DM510 Operating systems
Assignment 2
Linux kernel module

Frederik List
frlis21@student.sdu.dk

August 23, 2024

Introduction

Welcome to the sequel in this three-part demonstration of my ability to make ChatGPT do my homework for me. This time, ~~ChatGPT~~ *I* created a Linux character device driver with the following features and characteristics:

- Exactly 2 devices, where one must write to one to read from the other.
- Puts processes that want to read/write from devices to sleep when buffers are empty/full.
- Several processes at a time (up to a configurable limit) can open devices for reading.
- Only one process at a time can open a device for writing.
- Simple device control via `ioctl` to configure reader limit and buffer sizes.
- ~~Deletes the operating system.~~

The rest of this document outlines my design decisions and implementation of the kernel module.

Design

There isn't much to talk about concerning the design of this specific kernel module, as it is relatively simple and some rigid design constraints were placed on it. However, there are design principles that apply to every Linux kernel module. For instance, kernel modules must support concurrency. It also seems to be the convention to err on the side of simplicity, then speed.

As for functionality, each device must be able to read from one buffer and write to another. This is done nicely with circular buffers, which could allow reads and writes simultaneously. Unfortunately, you will see in the implementation section how we throw away this speed advantage, in part because of our special design contraints.

That is all for the design of this kernel module.

Implementation

Our character device driver implementation is mostly quite standard.

To start, we allocate memory for our pair of `dm510_dev` structs, which will hold our circular buffers, locks, wait queues, reader/writer counts, and so on. The `dm510_dev` struct also has a member `struct dm510_dev *buddy`, which points to the "partner" device that will be written to, thus allowing the "criss-cross" functionality we require. Because we have this cycle of dependencies, we need to initialize all of the device structs in one loop before we can add the corresponding character devices in another loop, lest a device should reference an uninitialized buddy.

Upon opening a device file, we first check whether the reader/writer limit was reached, and return `-EBUSY` if this is the case. Otherwise, we increment the reader/writer count of the device and continue as normal. Reader/writer counts are implemented with atomics, so no locking is needed.

Closing a device file just decrements reader/writer counts.

Upon reading a device file, we acquire the mutex lock for the device file and simply copy data from our circular buffer to the reader's buffer. If the circular buffer is empty, we block (details below).

Writing to a device file is much the same as reading, except we must acquire our buddy's mutex lock and write to our buddy's circular buffer.

Circular buffer

We use Linux's implementation of a circular buffer, namely the `kfifo`. `kfifo` allows reader and writer access simultaneously, requiring extra locking only when there is more than one of each.

We could use reader/writer mutex locks to make use of simultaneous reading and writing, but then we would have to use another (read-write?) lock to protect our device struct, as the `kfifo` buffer can change in an `ioctl` call. I think this extra complexity is not worth it (considering our user-mode Linux instance is only running on one core anyway), so we use one mutex per device.

`ioctl`

Our `ioctl` implementation is pretty standard as well, except we treat the argument as a value instead of a pointer to userspace memory.

When the `ioctl` command for adjusting the buffer size is called, we allocate a new `kfifo` before freeing and updating the old `kfifo`. If allocation fails, we just print an error message and keep the old `kfifo`.

Blocking

Processes are never put to sleep if they open the device file with `O_NONBLOCK`.

Processes are put to sleep when they try to read and the circular buffer is empty or when they write and circular buffers are full.

Processes are awoken again when the complementary operation is finished, e.g. when a write has finished, processes waiting to read are woken up.

Notably, before we sleep, we release our mutex, but in our wait condition, we reference the `kfifo`. I think this is safe, as checking for an empty/full `kfifo` only requires reading data that we can guarantee will always be in the same place, so the worst thing that can happen is we get the wrong result. However, we also check the condition again while holding the mutex lock, so in total, we occasionally use extra CPU time just to grab and release locks.

Tests

To test the kernel module, I use the provided `moduletest` program as well as an `ioctl` controller program (see Listing A.2). I also use the `echo` and `cat` commands to write and read data.

Figure 1: Tests with video timestamps

Test	**Time**
Module loading and unloading, general tests with `echo` and `cat`.	00:00
`moduletest` (seeing how it works under heavy load, whether it deadlocks, etc.)	00:31
Various `ioctl` shenanigans.	00:39
Verifying that blocking works with `echo` and `cat`. This is done by setting the buffer sizes to be very small, then `echo`ing a large number of characters into the devices and `cat`ing them out again, all the while observing how we are being blocked as the devices wait for data or space.	01:10

Concurrency

Only one process at a time may open a device for writing, so there can be no contention between writers. There is a configurable limit for how many processes at a time may open a device for reading, so depending on the configured limit, there may or may not be some contention between readers when they eat data intended for another process.

