CSci 5103: Operating Systems

Project 4: Linux Device Drivers

Due: Midnight December 17, 2015

1. Objective

In this project you will be exposed to aspects of the "insides" of a real OS. Much of the OS code deals with device drivers. In this lab, you will program a Linux device driver and link it into the kernel. In particular, you will implement a character device called scullbuffer that implements a bounded buffer of fixed size to synchronize any number of producer and consumer processes. It will implement the character device interface. You are welcome to work on your laptops and modify your own local version of Linux if you dare. We have provided a virtualized environment on the CS machines for you to do this without crashing the underlying OS. Details to be provided. Thanks to Prof. Anand Tripathi for providing details of this lab.

2. Project Details

Both producer and consumer processes will first call the open function to access the device. At end of their use of the device, they will call the release function. The buffer will store "items", where each item will be a block of up to 512 bytes and an integer count indicating the number of bytes in the block.
A producer process will open the device in "write" mode. It will call the write function to deposit an "item" in the buffer. If a producer makes the write call to deposit more than 512 bytes of data, the driver will accept only the first 512 bytes of data. The return value of the write function will be the number of bytes written into a buffer item, or -1 if there was any error. It will return 0 if the buffer is full and there are no consumer processes. If the buffer is full and there are some consumer processes, then the producer process would get blocked in the write function. A consumer processes will open the device in the "read" mode. It will call the read function to retrieve an item from the buffer. The read function will copy the data block of the next available item into the address-space of the process, at starting location specified in the read function call. The return value of this function will be the number of bytes copied from the device buffer to the process address-space. This function will block if there is no item currently available to be consumed and there are some producer processes. If the buffer is empty and there no producer processes, then the read call will return with value zero.

In your program, use counting semaphores. (See Chapter 5 of the Linux Device Drivers book.) You will allocate a buffer in the kernel memory using the function `kmalloc` to hold up to *NITEMS*, which will be a parameter to your driver and specified at the device installation time. You will be implementing the following functions: `read, and write`. As part of the open function, you will need to maintain the counts of the currently active producer and consumer processes. The functions read and write will perform the appropriate synchronization using counting semaphores.
Please write a script for installing and removing the scullbuffer device and also write code for the producer and consumer processes to test your device. *You will start with an existing code that has has a scull buffer for reading and writing a buffer of char. You are just making this a bounded buffer.*

3. Important Notes on Implementation (most of are already shown in the code provided)

A reference of some of the tools and functions that might be of use is listed below.

Modules:
insmod
modprobe
rmmod

User-space utilities that load modules into the running kernels and remove them.

#include <linux/init.h>
```
module_init (init_function);
module_exit (cleanup_function);
```

Macros that designate a module’s initialization and cleanup functions.

#include <linux/kernel.h>
```
int printk (const char * fmt, ...);
```

The analogue of `printk` for kernel code.

Device Numbers:
#include <linux/types.h>
```
dev_t
```

`cdev` is the type used to represent device numbers within the kernel.

```
int MAJOR (dev_t dev);
int MINOR (dev_t dev);
```

Macros that extract the major and minor numbers from a device number.

dev_t MKDEV (unsigned int major, unsigned int minor);

Macro that builds a dev_t data item from the major and minor numbers.

Character Devices:
#include <linux/fs.h>

```
int register_chrdev_region (dev_t first, unsigned int count,
                            char *name)
int alloc_chrdev_region (dev_t *dev, unsigned int firstminor,
                         unsigned int count, char *name)
void unregister_chrdev_region (dev_t first, unsigned int count);
```

Functions that allow a driver to allocate and free ranges of device numbers.

`register_chrdev_region` should be used when the desired major number is known in advance; for dynamic allocation, use `alloc_chrdev_region` instead.

```
struct file_operations;
struct file;
struct inode;
```

Three important data structures used by most device drivers. The file operations structure holds a char driver's methods; struct file represents an open file, and struct inode represents a file on disk.

#include <linux/cdev.h>

```
struct cdev *cdev_alloc(void);
void cdev_init (struct cdev *dev, struct file_operations *fops);
int cdev_add (struct cdev *dev, dev_t num, unsigned int count);
void cdev_del (struct cdev *dev);
```

Functions for the management of `cdev` structures, which represent char devices within the kernel.

#include <linux/kernel.h>

```
container_of (pointer, type, field);
```

A convenience macro that may be used to obtain a pointer to a structure from a pointer to some other structure contained within it.

#include <asm/uaccess.h>

```
unsigned long copy_from_user  (void *to, const void *from,
                               unsigned long count);

unsigned long copy_to_user (void *to, const void *from,
                            unsigned long count);
```

Copy data between user space and kernel space.

Semaphores:

#include <asm/semaphore.h>

```
void sema_init (struct semaphore *sem, int val);
void down (struct semaphore *sem);
int down_interruptible (struct semaphore *sem);
int down_trylock (struct semaphore *sem);
void up (struct semaphore *sem);
```

Lock and unlock a semaphore. `down` puts the calling process into an uninterruptible sleep if need be; `down_interruptible`, instead, can be interrupted by a signal. `down_trylock` does not sleep; instead, it returns immediately if the semaphore is unavailable. Code that locks a semaphore must eventually unlock it with `up`.

Memory Allocation:

#include <linux/slab.h>

```
void *kmalloc (size_t size, int flags);
void kfree (void *obj);
```

The most frequently used interface to memory allocation.

4. Submission and Grading

Testing:

- You can run and test on a virtualized CS Unix machine.
- **<Instructions> on how to use the virtual machine will be provided by Friday.**

Submission:

- Zip up/Tar your code and submit it online to include:
  - scullbuffer.c
  - Script for installing and removing the scullbuffer device
  - producer.c
  - consumer.c
- Please include a readme.txt file in your assignment which has the following information:
  - Your names;
  - Your ITLab/CS login names;
  - Your student IDs;
  - Assignment number;
  - The hostname of the ITLab/CS machine you compiled and tested your code;
  - How to run your program
- No need to submit a paper copy. Save trees instead.

Grading:

- Driver registration, initialization and cleanup: 20%
- Correct use of kernel data structures and function calls: 10%
- Synchronization using counting semaphores: 30%
- Read / write functionality: 20%
- Producer, consumer test code: 20%

5. Reading List

A list of chapters and sections which would be of use while working on this project are listed below. The minimum list is Chapters 2, 3, 5. The rest are listed for completeness so that you can look things up. Feel free to explore beyond what is listed.
You can find the book here: http://lwn.net/Kernel/LDD3/

Chapter 1: Introduction to Device Drivers
Loadable Modules
Classes of Devices and Modules

Chapter 2: Building and Running Modules
The Hello World Module
Kernel Modules Versus Applications
User Space and Kernel Space
Concurrency in the Kernel
Compiling and Loading
Loading and Unloading Modules
Preliminaries
Initialization and Shutdown
Module-Loading Races
Module Parameters

Chapter 3: Char devices
The Design of scull
Major and Minor Numbers
The Internal Representation of Device Numbers
Allocating and Freeing Device Numbers
Dynamic Allocation of Major Numbers
Some Important Data Structures
Char Device Registration
open and release
scull's Memory Usage
read and write

Chapter 4: Debugging Techniques
Debugging by Printing

Chapter 5: Concurrency and Race Conditions
Concurrency and Its Management
Semaphores and Mutexes
Locking Traps
Alternatives to Locking

Chapter 6: Advanced Char Driver Operations
Simple Sleeping

Chapter 8: Allocating Memory
The Real Story of kmalloc