Lab Assignment 2: Synchronization in Linux Kernel

Policies

- The assignment must be implemented in C. Inline assembly code is accepted.
- You should work on this assignment individually.

Notes

- You must use a virtual machine to implement this lab or you might damage your hardware (PS/2 keyboard and mouse).
- Don't try it on real hardware unless you are highly confident that your code works correctly.
- Don't flood your PS/2 controller with commands and data in a brute-force manner.

Objectives

- To get familiar with Linux kernel and kernel module development.
- To learn how the OS interfaces hardware and peripherals.
- To understand the difference between kernel-mode and user-mode.
- To learn how to use Linux kernel synchronization facilities.

Overview

You are required to write a Linux kernel module that directly controls the LEDs of your PS/2 keyboard (Num Lock, Caps Lock, and Scroll Lock).

Two basic routines must be provided by the module: `get_led_state` and `set_led_state`. User programs must be able to access these routines. This can be done through sysfs, device files, or system calls.

Next, you will run multiple user programs in parallel and illustrate why this would lead to race conditions if the programs are concurrently accessing the LEDs through your module. Finally, you will use kernel synchronization mechanisms (semaphores) to add atomicity to your module.

PART I: Kernel Module Specification

You are required to create a kernel module for Linux kernel. Refer to Appendix I for information on Linux kernel module development. The module must implement the following routines:

- `void set_led_state(int led, int state):`

  When this function is called, one of the three LEDs of PS/2 keyboard is turned on or off according to the values of `led` and `state` parameters. `led` parameter is the index of the LED to operate on. Table 1 shows the possible values for `led`. `state` should be 1 if it is required to turn on the LED. Otherwise, `state` must be 0.

- `int get_led_state(int led):`

  This function returns the state of the LED indexed by `led`. If the LED is currently turned on, the function returns 1. Otherwise, 0 is returned.

Index	Corresponding LED
0	Scroll Lock
1	Num Lock
2	Caps Lock

Table 1: LED Indices

In order to control the LEDs of PS/2 keyboard, the kernel module needs to interface 8042 chip (PS/2 keyboard controller) to send appropriate commands/data to the keyboard. Refer to Appendix II for information on PS/2 protocol.

PART II: Interfacing User-Space Programs

Next step is to expose `get_led_state()` and `set_led_state()` to user space programs. This can be done in several ways:

- Sysfs:

  When the module is loaded to kernel, it creates 3 files under `/sys` representing the 3 LEDs. User program can read the files to query the status of the LEDs, or write the files to update the status of the LEDs.

  Linux kernel provides a simple interface for modules to make use of sysfs functionality. When a sysfs file is read or written by the user, a routine in the module code is called to handle data.

- Devfs:

  When the module is loaded to kernel, it creates a file under `/dev` representing the module. User programs simply open the device file then use `ioctl()` system call on the device file to execute the routines provided by the module.

- System calls:

  When the module is loaded to kernel, it modifies Linux system call entry table to assign a system call entry for the module. User programs use this system call to execute the functionality provided by the module. You will need to modify and recompile kernel source code in order to export `sys_call_table` symbol.

You are required to implement **only one** of the three techniques presented above.

After implementing the user-space interface in your module, you will then create a user-space program that communicates with your kernel module to set and get LED status. The user-space program must be called from shell as a single command, called `leds`, with the following arguments:

- The operation to be performed on the LEDs (`set` or `get`).
- The LED to operate on (`num`, `caps`, or `scroll`).
- The new state of the LED (`on` or `off`) in case the operation (first argument) is `set`.

If the operation to be performed (first argument) is `set`, third argument indicates whether the LED will be turned on or off. If the operation is `get`, the third argument is omitted, and the command prints current state of the LED (`on` or `off`).

