Project 2 Specification
Kernel Module Programming
System Calls, Kernel Module, and Elevator Scheduling

Assigned: September 25th, 2015, 12:20pm
Due: October 26th, 2015, 11:59pm

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.

Restrictions:
C programming language
Linux kernel version 4.2

Part 1: System-call Tracing
Write a program that uses exactly ten system calls. You will not receive points if the program contains more or fewer than ten. 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, Notice that the outputs may differ if you run `strace` more than once on the same program. 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.

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_xtime` that calls `current_kernel_time()` and stores the value. `current_kernel_time()` holds the number of seconds since the Epoch and the number of nanoseconds since the last second.

When `my_xtime` is loaded (using `insmod`), it should create a `/proc` entry called `/proc/timed`. When `my_xtime` is unloaded (using `rmmod`), `/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: 1234567.000111222
$ sleep 5
```

```
$ cat /proc/timed
current time: 1234572.456123987
elapsed time: 5.456012765
$ sleep 3
$ cat /proc/timed
current time: 1234575.544212005
elapsed time: 3.088088018
```

Part 3: Elevator Scheduler

Your task is to implement a scheduling algorithm for a hotel elevator.

Your elevator must track the number of passengers and the total weight. Elevator load consists of five types of people: adults, children, bellhops, and room service:

- An adult counts as 1 passenger and 1 weight unit
- A child counts as 1 passenger and ½ weight unit
- A bellhop counts as 2 passengers and 2 weight unit
- Room service counts as 1 passenger and 2 weight units

Passengers randomly appear on a floor of their choosing. They always know where they wish to go. Most of the time, when a passenger is on a floor other than the first, they will choose to go to the first floor. A passenger must always board the elevator if it can accept them in FIFO order, unless it is moving in the opposite direction. Once they board the elevator, they may only get off when the elevator arrives at the destination. Passengers wait 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 8 weight units or 8 passenger units. The elevator must wait for 2 seconds when moving between floors, and it must wait for 1 second while loading/unloading passengers. The building has floor 1 being the minimum floor number 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: /Proc

The module must provide a `/proc` entry named `/proc/elevator`. Here, you will need to print:

- The elevator's movement state (IDLE, UP, DOWN, LOADING, STOPPED)
- The current floor the elevator is on
- The next floor the elevator intends to service
- The elevator's current load (passengers units and weight units)

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

- The load of the waiting passengers
- The total number of passengers that have been serviced

Step 3: 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: 323 for start_elevator(), 324 for issue_request(), and

325 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, and 0 for a successful elevator start. Initialize an elevator as follows:

- Direction: IDLE
- Current load: 0 passengers, 0 weight units
- Current floor: 1

```
int issue_request(int passenger_type,
                  int start_floor,
                  int destination_floor)
```

Description: Creates a passenger of type `passenger_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. A passenger type can be translated to an `int` as follows:

- Adult = 0
- Child = 1
- Bellhop = 2
- Room service = 3

`int stop_elevator(void)`
Description: Deactivates the elevator. At this point, this elevator will process no more requests. However, before an elevator ceases to exists, it must offload all of its current passengers. Only after the elevator is empty may it be deactivated. This function returns 1 if the elevator is already in the process of deactivating, and 0 otherwise.

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 <--start | --stop>
This program expects one flag and on argument:

- 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.

Extra Credit

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

Project Submission Procedure

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 20 minutes to present your project. This demonstration will take place on the lab machine assigned to you. You will be required to demonstrate:

- Project source files
  - System calls, module sources, user-space sources
- Project makefiles
- Successful run of Part 1
- Successful installation of Part 2
- Properly display xtime `/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. As before, any project that fails to compile will also receive a 0 for that portion of the project.

The grader, may also choose to question you on project components. You should be able to demonstrate an understanding of the project implementation.

Finally, you must submit to blackboard a tar archive of your project with the source code and make file for each part in a separate directory (e.g. part1/Makefile), and with the README and project report at the root of the archive. Each team only needs to submit once.