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

Assigned: September 15th, 2009, 11:00am
Due: October 18th, 11:59:59pm

Language Restrictions: C only

Purpose

This project introduces you to the nuts and bolts of system calls, kernel programming, and concurrency and synchronization in the kernel. This project 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 your program contains more or fewer than five system calls. The system calls available to your machine may be found in the file `/usr/include/unistd.h`. The command line tool, `strace`, will be valuable to you since it intercepts and records the system calls called by a process.

Once you've written this program, execute the following commands, where `<program_name>` should be substituted for the name of the executable for your program:

```
$> make
$> strace -o log1 <program_name>
```

Look at the output 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. Think about why this may be. To reduce the length of the output from `strace` into the log files, try to minimize the use of other function calls (like C standard library calls) in your program.

Part 2: `xtime` Kernel Module

In Unix-like operating systems, time is sometimes specified by the seconds or nanoseconds since the Epoch, which happens to be January 1st, 1970. You will create a kernel module called `my_xtime` that reads the kernel variable `xtime` and displays the value of `xtime` to the user through the `/proc` interface. When `my_xtime` is loaded (using the command `insmod`), it should create a `/proc` entry called `/proc/currenttime`. When a user reads `/proc/currenttime`, the user should see the time since Epoch in seconds and in nanoseconds, delimited by a single space:

```
When the my_xtime module is unloaded through the command `rmmod`, `/proc/currenttime` should be removed.

Example usage:
$> cat /proc/currenttime
1234567 22334231
$> echo 1 > /proc/currenttime
```
bash: /proc/currenttime: Permission denied

Additionally, you are required to write a user-space program, which outputs the value of the function gettimeofday() as two integers delimited by a space. It should also output the values exposed in /proc/currenttime on a separate line as two integers.

Example output:
$> ./xtime_test
gmtimeofday: 1234566 22334230
xtime: 1234567 22334231

Part 3: Elevator Scheduling

Your task is to implement an elevator scheduling algorithm. An elevator is defined as a device that may only move up and down. An elevator is stopped if there are no requests to process or if it is currently loading passengers. It has a starting floor, a current load, and a current floor. A maximum load and a maximum floor will be given. An elevator also takes a finite amount of time to move between floors and to collect passengers from a floor—this does not happen instantaneously.

Your elevator must keep track of the number of passengers and weight (units). Elevator load consists of four types of people: adults, children, delivery people, and elevator maintenance people. An adult counts as one passenger and one weight unit, a child counts as one passenger and ½ weight unit, a delivery person counts as one passenger and two weight units (delivery person + their load), and the elevator maintenance person counts as one passenger and one unit. Passengers randomly appear on a floor of their choosing. They always have in mind the floor they wish to go to. A passenger must always board the elevator if it can accept them in FIFO order, even if it is moving in the opposite direction. Once they board the elevator, they may only get off when the elevator arrives at the floor they initially requested—a passenger may not depart the elevator earlier than this. Once a passenger arrives at their destination floor, they cease to exist. Passengers wait infinitely, until the elevator arrives to collect them—they never get bored and choose to use the stairs.

Task Specification

This is a classic exercise in modeling consumers and producers. The producer produces passengers and the consumer is the elevator. There are many pieces needed to provide a complete implementation discussed below.

Step 1: Kernel Module with an Elevator
Develop a representation of an elevator. In this project, you will be required to support one having a maximum load of 6 weight units or 6 passengers. The elevator must wait for 2 seconds when moving from one floor to the next, and the elevator must wait for 1 second while loading passengers at a floor. A building has a maximum of 5 floors. No passengers may issue requests at the same time, but multiple passengers may wait on the same floor.

In addition, the loading of an elevator maintenance person invokes a special case. You may assume there is only one maintenance person in the building. If the maintenance person is being loaded, the elevator must pause for an additional 3 seconds while loading from that specific floor so that “maintenance” may be performed on the elevator. After the maintenance person is loaded, the elevator
will treat them like regular adults until they leave.

Step 2: /Proc

The module must provide a /proc entry named /proc/elevator. For your elevator, print:
- The elevator's direction of movement (UP, DOWN, LOADING, STOPPED)
- The current floor the elevator is on
- The next floor the elevator intends to service
- The number of passengers and weight units the elevator is carrying

For each floor of the building, print
- The number of passengers and weight units waiting on that floor
- The total number of passengers that have been serviced, where serviced is defined as the number of passengers that have departed from the elevator, or ceased to exist

Step 3: Add System Calls

Once you have a complete kernel module, you must modify the kernel by adding three system calls. These system calls will be used by a user-space application to control your elevators and create passengers.

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: STOPPED
- Current number of passengers and weight units: 0
- Current floor: 0

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. A passenger type can be translated to an int as follows:
- Adult = 0
- Child = 1
- Delivery person = 2
- Elevator maintenance person = 3
This function returns 1 if the request is not valid (some variable is out of range), and 0 otherwise.

int stop_elevator(void)

Description: Deactivates the elevator. At this point, this elevator will process no more requests. However, before an elevator ceases to exist, it must offload all 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 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 one 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 submissions as measured by the above evaluation procedure will receive +5 points to their project 3 grade. The metric to optimize is: total passengers serviced.

Project Submission Procedure

You will be required to schedule for a project demonstration. A week before the project is due, registration for demonstration will open. You will be given 20 minutes to present your project. This demonstration will take place by the computer you used to implement your project. 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
- Information stored in /proc/currenttime
- Successful removal of part2
- Successful installation of part 3
- Execution of an arbitrary number of consumers and producers for 5 minutes
- Information stored in /proc/elevator once per minute
- 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, at their discretion, may choose to question you on project components. You should be able to demonstrate an understanding of the project implementation.