For example, to turn on Caps Lock LED, the command is executed from shell with the following arguments:

```
[iocoder@inspiron lab2]$ ./leds set caps on
```

To get the status of Caps Lock:

```
[iocoder@inspiron lab2]$ ./leds get caps
on
```

Now it is easy to write shell scripts to test your work. The following script toggles the state of Caps Lock every 500 milliseconds:

```
#!/bin/bash

while true; do
    ./leds set caps on;
    sleep 0.5;
    ./leds set caps off;
    sleep 0.5;
done
```

Listing 1: toggle.sh

For more information about user-space interfacing, check the following links:

- `http://mirrors.neusoft.edu.cn/rpi-kernel/samples/kobject/`
- `http://www.tldp.org/LDP/lkmpg/2.6/html/x892.html`
- `http://www.tldp.org/LDP/lkmpg/2.6/html/x978.html`

PART III: Race Conditions

In this part, you are required to launch multiple user programs that access the LEDs in parallel. Adjust the timings of the processes such that race condition occurs. The module should log low-level hardware activity (using `printk()`). You should use your kernel logs to explain why race conditions will happen in this case. Refer to Appendix I for information about using `printk()` and `dmesg`.

As a simple example, a race condition might occur if intervened low-level I/O accesses from different processes occur at the same time. For example, if process 1 is trying to turn on Caps Lock, while process 2 is trying to turn off Num Lock. Both processes will call `set_led_state()` routine implemented in Part I through the user-space interface implemented in Part II.

The problem happens if the two processes call `set_led_state()` at the same time. `set_led_state()` actually performs a series of I/O operations to mutate the state of the LEDs (refer to Appendix II):

1. Send 'Set LEDs' command to PS/2 keyboard.
2. Wait for `ACK` from keyboard.
3. Send LED status.
4. Wait for another `ACK` from keyboard.

Assume the current program counter of process 1 (P1) and process 2 (P2) is set to operation 1 (Send 'Set LEDs' command), and the two processes are currently in the ready state. The following could happen:

- When the scheduler selects P1 to run on the processor, operation 1 is immediately executed, and a command will be sent to keyboard.
- Now assume that P1 is preempted just after it has executed operation 1 (Send 'Set LEDs' command), due to timeout. When the scheduler selects P2 to run, it executes operation 1 because its program counter is set to operation 1 as mentioned above. This results in sending another command to the keyboard.
- The result is that the PS/2 keyboard panics, because it has received two commands in series without handshaking (i.e., acknowledging the previous keyboard command). The PS/2 protocol requires that the host should wait after sending a command, to give time for PS/2 protocol to reply with a handshake, then the host should send the data associated with the command, and finally, the keyboard replies with another handshake. The invalid operation of sending two commands in series might put the keyboard hardware in unstable state.

Notice that the case illustrated above is not the only case that may cause a race condition. You need to identify parts of your code that will cause races if executed by concurrent processes at the same time.

PART IV: Kernel Semaphores

Finally, you are required to avoid race conditions in your device driver by using synchronization methods provided by Linux kernel. Linux provides a variety of kernel synchronization mechanisms, including semaphore, spinlock, and mutex. For instance, the semaphore interface is very simple as the main API consists of three basic function calls:

- `sema_init()`, initialize the semaphore
- `down()`, decrement the counter
- `up()`, increment the counter

Example code to initialize and use a semaphore:

```
#include <linux/semaphore.h>

static struct semaphore sema;

void init(void) {
    sema_init(&sema, COUNTER_VALUE);
}

void critical_routine(void) {
    down(&sema); // enter critical region
    // your atomic code goes here...
    up(&sema);   // exit critical region
}
```

Listing 2: Semaphore Example Code

For more information:

- `http://www.makelinux.net/ldd3/chp-5-sect-3`
- `http://www.makelinux.net/ldd3/chp-5-sect-5`

Deliverables

- Complete source code, commented thoroughly and clearly.
- Your Makefile(s).
- A report that describes the following: (1) how your code is organized, (2) its main functions, and (3) how to compile and run your code.
- Sample runs.
- All deliverables are to be put in one directory named lab2_XX, where XX is your ID and then zipped. Use GNU ZIP or BZIP2, not RAR.
- You need to send your code to CS333F16@gmail.com by the deadline stated below. The subject line should be: "Assignment 2 - SID:XX".

Appendix I: Linux Kernel Module

Kernel modules are object files that can be loaded and unloaded into the kernel in runtime. Kernel modules are used to add support for device drivers, filesystem drivers, and other functionality to the base kernel.

