Assessment 2 component: Simulator

Summary

The overseer and other components are designed so that they can be plugged into a system as long as it provides the shared memory interface to the building's mechanisms.

For testing purposes, however, since we don't have an actual building to test on, we need to provide a simulator. The simulator will pretend to be a collection of card readers, doors, elevators, cameras, temperature sensors etc. and communicate with your components via that shared memory.

Program name

```
simulator
```

Command-line arguments

```
{scenario file}
```

Shared memory structure

The simulator uses many shared memory structures - see the pages for each component to see its shared memory structure.

Simulator operation

The simulator reads a **scenario file** to determine how to play out the simulation. Different scenario files can be used for testing different combinations of components and different situations.

The scenario file is divided into two sections- the **init** section and the **scenario** section. Every line of the init section begins with the text INIT. Then, after the init section, there will be a line consisting just of the text SCENARIO, following which will be the scenario section.

The init section is where all the configurations for all of the components are provided. For example, if this scenario involves two card readers, one door, one fire alarm callpoint, the fire alarm unit and the overseer, the init section will consist of 6 lines to set all of that up. After reading in the init section, the simulator will prepare the shared memory region and then launch each of the processes that make up this simulation.

In the scenario section each line starts with a timestamp (in microseconds) and consists of an event - for example, an access card being read by a card reader, a fire alarm callpoint being pressed, a temperature sensor registering a particular temperature etc.

The simulator will wait for 1 second after starting up all processes before it starts simulating events, and it will wait for 1 second after the last event before terminating all of the processes and then closing.

Because the simulator needs to simulate the ordinary function of many devices at once, it is highly recommended that the simulator be implemented with multithreading.

Scenario file init section

The init section consists of some number of lines beginning with INIT. These are used to pass command-line arguments to the individual programs, but not all of them - the ip addresses, ports and shared memory paths and offsets are all determined by the simulator and supplied automatically.

- INIT overseer {door open duration (in microseconds)} {datagram resend delay (in microseconds)} {authorisation file} {connections file} {layout file}
- INIT firealarm {temperature threshold} {min detections} {detection period (in microseconds)} {reserved}
- INIT cardreader {id} {wait time (in microseconds)}
- INIT door {id} {FAIL_SAFE | FAIL_SECURE} {door open/close time (microseconds)}
- INIT callpoint {resend delay (in microseconds)}
- INIT tempsensor {id} {max condvar wait (microseconds)} {max update wait (microseconds)} {receiver list...}
- INIT elevator {id} {wait time (in microseconds)} {door stay open time (in microseconds)} {door open/close time (microseconds)} {travel time (microseconds)} {starting floor}
- INIT destselect {id} {wait time (in microseconds)}

- INIT camera {id} {temperature change threshold} {min angle} {max angle} {frame delay (microseconds)}

The underlined arguments are ones that get passed directly to the respective program through **execl() (https://linux.die.net/man/3/execl)** . The non-underlined arguments are special and are used for something else in the simulator.

The arguments that are missing from the INIT declarations but still need to be passed to programs are ones that the simulator determines. These include:

- {address:port} arguments for the overseer, fire alarm unit, door, temperature sensor, camera and elevator. The address used will always be 127.0.0.1 (as everything will be run locally.) The ports will be assigned starting at 3000 and increasing - for example, if your scenario has an overseer, fire alarm unit and three doors, you might assign them 127.0.0.1:3000 to 127.0.0.1:3004.
- {overseer address:port} arguments for the fire alarm unit, card reader, door, elevator, destination select and camera. Just pass these the address you gave the overseer.
- {fire alarm unit address:port} argument for the fire alarm manual call-point - similarly, pass it the address that was assigned to the fire alarm unit.
- {shared memory path} and {shared memory offset} - the simulator will have to create an appropriate shared memory segment and give each component an appropriately sized chunk of it. You can also create multiple shared memory segments if you prefer, but that might be more work to manage.

The receiver list for tempsensor is different, however- it consists of a series of short codes: F, O and S#. F corresponds to the fire alarm unit, O to the overseer and S# to another temperature sensor. S0 is the first temperature sensor that appears in the INIT second, S1 is the second, S2 is the third and so on. So, for example, if the receiver list is for the first temperature sensor is O S1 S2 that means this sensor is connected to the overseer and the second and third temperature sensors. The simulator will replace these with the appropriate address:port pairs when invoking the programs.

Because some of these are forward references (a temp sensor can refer to another temp sensor that has not been defined yet) you might want to take 2 passes over the INIT section - one to allocate ports to everything, and a second pass to determine the other arguments and launch all the processes.

Shared memory

The simulator is in charge of creating the shared memory space (with appropriate permissions), creating all of the mutexes and condition variables, setting all of the defaults and then giving each process the path to the file and the offset into that file.

The exact process of how you do this is up to you, but feel free to take advantage of the following limits (these limits only apply to the simulator, not the assignment as a whole):

- Exactly 1 overseer
- Maximum of 1 fire alarm unit
- Maximum of 40 card readers
- Maximum of 20 doors
- Maximum of 20 temperature sensors
- Maximum of 20 fire alarm call-points
- Maximum of 20 cameras
- Maximum of 20 destination select panels
- Maximum of 10 elevators

Given these limits, you could simply create a big struct with arrays of the maximum number of the individual shared memory structs and make that your shared memory segment. You might find **offsetof() (https://man7.org/linux/man-pages/man3/offsetof.3.html)** useful for finding the offset values to pass to each of the programs. (The struct should be 54592 bytes in total, though it might depend on the order of the elements in the struct.)

All of the shared memory structs include at least one pthread mutex and condition variable, sometimes multiple. The simulator will need to init all of these, and they need to be configured as **process shared** otherwise they wil not work (hint: use **pthread_mutexattr_setpshared() (https://pubs.opengroup.org/onlinepubs/7908799/xsh/pthread_mutexattr_setpshared.html)** and **pthread_condattr_setpshared() (https://pubs.opengroup.org/onlinepubs/7908799/xsh/pthread_condattr_setpshared.html)**)

The remaining part of this section will go over each shared struct and discuss how to default-initialise the values in each one:

Overseer struct

Initialise 'security_alarm' to `-`

Fire alarm unit struct

Initialise 'alarm' to `-`

Card reader controller struct

Set the entire 'scanned' array to `\0` (NUL byte), and do the same for the 'response' char

Door controller struct

Set 'status' to `C`

Fire alarm call-point controller struct

Initialise 'status' to `-`

Temperature sensor controller struct

Initialise 'temperature' to 22.0f

Elevator controller struct

Initialise 'status' to C, 'direction' to `-` and 'floor' to whatever {starting floor} value was specified in INIT

Destination select controller struct

Set the entire 'scanned' array and the 'response' char to `\0` (NUL byte), and set 'floor_select' to 0

Camera controller struct

Set 'min_angle' and 'max_angle' to the {min angle} and {max angle} values specified in INIT, set 'current_angle' to {min angle}, set 'status' to O. For the video array, see the 'Generating artificial camera frames' section below.

Spawning processes

After reading the INIT section and creating the shared memory structure, the simulator then needs to launch the other processes. The fork() and execl() functions will work nicely, but one thing to note is that the overseer process needs to be in the foreground so it can read standard input and allow you to type in commands. In addition, the overseer process needs to start first (as other components initialise with it). So the process for having the simulator launch these processes is as follows:

- Issue fork()
- The **parent process** should be the one to call execl to launch the overseer
- The **child process** should then wait (250 milliseconds should be enough), then call fork() and exec() to launch each of the other processes. The child process then continues as the simulator.

The result is that the original process is replaced by the overseer and the simulator lives on as a spawned child process, along with the other components.

Also note that you keep track of all of the process' PIDs so they can be terminated later when the simulation ends.

Scenario section

The scenario section begins with a line consisting of just the text SCENARIO. This serves as a delimiter, so the simulator knows the init section is over. The remaining lines of the file consist of events from the following selection:

- `{timestamp} CARD_SCAN {num} {code}`
- `{timestamp} CALLPOINT_TRIGGER {num}`
- `{timestamp} TEMP_CHANGE {num} {new temperature}`
- `{timestamp} DEST_SELECT {num} {code} {floor}`
- `{timestamp} CAMERA_MOTION {num}`

Each event begins with a timestamp, which is a number in microseconds, and indicates that this event will happen that many microseconds after the simulation starts.

Note that these events use {num} to refer to the particular card reader, call-point, temperature sensor etc. This refers to its location in the init section, relative to other components of the same type. For example, CARD_SCAN with a {num} of 0 refers to the first card reader specified in init, while 1 refers to the second card reader and so forth.

```
{timestamp} CARD_SCAN {num} {code}
```

This will cause a card reader to scan the ID card with the code {code} at {timestamp}.

```
{timestamp} CALLPOINT_TRIGGER {num}
```

This will cause a fire alarm manual call-point to trigger (causing the fire alarm to go off, assuming that system is working).

```
{timestamp} TEMP_CHANGE {num} {new temperature}
```

This will cause the specified temperature sensor to register a new temperature of {new temperature} (which is a floating-point number).

```
{timestamp} DEST_SELECT {num} {code} {floor}
```

This will cause the specified destination select panel to scan the ID card with {code} and to register a press of the button for floor {floor}.

```
{timestamp} CAMERA_MOTION {num}
```

This will cause the specified camera to, on its next frame, capture *something*. See the 'Generating artificial camera frames' section below for more details.

One second after the last event occurs, the simulator will repeatedly invoke the **kill() (https://man7.org/linux/man-pages/man2/kill.2.html)** system call to terminate all the other processes, then exit.

Simulation: Card readers

In the simulation, card readers only do anything when CARD_SCAN events occur. In a CARD_SCAN event, the mutex will be locked, the code written to the 'scanned' array and then the mutex is unlocked and the 'scanned_cond' condition variable is signalled.

Simulation: Doors

A door will start out by locking its mutex, then waiting on cond_start. Upon waking up, if status is `O` or `C`, it will wait on cond_start again. On the other hand, if status is `o` or `c`, it will sleep for {door open/close time} microseconds, change the status (to `O` if the status was `o` and `C` if the status was `c`), signal cond_end and wait on cond_start again.

Simulation: Call-points

A call-point does nothing until an event says it has been triggered - then it will lock the mutex, set status to `*`, unlock the mutex and signal the condition variable.

Simulation: Temperature sensor

A temperature sensor does nothing until an event says the temperature has changed - then it will lock the mutex, set temperature to whatever the temperature has changed to, unlock the mutex and signal the condition variable.

Simulation: Elevator

An elevator starts out by locking its mutex and waiting on cond_elevator. Upon waking up, it will check its current status

If its status is `o`, it will sleep for {door open/close time} microseconds, then set status to `O`, signal cond_controller, then wait on cond_elevator again, repeating the loop.

If its status is `C`, skip to the 'Otherwise' sentence below and check for direction.

If its status is `c`, it will sleep for {door open/close time} microseconds. Then, if direction is `-` it will set status to `C`, signal cond_controller, then wait on cond_elevator again, repeating the loop.

Otherwise (so either status is `C` and we got here from the 'skip to' step, or status is `c` and we got here because direction is not `-`), if direction is `U` or `D` it will set status to `M`, sleep for {travel time} microseconds, increase (if direction is `U`) or decrease (if direction is `D`) floor by 1, set direction to `-` and set status to `C`, signal cond_controller and wait on cond_elevator again, repeating the loop.

Simulation: Destination select panel

Similar to card readers, destination select panels only do anything when DEST_SELECT events occur. In a DEST_SELECT event, the mutex will be locked, the code written to the 'scanned' array and the floor written to the 'floor_select' value, then the mutex is unlocked and the 'scanned_cond' condition variable is signalled.

Simulation: Camera

The simulated camera will perform the following loop:

1. Sleep for {frame delay} microseconds
2. Lock the mutex
3. If status is `L` and current_angle != min_angle, decrease current_angle by 1 (if this would go below 0, wrap around to 359)
4. If status is `R` and current_angle != max_angle, increase current_angle by 1 (if this would go above 359, wrap around to 0)
5. If status is not `-`, generate a frame of video (see the section below for tips).
6. If status is `-`, fill the 'video' array with the value 0.
7. Unlock the mutex
8. Signal the condition variable
9. Loop back to 1

Generating artificial camera frames

You could do something very fancy here with generating fake camera frames for a simulation, with scrolling images, simulated camera grain etc. and if you're motivated to do that, go ahead - but ultimately what you need to test this out is just to have the camera image very dark most of the time, and then suddenly turn very bright when motion is detected. This means that you could produce an adequate camera simulation suitable for testing by just filling the array with 0s unless there's motion, then filling the array with 255s for the frame after the CAMERA_MOTION event.