As for safety, each device struct is protected by a single mutex which should be grabbed before making modifications.

Conclusion

Yep, it works! ~~Thanks ChatGPT!~~

For legal reasons, I didn't actually use ChatGPT for any of this (though I did try, but I only got garbage responses).

Appendix A

Source code

Listing A.1: dm510_dev.c

```
#ifndef __KERNEL__
#define __KERNEL__
#endif
#ifndef MODULE
#define MODULE
#endif

#include <linux/errno.h>
#include <linux/printk.h>
#include <linux/fs.h>
#include <linux/cdev.h>
#include <linux/module.h>
#include <linux/slab.h>
#include <linux/types.h>
#include <linux/wait.h>
#include <linux/mutex.h>
#include <linux/spinlock.h>
#include <linux/kfifo.h>
#include <linux/rwsem.h>

/* This would normally go in a .h file */
#define DEVICE_NAME "dm510_dev"
#define DM510_MAJOR 255
#define BASE_MINOR 0
#define DEVICE_COUNT 2

#define MAX_WRITERS 1
#define DEFAULT_MAX_READERS 16
#define DEFAULT_BUFFER_SIZE 1024

#define IOCTL_GET_MAX_READERS _IOR(DM510_MAJOR, 0, ssize_t)
#define IOCTL_SET_MAX_READERS _IOW(DM510_MAJOR, 1, size_t)
#define IOCTL_GET_BUFFER_SIZE _IOR(DM510_MAJOR, 2, ssize_t)
#define IOCTL_SET_BUFFER_SIZE _IOW(DM510_MAJOR, 3, size_t)
#define IOCTL_MAX_NR 3

struct dm510_dev {
        struct mutex mutex;
        struct dm510_dev *buddy;
        atomic_t nreaders, nwriters;
        unsigned long max_readers;
        struct kfifo kfifo;
        wait_queue_head_t readq, writeq;
        struct cdev cdev;
};

static int dm510_open(struct inode *, struct file *);
static int dm510_release(struct inode *, struct file *);
```

```
49  static ssize_t dm510_read(struct file *, char __user *, size_t, loff_t *);
50  static ssize_t dm510_write(struct file *, const char __user *, size_t,
51                             loff_t *);
52  static long dm510_ioctl(struct file *filp, unsigned int cmd, unsigned long arg);
53  /* end of what really should have been in a .h file */
54
55  static struct dm510_dev *dm510_devs = NULL;
56
57  static int dm510_open(struct inode *inode, struct file *filp)
58  {
59          struct dm510_dev *dev;
60          dev = container_of(inode->i_cdev, struct dm510_dev, cdev);
61          filp->private_data = dev;
62
63          pr_info("DM510: Opening DM510-%d.", MINOR(dev->cdev.dev));
64
65          if (filp->f_mode & FMODE_READ &&
66              atomic_fetch_inc(&dev->nreaders) >= dev->max_readers) {
67                  atomic_dec(&dev->nreaders);
68                  pr_info("DM510-%d: Too many readers.", MINOR(dev->cdev.dev));
69                  return -EBUSY;
70          }
71          /* Concurrent writers can kindly not */
72          if (filp->f_mode & FMODE_WRITE &&
73              atomic_fetch_inc(&dev->nwriters) >= MAX_WRITERS) {
74                  atomic_dec(&dev->nwriters);
75                  pr_info("DM510-%d: Too many writers.", MINOR(dev->cdev.dev));
76                  return -EBUSY;
77          }
78
79          /* We don't implement lseek, it doesn't make much sense here... */
80          return nonseekable_open(inode, filp);
81  }
82
83  static int dm510_release(struct inode *inode, struct file *filp)
84  {
85          struct dm510_dev *dev;
86          dev = filp->private_data;
87
88          pr_info("DM510: Closing DM510-%d.", MINOR(dev->cdev.dev));
89
90          if (filp->f_mode & FMODE_WRITE)
91                  atomic_dec(&dev->nwriters);
92          if (filp->f_mode & FMODE_READ)
93                  atomic_dec(&dev->nreaders);
94
95          return 0;
96  }
97
98  static ssize_t dm510_read(struct file *filp, char __user *buf, size_t count,
99                            loff_t *f_pos)
100 {
101         struct dm510_dev *dev;
102         int ret;
103         unsigned int copied;
104
105         dev = filp->private_data;
106         if (mutex_lock_interruptible(&dev->mutex))
107                 return -ERESTARTSYS;
108         while (kfifo_is_empty(&dev->kfifo)) {
109                 mutex_unlock(&dev->mutex);
110                 if (filp->f_flags & O_NONBLOCK)
111                         return -EAGAIN;
112                 if (wait_event_interruptible(dev->readq,
113                                              !kfifo_is_empty(&dev->kfifo)))
114                         return -ERESTARTSYS;
115                 if (mutex_lock_interruptible(&dev->mutex))
116                         return -ERESTARTSYS;
```

```
117         }
118         ret = kfifo_to_user(&dev->kfifo, buf, count, &copied);
119         mutex_unlock(&dev->mutex);
120         if (unlikely(ret))
121                 return ret;
122
123         wake_up_interruptible(&dev->buddy->writeq);
124
125         return copied;
126 }
127
128 static ssize_t dm510_write(struct file *filp, const char __user *buf,
129                            size_t count, loff_t *f_pos)
130 {
131         struct dm510_dev *dev, *buddy;
132         int ret;
133         unsigned int copied;
134
135         dev = filp->private_data;
136         buddy = dev->buddy;
137
138         if (mutex_lock_interruptible(&buddy->mutex))
139                 return -ERESTARTSYS;
140         while (kfifo_is_full(&buddy->kfifo)) {
141                 mutex_unlock(&buddy->mutex);
142                 if (filp->f_flags & O_NONBLOCK)
143                         return -EAGAIN;
144                 if (wait_event_interruptible(dev->writeq,
145                                              !kfifo_is_full(&buddy->kfifo)))
146                         return -ERESTARTSYS;
147                 if (mutex_lock_interruptible(&buddy->mutex))
148                         return -ERESTARTSYS;
149         }
150         ret = kfifo_from_user(&buddy->kfifo, buf, count, &copied);
151         mutex_unlock(&buddy->mutex);
152         if (unlikely(ret))
153                 return ret;
154
155         wake_up_interruptible(&buddy->readq);
156
157         return copied;
158 }
159
160 static long resize_kfifo(struct dm510_dev *dev, unsigned long buffer_size)
161 {
162         int ret;
163         struct kfifo new_kfifo;
164
165         ret = kfifo_alloc(&new_kfifo, buffer_size, GFP_KERNEL);
166         if (ret) {
167                 pr_err("DM510: Resizing kfifo for DM510-%d failed with error code %d.\n",
168                        MINOR(dev->cdev.dev), ret);
169                 return ret;
170         }
171         /* We're already holding the lock */
172         kfifo_free(&dev->kfifo);
173         dev->kfifo = new_kfifo;
174
175         pr_info("DM510: Resized kfifo for DM510-%d.\n", MINOR(dev->cdev.dev));
176
177         return 0;
178 }
179
180 static long dm510_ioctl(struct file *filp, unsigned int cmd, unsigned long arg)
181 {
182         struct dm510_dev *dev;
183         long ret = 0;
184
```

```
185         dev = filp->private_data;
186
187         if (mutex_lock_interruptible(&dev->mutex))
188                 return -ERESTARTSYS;
189
190         switch (cmd) {
191         case IOCTL_GET_MAX_READERS:
192                 pr_info("DM510: GET_MAX_READERS ioctl called.\n");
193                 ret = dev->max_readers;
194                 break;
195         case IOCTL_SET_MAX_READERS: {
196                 pr_info("DM510: SET_MAX_READERS ioctl called.\n");
197                 if (!capable(CAP_SYS_ADMIN)) {
198                         ret = -EPERM;
199                         break;
200                 }
201                 dev->max_readers = arg;
202                 break;
203         }
204         case IOCTL_GET_BUFFER_SIZE:
205                 pr_info("DM510: GET_BUFFER_SIZE ioctl called.\n");
206                 ret = kfifo_size(&dev->kfifo);
207                 break;
208         case IOCTL_SET_BUFFER_SIZE: {
209                 pr_info("DM510: SET_BUFFER_SIZE ioctl called.\n");
210                 if (!capable(CAP_SYS_ADMIN)) {
211                         ret = -EPERM;
212                         break;
213                 }
214                 ret = resize_kfifo(dev, arg);
215                 break;
216         }
217         default:
218                 pr_info("DM510: ioctl called with unknown command.\n");
219                 ret = -ENOIOCTLCMD;
220         }
221
222         mutex_unlock(&dev->mutex);
223
224         return ret;
225 }
226
227 static struct file_operations dm510_fops = {
228         .owner = THIS_MODULE,
229         .read = dm510_read,
230         .write = dm510_write,
231         .open = dm510_open,
232         .release = dm510_release,
233         .unlocked_ioctl = dm510_ioctl,
234         .llseek = no_llseek,
235 };
236
237 static void dm510_exit(void)
238 {
239         if (dm510_devs) {
240                 for (int i = 0; i < DEVICE_COUNT; i++) {
241                         cdev_del(&dm510_devs[i].cdev);
242                         kfifo_free(&dm510_devs[i].kfifo);
243                 }
244                 kfree(dm510_devs);
245         }
246
247         unregister_chrdev_region(MKDEV(DM510_MAJOR, BASE_MINOR), DEVICE_COUNT);
248         pr_info("DM510: Module unloaded.\n");
249 }
250
251 #define CHECK_ERR(exp)                  \
252         {                               \
```

```
253                 int __ret = exp;        \
254                 if (unlikely(__ret)) { \
255                         dm510_exit();   \
256                         return __ret;   \
257                 }                       \
258         }
259
260 static int __init dm510_init(void)
261 {
262         int ret;
263
264         ret = register_chrdev_region(MKDEV(DM510_MAJOR, BASE_MINOR),
265                                      DEVICE_COUNT, DEVICE_NAME);
266         if (ret)
267                 return ret;
268
269         dm510_devs =
270                 kmalloc(DEVICE_COUNT * sizeof(struct dm510_dev), GFP_KERNEL);
271         if (!dm510_devs) {
272                 dm510_exit();
273                 return -ENOMEM;
274         }
275
276         for (int i = 0; i < DEVICE_COUNT; i++) {
277                 cdev_init(&dm510_devs[i].cdev, &dm510_fops);
278                 dm510_devs[i].cdev.owner = THIS_MODULE;
279                 dm510_devs[i].buddy = &dm510_devs[(i + 1) % DEVICE_COUNT];
280                 dm510_devs[i].max_readers = DEFAULT_MAX_READERS;
281                 atomic_set(&dm510_devs[i].nreaders, 0);
282                 mutex_init(&dm510_devs[i].mutex);
283                 init_waitqueue_head(&dm510_devs[i].readq);
284                 init_waitqueue_head(&dm510_devs[i].writeq);
285                 CHECK_ERR(kfifo_alloc(&dm510_devs[i].kfifo, DEFAULT_BUFFER_SIZE,
286                                       GFP_KERNEL));
287         }
288
289         /* Add cdevs *after* initializing everything at once because we depend on buddies */
290         for (int i = 0; i < DEVICE_COUNT; i++) {
291                 CHECK_ERR(cdev_add(&dm510_devs[i].cdev,
292                                    MKDEV(DM510_MAJOR, BASE_MINOR + i), 1));
293         }
294
295         pr_info("DM510: Module installed.\n");
296
297         return 0;
298 }
299
300 module_init(dm510_init);
301 module_exit(dm510_exit);
302
303 MODULE_AUTHOR("...Frederik List");
304 MODULE_LICENSE("GPL");
305 MODULE_DESCRIPTION("DM510 module");
```

Listing A.2: `ioctl.c`

```
1  #include <stdio.h>
2  #include <unistd.h>
3  #include <sys/types.h>
4  #include <sys/ioctl.h>
5  #include <stdlib.h>
6  #include <fcntl.h>
7  #include <errno.h>
8  #include <string.h>
9
10 #define DM510_MAJOR 255
11 #define IOCTL_GET_MAX_READERS _IOR(DM510_MAJOR, 0, ssize_t)
```

```c
#define IOCTL_SET_MAX_READERS _IOW(DM510_MAJOR, 1, size_t)
#define IOCTL_GET_BUFFER_SIZE _IOR(DM510_MAJOR, 2, ssize_t)
#define IOCTL_SET_BUFFER_SIZE _IOW(DM510_MAJOR, 3, size_t)
#define IOCTL_MAX_NR 3

unsigned long ioctl_cmds[] = {
        IOCTL_GET_MAX_READERS,
        IOCTL_SET_MAX_READERS,
        IOCTL_GET_BUFFER_SIZE,
        IOCTL_SET_BUFFER_SIZE,
};

char *cmd_names[] = {
        "get_max_readers",
        "set_max_readers",
        "get_buffer_size",
        "set_buffer_size",
};

int main(int argc, char *argv[])
{
        if (argc < 3) {
                return 1;
        }

        int fd;
        long result;
        char *cmd_name;
        unsigned long cmd, arg;

        fd = open(argv[1], O_RDWR);
        if (fd < 0) {
                perror("open");
                return 1;
        }

        cmd_name = argv[2];
        if (argc > 3)
                arg = strtol(argv[3], NULL, 10);

        for (int i = 0; i <= IOCTL_MAX_NR; i++) {
                if (strcmp(cmd_name, cmd_names[i]) == 0) {
                        cmd = ioctl_cmds[i];
                        break;
                }
        }

        result = argc > 3 ? ioctl(fd, cmd, arg) : ioctl(fd, cmd);
        close(fd);

        if (result < 0) {
                perror("ioctl");
                return result;
        }

        if (argc == 3)
                printf("%ld\n", result);

        return 0;
}
```