In order to write a kernel module for your Linux kernel, you will need to install kernel headers (through the package manager of your distribution). The version of the kernel headers must match the version of your Linux kernel.

Next, you can get started by writing a simple Hello World kernel module:

```
#include <linux/module.h>
#include <linux/kernel.h>

int init_module(void) {
  printk(KERN_INFO "Hello world!\n");
  return 0;
}

void cleanup_module(void) {
  printk(KERN_INFO "Goodbye world!\n");
}
```

Listing 3: hello.c

The function call `printk()` is actually a logging mechanism for the kernel. You can use it to log information about what is going on inside the kernel, or give warnings to the users.

To compile your module, you need to provide a Makefile to translate C code into a kernel loadable object file (`.ko`):

```
obj-m += hello.o

all:
  make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules

clean:
  make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean
```

Listing 4: Makefile

In the first line, target object files are specified `(module_name.o)`. Make utility will look for `module_name.c` file to generate the object file. To execute the Makefile, type `make` in your terminal emulator and make sure that current directory is set to the directory where your kernel module and Makefile are stored:

```
[iocoder@inspiron hello]$ make
```

If your code is compiled successfully, `hello.ko` will be generated in the same directory where you are doing your work. You can load it to the kernel using `insmod` command:

```
[iocoder@inspiron hello]$ sudo insmod hello.ko
[sudo] password for iocoder:
```

Once the kernel module is loaded to the kernel, the function `init_module()` (declared in `hello.c`) is called. Check kernel logs to find the string printed by `printk`. This can be done using `dmesg` command:

```
[iocoder@inspiron hello]$ dmesg | tail
...
[14471.238291] Hello world!
```

Finally, unload the module and check `dmesg` for the goodbye message printed by `printk` in `cleanup_module()`:

```
[iocoder@inspiron hello]$ sudo rmmod hello
[iocoder@inspiron hello]$ dmesg | tail
...
[14471.238291] Hello world!
...
[14593.040704] Goodbye world!
```

For more information, you can refer to The Linux Kernel Module Programming Guide (LKMPG) under TLDP (The Linux Documentation Project) website:

- `http://tldp.org/LDP/lkmpg/2.6/html/index.html`

Appendix II: PS/2 Protocol

One of the main responsibilities of the operating system is to manage various hardware devices of the system. I/O devices are connected to the CPU through system bus. In IBM-compatible personal computers (PC), I/O devices appear to software as memory-mapped I/O registers, or as I/O ports in the isolated I/O address space.

In order to send commands to I/O devices and exchange data with them, device drivers need to access the memory-mapped registers or I/O ports allocated for the device. For the legacy PS/2 controller chip, two I/O ports are used:

- Port 0x60: `DATA` port.
- Port 0x64: `STATUS` port.

In x86, I/O ports are accessed using **`in`** and **`out`** assembly instructions. Linux kernel provides `inb()` and `outb()` as wrappers for assembly's 8-bit `in` and `out` instructions. For example, `inb(0x60)` will return the current reading of `DATA` port of PS/2 controller.

PS/2 controller is the I/O device that is responsible for interfacing your PS/2 keyboard, PS/2 mouse, speaker, A20 gate, and other pieces of hardware. PS/2 controller works as an interface between the operating system and the physical PS/2 keyboard. Figure 1 shows PS/2 keyboard subsystem.

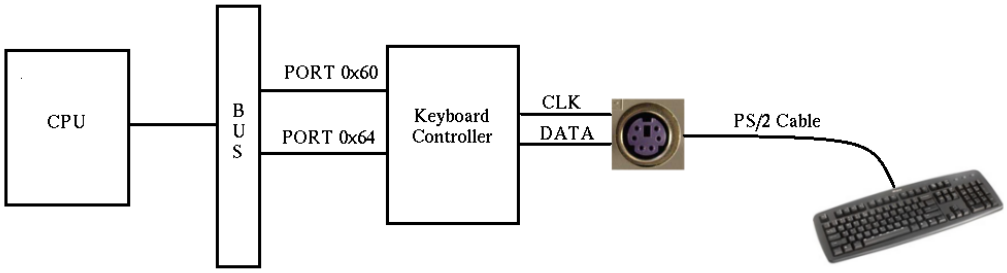

Figure 1: PS/2 Keyboard Subsystem

In the original IBM Personal System/2 computers, PS/2 keyboard controller was actually an 8042 microcontroller programmed with firmware that implemented PS/2 controller. In modern PC systems, the functionality of the PS/2 controller is implemented as part of Super I/O chip.

