop button has been pressed, else 0
    uint8_t individual_service_mode; // 1 if in individual service mode, else 0
    uint8_t emergency_mode;         // 1 if in emergency mode, else 0
} car_shared_mem;
  
```

The elevator's status will be one of the following NUL-terminated C strings:

- `Opening` - indicating that the doors are currently in the process of opening
- `Open` - indicating that the doors are currently open
- `Closing` - indicating that the doors are closing
- `Closed` - indicating that the doors are shut
- `Between` - indicating that the car is presently between floors

The car component will create and initialise this shared memory segment. The internal controls and safety system will open this shared memory segment and use it.

When accessing the shared memory segment from any component, two rules need to be followed at all times:

- The mutex must be acquired when reading OR writing data in the segment
- The condition variable must be signalled (use broadcast as there may be multiple observers) after changing data in the segment

The former rule ensures the consistency of the data; the latter rule allows interested observers (like the car itself) to be notified that something has changed.

Components that need to wait for changes in the shared memory structure (car, safety system) must not endlessly loop checking these values (as this uses up a lot of CPU unnecessarily, as well as causing problems for other processes as checking the state requires locking the mutex) and should not simply incorporate a short delay between checks (because if the delay chosen is too small it will have the same problem, and if the delay chosen is too large the component will have poor responsiveness - and there is no delay that is correct in all situations.) Instead, if the **only** thing the component is waiting for is the internal structure it should lock the mutex and wait on the condition variable. If your component needs to do other things as well as monitor shared memory, consider either using `pthread_cond_timedwait` or another thread.

TCP-IP

The control system is a TCP-IP **server** which will run on port 3000. The call pad and car will both connect to this server, in different ways and for different reasons.

- Whenever someone presses a button on a call pad, the call pad will connect to the control system, send a message telling the control system which floor the call pad is on and which floor the user requested. The control system will send an immediate reply indicating which elevator has been dispatched to handle this call, or an error, if no elevators could be dispatched. **A new connection will be made every time the call pad is used.**
- The car will attempt to connect to the control system immediately, and will maintain this connection as long as the car is operating. If the connection is broken, the car will attempt to reconnect. The car will provide the elevator with information about the particular car it is upon connecting. The elevator will then send the car messages, telling the car to go to a particular floor.

The following protocol is used for sending messages: each message begins with a 32-bit unsigned integer in **network byte order** indicating the number of bytes of message that will follow. The following bytes will contain the message, which will be an ASCII string. The string is **not** NUL-terminated. To see example code using this protocol, see the `server.c` and `client.c` files in [8.L Weekly Live Lecture \(https://canvas.qut.edu.au/courses/16677/pages/8-dot-l-weekly-live-lecture\)](https://canvas.qut.edu.au/courses/16677/pages/8-dot-l-weekly-live-lecture). The exact format of the messages is documented in the specification for the respective components.

In this assignment all components will run on the same machine, and therefore the call pad and car will both just connect to 127.0.0.1 or localhost on port 3000.

It is important that you follow the protocols described in this specification because your components will be tested both together and in isolation. So if you invent your own protocol allowing your control system to talk to your cars, those two components might work together, but will fail in isolation.

Makefile

You must provide a makefile - named either `GNUmakefile`, `makefile` or `Makefile` - in your submission. We will place all of your submission files in a directory, `cd` into that directory and enter `make` to build your program. This will happen automatically, so make sure that it works.

Here is how we expect your makefile to work:

- If we just type `make` all 5 components will be built and have the filenames listed below
- If we type `make car` the elevator car component will be built and the resulting filename of the executable will be `car`
- If we type `make controller` the control system component will be built and will have the filename `controller`
- If we type `make call` the call pad component will be built and will have the filename `call`
- If we type `make internal` the internal controls component will be built and have the filename `internal`
- If we type `make safety` the safety critical component will be built and have the filename `safety`

Feel free to add additional dummy targets, such as `make clean` to delete objects/executables, but this is optional.

Car component

The car component is executed with the following command-line arguments:

```
./car {name} {lowest floor} {highest floor} {delay}
```

- **name** - this argument is the name of the particular elevator car. The name is used for two purposes - it is used to identify the elevator to the user when one is dispatched (so the user knows which elevator to wait for). Cars will typically be named things like A, B and C, but also might be named things like Service. The name is also used to determine the shared memory object name (the string that is passed to the first argument of [shm_open](https://www.man7.org/linux/man-pages/man3/shm_open.3.html) https://www.man7.org/linux/man-pages/man3/shm_open.3.html) - the shared memory object name will be `/car{name}` - e.g. if the name is A the object will be `/carA`
- **lowest floor** - this argument is a string in the format # or B#, in the range B99 to 999. This is the lowest floor that this car can reach, and also the floor that the car is assumed to start at. (In this simulation elevator cars can access any floors within their range.)
- **highest floor** - this argument is a string in the format # or B#, in the range B99 to 999. This is the highest floor that this car can reach.
- **delay** - this argument is the number of **milliseconds** that various operations are simulated to take. Affected durations are:
 - the amount of time it takes for the doors to open
 - the amount of time the doors stay open (in normal operation)
 - the amount of time it takes for the doors to shut
 - the amount of time it takes for the car to travel 1 floor (e.g. from floor 6 to floor 7 or from floor 1 to floor B1)

So, for example, if the car component is invoked with:

```
./car A 1 8 1000
```

This will mean that the car is named A, uses the shared memory object `/carA`, can access floors 1 through 8, and it takes 1000 milliseconds (1 second) for the doors to open, stay open, close and for the car to move from one floor to the next floor.

The delay is configurable for the purposes of enabling simulations - it means that you can set the time to 1000 or 500 or a similar number when testing so you can see what is happening, but our testing runs can set it to very low values so many tests can be performed quickly.

Shared memory initialisation

The car is essentially the 'owner' of its shared memory structure - when the car starts, it creates the shared memory object and initialises all of the values. The layout is as given above (you can just paste that struct into your

code and cast the shared memory pointer you get into its type). The default values of the struct are to be as follows:

- **mutex** - initialised with `pthread_mutex_init` and with the `pshared` attr enabled
- **cond** - initialised with `pthread_cond_init` and with the `pshared` attr enabled
- **current_floor** - initialised with the car's lowest floor
- **destination_floor** - initialised with the car's lowest floor
- **status** - initialised with `Closed`
- **open_button** - initialised with 0
- **close_button** - initialised with 0
- **door_obstruction** - initialised with 0
- **overload** - initialised with 0
- **emergency_stop** - initialised with 0
- **individual_service_mode** - initialised with 0
- **emergency_mode** - initialised with 0

Manual operation

The car may be in a system with or without an controller - for testing purposes, and it is also possible that the elevator controller will fail. Under these conditions the open and close buttons will still work, and an elevator technician will be able to put the car into individual service mode, which will allow the technician to control the elevator (move it between floors etc.) from the internal controls.

The internal controls component will talk to the car by changing values in the shared memory structure. Values that may be changed by the internal controls component are:

- **open_button** - set to 1 when the open doors button is pressed. The internal controls will only set this to 1 - after detecting that the value has been set to 1, the car must change it back to 0
- **close_button** - same as above, but with the close doors button
- **individual_service_mode** - set to 1 when the elevator is put into individual service mode and 0 when it is taken out of this mode. The internal controls will set both of these values
- **destination_floor** - only set by the internal controls when the car is in individual service mode
- **emergency_mode** - set to 0 when the elevator is put into individual service mode

Communication with the controller

When the car component starts up, it must attempt to connect to the controller (over TCP, on 127.0.0.1, port 3000). It's possible it will not be able to connect, as the controller may not be running. In this case it should attempt to reconnect once every (delay) ms. The TCP interaction should probably happen in a separate thread so that the other operations that the car needs to perform are not affected. Once the controller has established a connection, it must send the following message (using the length-prefixed protocol described above):

```
CAR {name} {lowest floor} {highest floor}
```

This will register the car with the controller (so the controller knows that it has a car capable of servicing the range of floors provided). This connection will then be maintained indefinitely unless the car is put into individual service mode, at which point the car will disconnect from the network (and reconnect once the car is no longer in individual service mode). Each time the car connects to the controller it needs to send the above message.

The controller will send scheduling requests to the car. These will be in the form of length-prefixed messages:

```
FLOOR {floor}
```

Upon receiving this message, the car will change its destination floor to the floor specified. If the car is already on the floor indicated, the car will open its doors.

The car will also send messages to the controller, letting the controller know the car's current status (to help scheduling decisions). These will be in the form of length-prefixed messages:

