Project Specification

Kernel Module Programming
System Calls, Kernel Module, and Elevator Scheduling

Purpose

This project introduces you to the nuts and bolts of system calls, kernel programming, concurrency, and synchronization in the kernel. It is divided into three parts.

Part 1: System-call Tracing

Write an emtpy C program, `empty.c` Then, create a copy of this program called `part1.c` and add exactly seven system calls to the program. The
system calls available to your machine can be found within `/usr/include/unistd.h` . Further, you can use the command line tool, `strace`, to intercept and record the system calls called by a process.

To confirm you have added the correct number of system calls, execute the following commands:

```
$ gcc -o empty.x empty.c
$ strace -o empty.trace ./empty.x

$ gcc -o part1.x part1.c
$ strace -o part1.trace ./part1.x
```

To reduce the length of the output from `strace`, try to minimize the use of other function calls (e.g. `stdlib.h`) in your program.

Note: Using `strace` on an empty C program will produce a number of system calls, so when using `strace` on your part1 code, it should produce 7 more system calls than that.

, .

Part 2: Kernel Module
In Unix-like operating systems, time is sometimes specified to be the seconds since the Unix Epoch (January 1st, 1970).

You will create a kernel module called `my_timer` that calls `current_kernel_time()` and stores the time value. `current_kernel_time()` holds the number of seconds and nanoseconds since the Epoch.

When `my_timer` is loaded (using `insmod`), it should create a proc entry called `/proc/timer`.

When `my_timer` is unloaded (using `rmmod`), `/proc/timer` should be removed.

On each read you will use the proc interface to both print the current time as well as the amount of time that's passed since the last call (if valid).

Example usage:

```
$ cat /proc/timer
current time: 1518647111.760933999

$ sleep 1
$ cat /proc/timer
current time: 1518647112.768429998
elapsed time: 1.007495999

$ sleep 3
$ cat /proc/timer
current time: 1518647115.774925999
elapsed time: 3.006496001

$ sleep 5
$ cat /proc/timer
current time: 1518647120.780421999
elapsed time: 5.005496000
```

Part 3: Elevator Scheduler: Humans v. Zombies
Your task is to implement a scheduling algorithm for an elevator, Halloween edition. The elevator can only hold 10 passengers at a time. Each passenger is either a human or zombie (randomly chosen, equally likely). Your elevator must track the number of humans and zombies onboard the elevator. Passengers will appear on a floor of their choosing and always know where they wish to go. For optimization purposes, you can assume most passengers not starting on the first floor are going to the lobby (first floor). Passengers board the elevator in FIFO order. Humans will only board the elevator if no zombies are present. Zombies can always board the elevator. If a zombie boards an elevator which contains humans, all humans onboard the elevator become zombies. Once someone boards the elevator, they may only get off when the elevator arrives at the destination. Passengers will wait on floors to be serviced indefinitely.

Step 1: Kernel Module with an Elevator
Develop a representation of an elevator. In this project, you will be required to support having a maximum load of 10 passengers (this limit can never be exceeded by the elevator). The elevator must wait for 2.0 seconds when moving between floors, and it must wait for 1.0 seconds while loading/unloading passengers. The building has floor 1 as the minimum floor number (lobby) and floor 10 being the maximum floor number. New passengers can arrive at any time and each floor needs to support an arbitrary number of them.

Step 2: Add System Calls
Once you have a kernel module, you must modify the kernel by adding three system calls. These calls will be used by a user-space application to control your elevator and create passengers. You need to assign the system calls the following numbers:

- 335 for `start_elevator()`
- 336 for `issue_request()`
- 337 for `stop_elevator()`

`int start_elevator(void)`
Description: Activates the elevator for service. From that point onward, the elevator exists and will begin to service requests. This system call will return 1 if the elevator is already active, 0 for a successful elevator start, and `-ERRORNUM` if it could not initialize (e.g. `-ENOMEM` if it couldn't allocate memory). Initialize an elevator as follows:

- State: `IDLE`
- Current floor: 1
- Current load: 0 passengers

`int issue_request(int start_floor, int destination_floor, int type)`
Description: Creates a request for a passenger at `start_floor` that wishes to go to `destination_floor`. `type` is an indicator variable where 0 represents a human and 1 represents a zombie. This function returns 1 if the request is not valid (one of the variables is out of range or invalid type), and 0 otherwise.

`int stop_elevator(void)`
Description: Deactivates the elevator. At this point, the elevator will process no more new requests (that is, passengers waiting on floors). However, before an elevator completely stops, it must offload all of its current passengers. Only after the elevator is empty may it be deactivated (state = `OFFLINE`).

This function returns 1 if the elevator is already in the process of deactivating, and 0 otherwise.

Step 3: /Proc

The module must provide a proc entry named `/proc/elevator`. Here, you will need to print the following (each labeled appropriately):

- The elevator's movement state:
  - `OFFLINE`: when the module is installed but the elevator isn't running (initial state)
  - `IDLE`: elevator is stopped on a floor because there are no more passengers to service
  - `LOADING`: elevator is stopped on a floor to load and unload passengers
  - `UP`: elevator is moving from a lower floor to a higher floor
  - `DOWN`: elevator is moving from a higher floor to a lower floor
- The status of the elevator: infected or not
- The current floor the elevator is on
- The elevator's current load (passenger count)
- The total number of passengers waiting
- The number of passengers serviced

You will also need to print the following for each floor of the building:

- An indicator for whether or not the elevator is on the floor
- The count of the waiting passengers
- For each waiting passenger, a character indicating the passenger type

```
sample_proc.txt:
Elevator state: UP
Elevator status: Infected
Current floor: 4
Number of passengers: 6
Number of passengers waiting: 10
Number passengers serviced: 61


[ ] Floor 10:   3   | | X
[ ] Floor  9:   0
[ ] Floor  8:   2   X X
[ ] Floor  7:   0
[ ] Floor  6:   1   X
[ ] Floor  5:   0
[*] Floor  4:   2   | X
[ ] Floor  3:   2   X |
[ ] Floor  2:   0
[ ] Floor  1:   0

( "|" for human, "X" for zombie )
```

Step 4: Test
Once you've implemented your system calls, you must interact with two provided user-space applications that will allow communication with your kernel module.

`producer.c`: This program will issue N random requests, specified by input.
`consumer.c`: This program expects one flag:

- If the flag is `--start`, then the program must start the elevator
- If the flag is `--stop`, then the program must stop the elevator

`producer.c` and `consumer.c` will be provided to you.

Implementation Requirements

- The list of passengers waiting at each floor and the list of passengers on the elevator must be stored in a kernel linked list (linux/list.h).
- The passengers must be allocated dynamically using `kmalloc`.
- The elevator activity must be controlled within a `kthread`.
- The elevator must use locking around shared data.