

Project 2 Specification

Kernel Module Programming

System Calls, Kernel Module, and Elevator Scheduling

Code Due: Wednesday, March 4 11:59PM

Project Demos: March 5-6

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 a program that uses exactly five system calls. You will not receive points if the program contains more or fewer than five. 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.

Once you've written this program, execute the following commands:

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

Look at the log file to figure out how many system calls your program is calling. 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 code, it should produce 5 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/timed`.

When `my_timer` is unloaded (using `rmmmod`), `/proc/timed` 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/timed
current time: 1518647111.760933999
```

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

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

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

Part 3: Elevator Scheduler

Your task is to implement a scheduling algorithm for a pet-friendly hotel elevator. Your elevator must track the number of passengers and the total weight onboard the elevator. There are three types of passengers:

- A person counts as 3 weight units
- A dog counts as 2 weight units
- A cat counts as 1 weight unit

People will appear on a floor of their choosing and always know where they wish to go. Each person may have 0-3 pets (randomly chosen). A person will never have both a cat and a dog (species is randomly chosen). All dogs and cats must be accompanied by an owner (person). If a person does have pets, either the person and all pets board the elevator, or no one from the group boards the elevator. 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. To avoid potential scuffles, a dog cannot enter an elevator containing a cat and vice versa. If there is space and there is not a conflict of species, the elevator must accept the person and their pets unless the elevator is moving in the opposite direction from where they wish to go. Once they board 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 15 weight units (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, 0 weight units

```
int issue_request(int num_pets, int pet_type, int start_floor, int destination_floor)
```

Description: Creates a request for a person with `num_pets` of type `pet_type` at `start_floor` that wishes to go to `destination_floor`. This function returns 1 if the request is not valid (one of the variables is out of range), and 0 otherwise. `pet_type` is 1 for cat and 2 for dog.

```
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 type of animals on the elevator, if any
- The current floor the elevator is on
- The elevator's current load (in terms of both passengers units and weight units)
- 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 animals: dog

Current floor: 4

Number of passengers: 6

Current weight: 14

Number of passengers waiting: 34

Number passengers serviced: 61

```
[ ] Floor 10: 11 | o o o | x x x | o o
[ ] Floor  9:  0
[ ] Floor  8:  5 | o | x x
[ ] Floor  7:  0
[ ] Floor  6:  0
[ ] Floor  5:  0
[*] Floor  4:  8 | x x x | x x x
[ ] Floor  3: 10 | x | x x x | x x x
[ ] Floor  2:  0
[ ] Floor  1:  0
```

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 linked list.
- The passengers must be allocated dynamically using `kmalloc`.
- The elevator activity must be controlled within a `kthread`.

Extra Credit

The top three elevator schedulers will receive +5 points to their project 2 grade. The metric to optimize is the total number of passengers serviced within a span of time. The next three schedulers will receive +2 points to their project 2 grade.

Project Submission Procedure

The following files must be tar'd and submitted on canvas (create a folder for each part and place the files in the corresponding folders):

- README
- Part 1:
 - `empty.c`
 - `part1.c`
 - `empty.trace`
 - `part1.trace`
- Part 2:
 - `my_timer.c`
 - Makefile
- Part 3:
 - `elevator.c`
 - Makefile

****** If you split your code into multiple header/c files, those must also be included in your tar file.

You will be required to schedule for a project demonstration. A week before the project is due, registration for the demonstration will open. You will be given 15 minutes to present your project. You will be required to demonstrate:

- Project source files
 - System calls, module sources, user-space sources, Makefiles
- Successful run of Part 1
- Successful installation of Part 2
- Properly display `my_timer /proc/timed`
- Successful remove of Part 2
- Successful installation of Part 3
- Execution of an arbitrary number of consumers and producers for 5 minutes
- Information stored in `/proc/elevator` once per second
- Successful removal of Part 3

Any demonstration failing to install a portion of the project successfully during their allotted time will receive a 0 for that portion of the project. Be absolutely sure that you can at least install and remove all kernel modules before attempting to demonstrate. The grader, may also choose to question you on project components. You should be able to demonstrate an understanding of the project implementation.

Contact person for the project: Leah Rumancik, ler13f@my.fsu.edu

Office Hours: MCH202, Tuesday 4PM-5PM, Wednesday 4PM-5PM, and gladly by appointment