```
STATUS {status} {current floor} {destination floor}
```

(The 'status' value is the string in the shared memory struct, i.e. one of `Opening`, `Open`, `Closed`, `Closing`, `Between`.)

One of these messages will be sent to the controller immediately after the CAR initialisation message, as well as every time any of these values change, or if (delay) ms have passed since the last message. This also functions as a way to determine if the controller is still functioning - so the car can close the socket and start trying to reconnect to it. You most likely want to use `signal(SIGPIPE, SIG_IGN);` so your car component doesn't crash when the write calls fail.

Normal operation (when the elevator is not in individual service mode)

In the elevator's default state, the doors are closed and the destination floor is the same as the current floor. The car will remain in this state until something changes.

If the destination floor is different from the current floor and the doors are closed, the car will:

- Change its status to `Between`
- Wait (delay) ms
- Change its current floor to be 1 closer to the destination floor, and its status to `Closed`
- If the current floor is now the destination floor:
 - Change its status to `Opening`
 - Wait (delay) ms
 - Change its status to `Open`
 - Wait (delay) ms
 - Change its status to `Closing`
 - Wait (delay) ms
 - Change its status to `Closed`

(Note: the above sequence is the typical progression. However, the **safety system** can also change the car status (e.g. if the door was closing and there is an obstruction, the safety system may set the door to opening again. For this reason after the delay the car must check the status again to ensure that it has not changed. If it has changed, the next step should be the next logical one - for example, if the car transitioned back to `Opening` it should go through `Open`, `Closing`, `Closed` again.

If a FLOOR message arrives and the floor specified is the same floor the car is already on, the car will perform the 'current floor is now the destination floor' steps above - that is, the doors will open, stay open for (delay), then close. This behaviour is necessary because elevator cars are by default closed, but if the car is on floor 1 and a passenger on floor 1 wants to go to floor 2, the doors will need to open so the passenger can get in.

New destinations may arrive at any time, but if the controller has multiple stops in mind for the car, it will send the new request after the car changes to `Opening`. Once the car reaches `Closed`, if the destination is different to the current floor, it must immediately repeat the above steps.

- If the open button is pressed:
 - If the status is `Open` the car should wait another (delay) ms before switching to `Closing`.
 - If the status is `Closing` or `Closed` the car should switch to `Opening` and repeat the steps from there
 - If the status is `Opening` or `Between` the button does nothing
- If the close button is pressed
 - If the status is `Open` the car should immediately switch to `Closing`

If a new destination arrives while the car is in the Between status, that destination will not replace the car's current destination until the car reaches the next floor.

Individual service mode

The car may be put into individual service mode at any time. When this happens, if the car is currently Opening, Closing or Between it will continue this until the car doors have fully opened/closed or the car has moved to the next floor. After this, if the car was in Between it must change its status to Closed and the destination floor must be set to the current floor. If the car is Open or Closed it will stay that way - the car doors will not open or close automatically when the car is in individual service mode - the technician must press the buttons to manually open/close the doors. Opening/closing works in the normal way - the status first changes to Opening/Closing, there is a delay of (delay) ms and then the status is set to Open/Closed.

The technician can manually make the elevator go up and down (but only when the doors are shut) - the way this happens is that the internal controls will set the destination floor to a floor that is one higher or lower than the current floor. The car must detect that the destination floor has changed and, if the car is currently Closed, it must set the status to Between, wait (delay) ms, set the current floor to the destination floor and set the status to Closed.

Note that the technician may press up or down even if the car is on the top/bottom floor. In this case the destination floor will be set to a floor higher than (highest floor) or lower than (lowest floor) - if that happens the car must simply reset destination floor to current floor. Under no circumstances can the car's current floor go outside the range of (lowest floor) to (highest floor).

When the car is pulled out of individual service mode, if the car's current status is Opening, Closing or Between it will continue that until the status is Open or Closed. Then, if the status is Open it will continue from the 'Change its status to Open' step in the normal operation section.

If the car is connected to the controller when it is put into individual service mode, will send the following length-prefixed message before disconnecting:

```
INDIVIDUAL SERVICE
```

Emergency mode

The car does **not** have to check the emergency_stop variable (as the safety system is responsible for this). However, the safety system may set emergency_mode to 1. In emergency mode the elevator will cease all operations. A technician will come along, pull the elevator out of emergency mode and put it into individual service mode eventually. If the car is connected to the controller, it must send the following length-prefixed message to it before disconnecting:

```
EMERGENCY
```

Emergency mode is similar to individual service mode - the open/close buttons will still work, and the doors will not close automatically. However the car will not move in emergency mode. The internal controls can be used to pull the car out of emergency mode by putting it into individual service mode.

Program termination

The program can be terminated using Ctrl+C (which invokes the SIGINT signal). When this happens, the program must delete the shared memory segment for that car (use `shm_unlink()`).

Call pad component

The call pad component is executed with the following command-line arguments:

```
./call {source floor} {destination floor}
```

- **source floor** - the floor that the call pad resides on. This is the floor that the user that called the elevator is currently on.
- **destination floor** - this is the floor the user wants to get to

For example, `./call 1 B2` will call an elevator to come to floor 1, pick up a passenger, then take that passenger to floor B2.

The call pad does this by connecting to the controller via TCP (on 127.0.0.1, port 3000) and sending a length-prefixed string in the following format:

```
CALL {source floor} {destination floor}
```

The controller will then respond in one of two ways:

```
CAR {car name}
```

If this is the response, an elevator has been dispatched to handle this request. The call pad should print out a message such as:

```
Car {car name} is arriving.
```

Alternatively, the controller might respond with this:

```
UNAVAILABLE
```

If that is the case, there was no car capable of servicing this request. The call pad should print out a message such as:

```
Sorry, no car is available to take this request.
```

Both floors must be in the floor name format described earlier and be in the range B99 to 999. If either floor is invalid, print out an appropriate error message:

```
Invalid floor(s) specified.
```

The floors selected must be different. If both the source and destination floors are the same, print out an appropriate error message:

```
You are already on that floor!
```

The other thing that may happen is that the controller may be offline or the call point may be unable to connect to it. If that is the case, print out a message like:

```
Unable to connect to elevator system.
```

Whatever the outcome, the program immediately terminates afterwards, and the user can invoke it again if they want to call an elevator again.

Controller component

The controller component is executed without any command-line arguments:

```
./controller
```

The controller's job is to handle the scheduling of elevators to ensure that passengers get to their destinations in a reasonable timeframe.

The controller does not initially know anything about the building or the elevators. As cars connect to it, it will update its records to know which cars service which floors, so that when requests come in it can dispatch the correct cars to service them.

When the controller starts up, it will bind port 3000 and listen on it with a TCP socket for clients to connect. There are two types of clients that will connect to the controller:

- **Cars** will connect to the controller, provide some information about themselves, then remain connected in order to provide status updates to the controller and to receive destinations.
- **Call pads** will connect to the controller, provide the call pad's floor and destination, receive a dispatch notification and then disconnect. One connection will be made each time a user calls an elevator.

The protocols used by these components are described in the respective sections for those components.

Cars will send regular status update messages to the controller, which the controller will keep track of - so it knows where each car is, where it is going and its status.

If a car is put into individual service mode, goes into emergency mode or the TCP connection is closed for any other reason, the car is considered to be out of service and will be no longer scheduled. If this happens, the controller should forget about that particular car, If the car reconnects it will provide its information again and it should be treated like a new car.

(Note: There is no need to schedule other cars to handle the queue of an elevator that has left service. The passengers will figure it out and press the call buttons again.)

Elevator scheduling

Cars only keep track of their next destination, and if they receive a new destination it will overwrite the old one. For this reason the **controller** has to do the long-term scheduling of elevators in the system.

For example, let's assume a simplified system of only 1 elevator. A group of 3 users arrive on floor 1 and they press the buttons for floors 2, 3 and 4. The ideal approach is for the car to go to floor 1 (if it is not already there), open its doors, then travel to floors 2, 3 and 4 in that order. After the car doors close on floor 4 it will wait there.

Because the car is not able to remember to go to floors 1, 2, 3 then 4 in that order, the controller will send the message **FLOOR 1**. Then, once the car opens its doors on floor 1, it will send the message **FLOOR 2**, and so on. So the controller will keep a queue for each elevator car and will tell the car to go to the next floor in its queue upon arriving.

When a call arrives for an elevator it has a **source floor** and a **destination floor**. The controller's first task is to find an car that can access both of those floors. If no cars can service both floors the controller will respond to the request with the message **UNAVAILABLE**. If a car has the capability of visiting both floors, the controller will respond with the request **CAR {car name}**, to let the user know which elevator to wait at. If there are multiple cars, the controller will have to choose one of them to take the request.

The controller will maintain a queue for each car. When the controller chooses a car to respond to a call, it will enqueue both the source floor and the destination floor.

If the car is already going to visit a floor it shouldn't be added to the queue twice - however, there are **two** important corollaries to that:

- **The car must visit the source floor before visiting the destination floor.** Otherwise the passenger won't be able to get into the car.
- **The car must not change direction between visiting the source floor and visiting the destination floor.** Passengers that want to go up will not enter an elevator car that is going down!

After that, the next important thing to consider is where to add the floors to the queue. The key concept here is to **minimise changes of direction**. Inserting a new floor between the ones the car is already going to visit makes sense. However, this needs to be done carefully, because we do not want to disrupt any other passengers' pre-existing routes.

Suggested scheduling approach:

Here's an appropriate approach for queuing floors, which follows the requirements described above.

- The important thing about the floor queue that we use is that we register whether a floor is being stored as a **'up' floor** or as a **'down' floor**, corresponding to which direction the passenger taking the elevator was going. If a request comes for a lift from 3 to 2 we might store that as D3 D2 as the elevator must be going down or the passenger won't take it.
- We **don't** merge adjacent D entries and U entries of the same floor. Let's say someone else calls for a lift from 2 to 4. Obviously this means a total of 3 stops for the elevator, but in the queue, we would store this as D3 D2 U2 U4. In reality, the elevator will only make 3 stops- on floors 3, 2 and 4- but the way we can handle this is simply by removing D2 and U2 both from the queue when arriving at that floor.
- When we request arrives, we note the **direction** of the request. A request from 3 to 2 is going **down**, while a request from B4 to 1 is going **up**. This is important because we can only insert these floors in the queue in a continuous run of floors going in the same direction.
- We begin by iterating from the start of the queue, looking for a place to put the **from** floor. For this purpose we actually consider the elevator's current floor as the first member of the queue.
 - If the elevator's status is Between we consider the current floor to be the next floor the elevator would go to - so if the floor is currently on floor 4 and the destination floor is floor 7 the virtual first member of the queue will be 5.
 - However, if the first (real) member of the queue is the destination floor **and** the status is 'Between', then we consider the first member of the queue to be where the queue starts, and there's no virtual first member before it. This is because the car is already on its way to that floor and there's no room for anything else to happen before then.
 - This virtual first member of the queue needs to have a direction as well. If the elevator is in 'Between' status you know the direction it's going in - use that. Otherwise, base the direction by looking whether the car would have to go up or down to get to the first real entry in the queue.
 - If the first real entry in the queue is the same floor as the current floor (usually this is only the case very temporarily), just take that entry's direction.
 - If the queue is empty, set the direction to the direction being requested by the call.

Now, as an example, the queue might be [U6 U8 D7 D5 D4 D3 U1 U2 U4] and the current car floor is 5. Because the first entry (U6) is higher than 5, the car would need to go up, so we treat it as U5:



The queue will consist of some number of contiguous blocks of up/down floors. The up blocks are in ascending order and the down blocks are in descending order. The goal here is to find the first block where both the 'from' and 'to' floors can be inserted, then insert the floors into the correct positions in there (which could be at the start or end of the block). The first block in the queue is the only complex one, as we cannot insert anything before the first element (remember this is the virtual entry, so the car is already here, or about to be)- which means that some

pairs of floors will not be able to be added. If we want to add U7 U9 that can go in the first block just fine (between U6 and U8, and after U8 respectively). On the other hand, U3 U7 cannot be added to the first block and will therefore need to be added to the 3rd block (as the 2nd block is a down block, they can't be added there). The queue will never feature more than three blocks (because any new floors will always fit in one of them), although it may have fewer than that, and if there are 3 it could be either in a UDU or DUD configuration.

If there are 2 or fewer blocks, it is possible that the two floors can't be added to any of them (e.g. if they don't fit in the first block and the second block is in the wrong direction, for example). In that case just add the two floors in the order {from, to} to the end of the queue. This will make a new block.

A special case is when the from floor is equal to the current floor (the virtual first item in the queue) and in the same direction. If that's the case, the correct behaviour depends on the car's status:

- Closed/Opening/Open - insert the 'from' and 'to' elements into the first block.
- Closing - it's too late, so the from and to floors will need to be inserted into the 3rd block.
- Between - insert the 'from' and 'to' elements both into the first block. The 'current floor' is the one that the elevator is currently heading to, due to the special 'Between' handling above, so inserting the elements in there.

Every time you update the queue, make sure that the car's destination floor is the same as the next item in the queue. Otherwise send it a FLOOR message, but don't remove an item from the queue until the car arrives at that floor.

Multi-car scheduling

When multiple cars can service the source and destination floors, the approach to choosing a car is up to you. You could pick the car that would be disrupted the least by servicing another floor, or pick a car that will be able to get to the floors in question the fastest etc. Document your choice and what lead you to make that choice in the comments of your code.

If, as a result of a queue change, the first floor in the car's queue differs from its current destination, a new FLOOR message should be sent to the car as soon as possible. Otherwise, when the car reaches the floor at the start of that car's queue and starts opening its doors, that floor should be removed from the queue and the next floor in the queue send in a FLOOR message.

Program termination

The program can be terminated using Ctrl+C (which invokes the SIGINT signal). When this happens, it should properly close the listening socket so the address can be reused.

Address reuse

Before `bind()` ing the listening socket in the first place, you will want to enable address reuse:

```
int opt_enable = 1;
setsockopt(sockfd, SOL_SOCKET, SO_REUSEADDR, &opt_enable, sizeof(opt_enable));
```

This will make it easier to develop the controller software (as you will not have to wait around for the port to become available again) and will be less likely to cause problems during testing, where we will launch and terminate the the controller many times in quick succession.

As with the car component, you should use `signal(SIGPIPE, SIG_IGN);` so your controller component doesn't crash when reads/writes fail (as cars may disconnect at any time).

Internal controls component

The internal controls component is executed with the following command-line arguments:

```
./internal {car name} {operation}
```

- **car name** - the name of the car the internal controls are for
- **operation** - the button or control to invoke

The car name is used to access the shared memory segment /car{car name} as described earlier.

The operation is one of the following:

- **open** - sets **open_button** in the shared memory segment to 1
- **close** - sets **close_button** in the shared memory segment to 1
- **stop** - sets **emergency_stop** in the shared memory segment to 1
- **service_on** - sets **individual_service_mode** in the shared memory segment to 1 and **emergency_mode** to 0
- **service_off** - sets **individual_service_mode** in the shared memory segment to 0
- **up** - sets the destination floor to the next floor up from the current floor. Only usable in individual service mode, when the elevator is not moving and the doors are closed
- **down** - sets the destination floor to the next down from the current floor. Only usable in individual service mode, when the elevator is not moving and the doors are closed

If the shared memory segment for the car is not there, the program must display an appropriate error, like:

```
Unable to access car {car name}.
```

If the up or down command is entered but the car is not in individual service mode, the program must display an appropriate error, such as:

```
Operation only allowed in service mode.
```

If the up or down command is entered but the doors are not closed, the program must display an appropriate error, such as:

```
Operation not allowed while doors are open.
```

If the up or down command is entered but the elevator is between floors, the program must display an appropriate error, such as:

```
Operation not allowed while elevator is moving.
```

If the argument specified is not one of the 7 operations specified above, an appropriate error message such as this must be printed:

```
Invalid operation.
```

Whatever the outcome, the program immediately terminates afterwards, and the user can invoke it again if they want to invoke another internal control.

Safety system component

The safety system component is executed with the following command-line argument:

```
./safety {car name}
```

- **car name** - the name of the car the safety system is in

The car name is used to access the shared memory segment /car{car name} as described earlier.

If the shared memory segment for the car is not there, the program must display an appropriate error and then terminate:

```
Unable to access car {car name}.
```

Otherwise the system loops endlessly, waiting on the condition variable and checking the shared memory segment values and ensuring that everything looks reasonable. If something is wrong, it must take the appropriate action.

Here are the fields the safety system needs to check on, and the actions it needs to take:

- door_obstruction is 1 and status is Closing:
 - An obstruction was detected. Set status to Opening
- emergency_stop is 1 and emergency_mode is 0:
 - Print a message to inform the operator, e.g.:

The emergency stop button has been pressed!
 - Put the car into emergency mode
- overload is 1 and emergency_mode is 0:
 - Print a message to inform the operator, e.g.:

The overload sensor has been tripped!
 - Put the car into emergency mode
- emergency_mode is not 1 and any of the following is true: current_floor or destination_floor does not contain a valid floor string, or status is not one of the 5 valid statuses, or any of the uint8_t fields in that struct that are only supposed to be either 0 or 1 has a value of 2 or greater, or door_obstruction is 1 and status is something other than Opening or Closing:
 - Print a message to inform the operator, e.g.:


Data consistency error!
 - Put the car into emergency mode

This component is considered to be **safety-critical** and must be programmed to a higher standard accordingly. See the Safety Critical lecture and practical for more information about safety-critical programming.

The main source file for this component needs to feature a block comment near the top detailing any exceptions or deviations to safety-critical standards in the code and justifying your decision to deviate. This will be taken into account when marking your code for safety-critical adherence.

Testing your submission

As this project consists of multiple components, none of which can work in isolation, it can be difficult to test your work. For this reason, you are provided with some test harnesses, which are designed to test individual and in-progress components.

[asgn2-testers.zip \(https://canvas.qut.edu.au/courses/16677/files/5156056?wrap=1\)](https://canvas.qut.edu.au/courses/16677/files/5156056?wrap=1) 
(https://canvas.qut.edu.au/courses/16677/files/5156056/download?download_frd=1)

Extract this directory somewhere, navigate to the directory and type `make`. This should build all of the testers. To run them, navigate to the directory where your assignment components are and type in the full or relative path to the tester you want to run. I recommend extracting the testers zip into the directory where you are working on your assignment. They will be extracted into a directory named 'test' and you can then run them by typing, for example, `test/test-call` (to test your 'call' component').

The testers work by imitating the other components that your component needs to talk to - for example, test-call will start a TCP server for the purposes of pretending to be the controller component, then report the messages that come from the call component and return fake output.

The testers print out lines starting with `###` - these lines show the output that the tester expects to see on the next line (either produced by the tester itself, or by the program being tested.) If the messages printed out by your programs exactly match the examples given in this specification, it will make checking the output of the testers easier.

Here is an example (successful) run of the test-call tester:

```
$ test/test-call
### Unable to connect to elevator system.
    Unable to connect to elevator system.
### RECV: CALL B21 337 : Car Test is arriving.
    RECV: CALL B21 337 : Car Test is arriving.
### You are already on that floor!
    You are already on that floor!
### RECV: CALL 416 B68 : Sorry, no car is available to take this request.
    RECV: CALL 416 B68 : Sorry, no car is available to take this request.
### Invalid floor(s) specified.
    Invalid floor(s) specified.
### Invalid floor(s) specified.
    Invalid floor(s) specified.
### Invalid floor(s) specified.
    Invalid floor(s) specified.
### Invalid floor(s) specified.
    Invalid floor(s) specified.

Tests completed.
```

We can tell the test is successful because each line of output matches the example line and because it gets to the end and prints 'Tests completed.' - if you run the test and it doesn't get that far, it crash or froze partway through.

You can read the source code of each test to see exactly what it is trying. In this case, test-call does the following:

- Before the server is launched, it issues the command `./call B1 3`. This is a valid pair of floors, but the controller is not running so it prints out an error message.
- After starting the server up, it issues `./call B21 337`. This is a valid pair of floors. The first part of the line displays the message received by the fake controller, while the second part of the line contains the output printed out by the call component. Because the fake controller sent back the message `CAR Test`, the component prints out that the car Test is arriving to service the request.
- It then tries `./call 152 152` - here the source and destination floors are the same, so the component prints an error message.
- It then tries `./call 416 B68` - these are also valid floors, but this time the fake controller sends `UNAVAILABLE` as the reply, so the call component prints out a message explaining that no car is available to service this request.
- The next three tests try three invalid invocations: `./call L4 8` (invalid because the L at the start is not supposed to be there), `./call B100 B98` (invalid because B100 is lower than the lowest floor in the simulation, which is B99) and `./call 800 1000` (invalid because 1000 is higher than the highest floor in the simulation, which is 999.)

These tests are also run on your components when you submit to Gradescope, but you will likely find it more convenient to run them on your own machine during development, so you can more easily debug your programs.

Correct code may differ in output from the output expected by these testers. For one thing, the output printed out by your program is not strictly required to be the same as the examples we give - you need to print out appropriate messages where required, but it is not required that they precisely match what we use (though using the same messages will make testing your code with the testers easier.) Another issue is that some of the behaviours are subject to **race conditions** and the timing may be slightly different, even if the elevator does then proceed to travel in the same way as we expect. Just keep this in mind - minor differences between your output and the testing data may not mean your implementation has a bug.

Here is a summary of the testers:

- **test-call:** Tests the 'call' component. This tester binds port 3000 and will not function correctly if another program (e.g. a real controller component) is running.
- **test-internal:** Tests the 'internal' component by simulating the shared memory segment of a car. It tests all of the different commands and displays the state of the shared memory segment each time it is modified, to ensure that the component made the correct modifications.
- **test-safety:** Tests the 'safety' component by simulating the shared memory segment of a car and makes modifications to simulate various safety events to ensure the safety system has the correct response each time.
- **test-car-1:** Tests the 'car' component's shared memory and door operations, as well as its ability to run without a controller. As long as the doors open and close properly when buttons are pressed and the delay is handled correctly, your car component should pass this test.
- **test-car-2:** Tests the 'car' component during individual service mode. Once again, there is no controller present.
- **test-car-3:** Tests the 'car' component purely from the perspective of the controller, testing that the component registers itself with the component correctly, sends updates and handles requests to travel to floors correctly.
- **test-car-4:** Tests the 'car' component with both a controller and in individual service mode, checking that the car component sends the correct updates and handles disconnecting and reconnecting when put in and taken out of individual service mode / emergency mode.
- **test-car-5:** Tests that the 'car' component correctly unlinks the shared memory segment when terminated with an interrupt signal (SIGINT).
- **test-controller-1:** Tests that the 'controller' component is listening on the correct port and is capable of dispatching a single elevator car and sending FLOOR messages at the right time.
- **test-controller-2:** Tests that the 'controller' component can handle three elevator cars at the same time and dispatch requests to the car that can handle them.
- **test-controller-3:** Tests that the 'controller' component correctly handles a car being put into/taken out of emergency mode, and is able to successfully remove cars from its scheduling pool when this happens (and readd them later).
- **test-controller-4:** Tests that the 'controller' component can correctly schedule a single car and queues up floors in the correct order and with the correct timing.
- **test-sched:** Tests your 'call', 'car' and 'controller' components together in a unified test provides a simulation of passengers that displays how long the simulated passengers had to wait for an elevator and how much time they spent in the elevator. This tester is heavily configurable; see below.

There are no marks directly associated with passing these tests - they are provided for your benefit, to make it easier to get your components working correctly.

display-cars

There is also another program in the testers zip that is not built automatically when typing 'make', but you can compile it separately. This is a program that uses ASCII graphics to monitor the state of the elevator cars by inspecting shared memory.

To build it, you will first have to install the **ncurses** library, which is a library providing console drawing operations. To install this on Ubuntu type