When a key is pressed down, PS/2 keyboard sends the scancode of the pressed key through DATA line of PS/2 cable. PS/2 controller receives the scancode and stores it in an 'output buffer', until OS reads port 0x60.

PS/2 protocol also supports sending data in the other direction. To send data to keyboard, the OS should write the databyte to port 0x60. PS/2 keyboard controller stores the databyte in an 'input buffer' until it is transferred over DATA line to the keyboard itself.

Data shouldn't be written to port 0x60 until 'input buffer' becomes empty. In the same way, data shouldn't be read from port 0x60 until 'output buffer' becomes full. To check the status of input/output buffers, the device driver needs to read port 0x64 (`STATUS` port). The format of status word is shown in Table 2.

Bit	Description
0	Output buffer status (0: empty, 1: full)
1	Input buffer status (0: empty, 1: full)
2	System flag
3	Command/data
4	Inhibit switch
5	Transmit timeout
6	Receive timeout
7	Parity error

Table 2: Status I/O Port

To interface PS/2 keyboard controller, you may need to implement the following procedures:

- `kbd_read_status()`, which simply does the following:
  1. Issue `inb(0x64)` to read the status of the controller.
  2. Return the status byte to the caller.
- `kbd_read_data()`, which follows these steps:
  1. Read `STATUS` register using `kbd_read_status()`.
  2. Check bit 0 of the `STATUS` register. If bit 0 is clear, this means that PS/2 controller didn't receive anything from the keyboard up till the moment. Goto step 1 and repeat the process again. Otherwise (bit 0 is set), continue to step 3.
  3. If control reaches this step, this means that the keyboard has actually sent some data to keyboard controller. Read the data using `inb(0x60)`.
  4. Return data to the caller.
- `kbd_write_data(unsigned char data)`:
  1. Read `STATUS` register using `kbd_read_status()`.
  2. Check bit 1 of the `STATUS` register. If bit 1 is set, this means that PS/2 controller is still processing last command/data sent to it. Goto step 1. Otherwise (bit 1 is clear), continue to step 3.
  3. If control reaches this step, this means that the keyboard controller is ready to receive data into its input buffer. Write the data using `outb(data, 0x60)`. The 8-bit data value will then be sent to PS/2 keyboard.

Now you can use the routines described above to exchange data with PS/2 keyboard. To control the state of the LEDs, send 0xED command byte to PS/2 keyboard, followed by the state of the three LEDs. The keyboard replies to each byte with 0xFA (`ACK`) on success, or 0xFE (`RESEND`) on failure. The following pseudo-code shows the steps of the algorithm.

```
int update_leds(unsigned char led_status_word) {
    unsigned char ret;
    // send 'Set LEDs' command
    kbd_write_data(0xED);
    // wait for ACK
    if (kbd_read_data() != 0xFA)
        return -1;
    // now send LED states
    kbd_write_data(led_status_word);
    // wait for ACK
    if (kbd_read_data() != 0xFA)
        return -1;
    // success
    return 0;
}
```

Listing 5: Update LEDs Routine

PS/2 keyboard uses `led_status_word` to update the states of the LEDs. The format of `led_status_word` is shown in Table 3. For example, `update_leds(5)` should turn on scroll lock and caps lock, and turn off num lock.

Bit	Description
0	Scroll lock
1	Num lock
2	Caps lock
3-7	Should be 0

Table 3: Format of LED Status Word

For more information, refer to the following links:

- `http://wiki.osdev.org/%228042%22_PS/2_Controller`
- `http://wiki.osdev.org/PS/2_Keyboard`
- `http://computer-engineering.org/ps2protocol/`
- `http://computer-engineering.org/ps2keyboard/`
- `http://zet.aluzina.org/images/d/d4/8042.pdf`