```
sudo apt install ncurses-dev
```

Followed by `make display-cars` to compile the program. Then run it:

```
./display-cars {lowest floor} {highest floor}
```

It will continuously monitor the shared memory regions and display the current status of all elevators in your system. You can use this for testing the behaviour of your scheduling system.

```

...
./display-cars - test
4
3
2
1
====( B )====
||...||...||
||...||...||
====Closed====
====( C )====
||...||...||
||...||...||
====Closed====
====( A )====
||...||...||
||...||...||
====Closed====

```

test-sched

The scheduling tester provides a simulated test of your call, car and controller components- to evaluate how well they work together, to put them under more pressure than the existing tests (which can be hardcoded around to some extent) and to evaluate the multi-car scheduling performance. This tester simulates a number of passengers, each appearing at a random time on a random floor and with a random destination in mind. Each passenger will invoke the 'call' component to ask for an elevator, and will then check that elevator's shared memory segment, waiting until the doors open on the passenger's floor. The passenger will then wait until the car reaches the destination floor and the doors open to leave, and will keep track of how long the passenger had to wait after pressing the button and how long the passenger was in the elevator for. The program then displays average and maximum waiting times, as well as histograms of the population.

The scheduling tester is highly configurable via command-line arguments, and 4 different configurations are run on Gradescope automatically (their arguments are shown in each respective test). The command-line arguments supported include the following:

- `--car-delay {number}` - This argument is the delay argument passed to the car component
- `--cars {number}` - This is the number of elevator cars to simulate. An instance of the car component will be started for each of these
- `--num-passengers {number}` - This is the number of passengers to simulate

- `--lowest-floor {floor}` - This is the lowest floor in the simulation - each elevator car will be started with this value as the lowest floor and each passengers will randomly appear on a floor between this and the highest floor, and have a random floor between this and the highest floor set as that passenger's destination
- `--highest-floor {floor}` - Same as above, but for the highest floor
- `--sim-start {number}` - This value is in milliseconds. Passengers will randomly appear at times between this value and the `--sim-end` value.
- `--sim-end {number}` - This value is in milliseconds. Passengers will randomly appear at times between this value and the `--sim-start` value.
- `--histogram-len {number}` - This indicates the number of bins the histograms displayed have.
- `--svg {filename}` - This argument produces an animated SVG file of the simulation, which is saved to the given filename. Another argument, `--svg-timescale {factor}` indicates how time should be scaled (multiplied) in the SVG compared to the simulation - useful for running a quick simulation where most of the times are in milliseconds, but slowing it down so you can see what's actually happening. SVG is a vector graphics format that supports animation and can be played in most web browsers.

The SVG output mode can be useful for providing a visual demonstration, similarly to the 'display-cars' tool, for trying to determine how your scheduler behaves in various scenarios:

