Advanced Operating Systems
Assignment Week 4

Paul Lödige
Student ID: 37-229753

October 31, 2022

1 Assignment

Modify the kernel module you created in the #3 Homework to introduce the ioctl mechanism that sets and gets messages. Compile and run the program. You should also write a user-space program that calls ioctl(). Submit the source code and the screen shot of the result of execution displayed by the dmesg command.

2 Code

2.1 Kernel Module Code

2.1.1 chardev2.c

```c
/*
 * chardev2.c - Create an input/output character device
 */

#include <linux/cdev.h>
#include <linux/delay.h>
#include <linux/device.h>
#include <linux/fs.h>
#include <linux/init.h>
#include <linux/irq.h>
#include <linux/kernel.h> /* We are doing kernel work */
#include <linux/module.h> /* Specifically, a module */
#include <linux/poll.h>

#include "chardev.h"
#define SUCCESS 0
#define DEVICE_NAME "char_dev"
#define BUF_LEN 80

enum {
    CDEV_NOT_USED = 0,
    CDEV_EXCLUSIVE_OPEN = 1,
};

/* Is the device open right now? Used to prevent concurrent access into
 * the same device
 */
static atomic_t already_open = ATOMIC_INIT(CDEV_NOT_USED);

/* The message the device will give when asked */
static char message[BUF_LEN + 1];

static struct class *cls;

/* This is called whenever a process attempts to open the device file */
static int device_open(struct inode *inode, struct file *file)
{
    pr_info("device_open(%p)\n", file);

    try_module_get(THIS_MODULE);
    return SUCCESS;
}

static int device_release(struct inode *inode, struct file *file)
{
    pr_info("device_release(%p,%p)\n", inode, file);

    module_put(THIS_MODULE);
    return SUCCESS;
}

/* This function is called whenever a process which has already opened the
 * device file attempts to read from it.
 */
```

```
54 static ssize_t device_read(struct file *file, /* see include/linux/fs.h   */
55                            char __user *buffer, /* buffer to be filled  */
56                            size_t length, /* length of the buffer     */
57                            loff_t *offset)
58 {
59     /* Number of bytes actually written to the buffer */
60     int bytes_read = 0;
61     /* How far did the process reading the message get? Useful if the
       message
62      * is larger than the size of the buffer we get to fill in device_read.
63      */
64     const char *message_ptr = message;
65 
66     if (!*(message_ptr + *offset)) { /* we are at the end of message */
67         *offset = 0; /* reset the offset */
68         return 0; /* signify end of file */
69     }
70 
71     message_ptr += *offset;
72 
73     /* Actually put the data into the buffer */
74     while (length && *message_ptr) {
75         /* Because the buffer is in the user data segment, not the kernel
76          * data segment, assignment would not work. Instead, we have to
77          * use put_user which copies data from the kernel data segment to
78          * the user data segment.
79          */
80         put_user(*(message_ptr++), buffer++);
81         length--;
82         bytes_read++;
83     }
84 
85     pr_info("Read %d bytes, %ld left\n", bytes_read, length);
86 
87     *offset += bytes_read;
88 
89     /* Read functions are supposed to return the number of bytes actually
90      * inserted into the buffer.
91      */
92     return bytes_read;
93 }
94 
95 /* called when somebody tries to write into our device file. */
96 static ssize_t device_write(struct file *file, const char __user *buffer,
97                             size_t length, loff_t *offset)
98 {
99     int i;
100 
101     pr_info("device_write(%p,%p,%ld)", file, buffer, length);
102 
103     for (i = 0; i < length && i < BUF_LEN; i++)
104         get_user(message[i], buffer + i);
105 
106     /* Again, return the number of input characters used. */
107     return i;
108 }
109 
110 /* This function is called whenever a process tries to do an ioctl on our
111  * device file. We get two extra parameters (additional to the inode and
       file
112  * structures, which all device functions get): the number of the ioctl
       called
```

```
113  * and the parameter given to the ioctl function.
114  *
115  * If the ioctl is write or read/write (meaning output is returned to the
116  * calling process), the ioctl call returns the output of this function.
117  */
118 static long
119 device_ioctl(struct file *file, /* ditto */
120              unsigned int ioctl_num, /* number and param for ioctl */
121              unsigned long ioctl_param)
122 {
123     int i;
124     long ret = SUCCESS;
125 
126     /* We don't want to talk to two processes at the same time. */
127     if (atomic_cmpxchg(&already_open, CDEV_NOT_USED, CDEV_EXCLUSIVE_OPEN))
128         return -EBUSY;
129 
130     /* Switch according to the ioctl called */
131     switch (ioctl_num) {
132     case IOCTL_SET_MSG: {
133         /* Receive a pointer to a message (in user space) and set that to
134          * be the device's message. Get the parameter given to ioctl by
135          * the process.
136          */
137         char __user *tmp = (char __user *)ioctl_param;
138         char ch;
139 
140         /* Find the length of the message */
141         get_user(ch, tmp);
142         for (i = 0; ch && i < BUF_LEN; i++, tmp++)
143             get_user(ch, tmp);
144 
145         device_write(file, (char __user *)ioctl_param, i, NULL);
146         break;
147     }
148     case IOCTL_GET_MSG: {
149         loff_t offset = 0;
150 
151         /* Give the current message to the calling process - the parameter
152          * we got is a pointer, fill it.
153          */
154         i = device_read(file, (char __user *)ioctl_param, 99, &offset);
155 
156         /* Put a zero at the end of the buffer, so it will be properly
157          * terminated.
158          */
159         put_user('\0', (char __user *)ioctl_param + i);
160         break;
161     }
162     case IOCTL_GET_NTH_BYTE:
163         /* This ioctl is both input (ioctl_param) and output (the return
164          * value of this function).
165          */
166         ret = (long)message[ioctl_param];
167         break;
168     }
169 
170     /* We're now ready for our next caller */
171     atomic_set(&already_open, CDEV_NOT_USED);
172 
173     return ret;
174 }
```

```
175
176 /* Module Declarations */
177
178 /* This structure will hold the functions to be called when a process does
179  * something to the device we created. Since a pointer to this structure
180  * is kept in the devices table, it can't be local to init_module. NULL is
181  * for unimplemented functions.
182  */
183 static struct file_operations fops = {
184     .read = device_read,
185     .write = device_write,
186     .unlocked_ioctl = device_ioctl,
187     .open = device_open,
188     .release = device_release, /* a.k.a. close */
189 };
190
191 /* Initialize the module - Register the character device */
192 static int __init chardev2_init(void)
193 {
194     /* Register the character device (atleast try) */
195     int ret_val = register_chrdev(MAJOR_NUM, DEVICE_NAME, &fops);
196
197     /* Negative values signify an error */
198     if (ret_val < 0) {
199         pr_alert("%s failed with %d\n",
200                  "Sorry, registering the character device ", ret_val);
201         return ret_val;
202     }
203
204     cls = class_create(THIS_MODULE, DEVICE_FILE_NAME);
205     device_create(cls, NULL, MKDEV(MAJOR_NUM, 0), NULL, DEVICE_FILE_NAME);
206
207     pr_info("Device created on /dev/%s\n", DEVICE_FILE_NAME);
208
209     return 0;
210 }
211
212 /* Cleanup - unregister the appropriate file from /proc */
213 static void __exit chardev2_exit(void)
214 {
215     device_destroy(cls, MKDEV(MAJOR_NUM, 0));
216     class_destroy(cls);
217
218     /* Unregister the device */
219     unregister_chrdev(MAJOR_NUM, DEVICE_NAME);
220 }
221
222 module_init(chardev2_init);
223 module_exit(chardev2_exit);
224
225 MODULE_LICENSE("GPL");
```

2.1.2 chardev.h

```c
/*
 * chardev.h - the header file with the ioctl definitions.
 *
 * The declarations here have to be in a header file, because they need
 * to be known both to the kernel module (in chardev2.c) and the process
 * calling ioctl() (in userspace_ioctl.c).
 */

#ifndef CHARDEV_H
#define CHARDEV_H

#include <linux/ioctl.h>

/* The major device number. We can not rely on dynamic registration
 * any more, because ioctls need to know it.
 */
#define MAJOR_NUM 100

/* Set the message of the device driver */
#define IOCTL_SET_MSG _IOW(MAJOR_NUM, 0, char *)
/* _IOW means that we are creating an ioctl command number for passing
 * information from a user process to the kernel module.
 *
 * The first arguments, MAJOR_NUM, is the major device number we are using.
 *
 * The second argument is the number of the command (there could be several
 * with different meanings).
 *
 * The third argument is the type we want to get from the process to the
 * kernel.
 */

/* Get the message of the device driver */
#define IOCTL_GET_MSG _IOR(MAJOR_NUM, 1, char *)
/* This IOCTL is used for output, to get the message of the device driver.
 * However, we still need the buffer to place the message in to be input,
 * as it is allocated by the process.
 */

/* Get the n'th byte of the message */
#define IOCTL_GET_NTH_BYTE _IOWR(MAJOR_NUM, 2, int)
/* The IOCTL is used for both input and output. It receives from the user
 * a number, n, and returns message[n].
 */

/* The name of the device file */
#define DEVICE_FILE_NAME "char_dev"
#define DEVICE_PATH "/dev/char_dev"

#endif
```

2.2 IOCTL Client Code

2.2.1 userspace_ioctl.c

```c
/*  userspace_ioctl.c - the process to use ioctl's to control the kernel
    module
 *
 *  Until now we could have used cat for input and output.  But now
 *  we need to do ioctl's, which require writing our own process.
 */

/* device specifics, such as ioctl numbers and the
 * major device file. */
#include "chardev.h"

#include <stdio.h> /* standard I/O */
#include <fcntl.h> /* open */
#include <unistd.h> /* close */
#include <stdlib.h> /* exit */
#include <sys/ioctl.h> /* ioctl */

/* Functions for the ioctl calls */

int ioctl_set_msg(int file_desc, char *message)
{
    int ret_val;

    ret_val = ioctl(file_desc, IOCTL_SET_MSG, message);

    if (ret_val < 0) {
        printf("ioctl_set_msg failed:%d\n", ret_val);
    }

    return ret_val;
}

int ioctl_get_msg(int file_desc)
{
    int ret_val;
    char message[100] = { 0 };

    /* Warning - this is dangerous because we don't tell
   * the kernel how far it's allowed to write, so it
   * might overflow the buffer. In a real production
   * program, we would have used two ioctls - one to tell
   * the kernel the buffer length and another to give
   * it the buffer to fill
   */
    ret_val = ioctl(file_desc, IOCTL_GET_MSG, message);

    if (ret_val < 0) {
        printf("ioctl_get_msg failed:%d\n", ret_val);
    }
    printf("get_msg message:%s", message);

    return ret_val;
}

int ioctl_get_nth_byte(int file_desc)
{
    int i, c;
```

```
56
57      printf("get_nth_byte message:");
58
59      i = 0;
60      do {
61          c = ioctl(file_desc, IOCTL_GET_NTH_BYTE, i++);
62
63          if (c < 0) {
64              printf("\nioctl_get_nth_byte failed at the %d'th byte:\n", i);
65              return c;
66          }
67
68          putchar(c);
69      } while (c != 0);
70
71      return 0;
72  }
73
74  /* Main - Call the ioctl functions */
75  int main(void)
76  {
77      int file_desc, ret_val;
78      char *msg = "Message passed by ioctl\n";
79
80      file_desc = open(DEVICE_PATH, O_RDWR);
81      if (file_desc < 0) {
82          printf("Can't open device file: %s, error:%d\n", DEVICE_PATH,
83                 file_desc);
84          exit(EXIT_FAILURE);
85      }
86
87      ret_val = ioctl_set_msg(file_desc, msg);
88      if (ret_val)
89          goto error;
90      ret_val = ioctl_get_nth_byte(file_desc);
91      if (ret_val)
92          goto error;
93      ret_val = ioctl_get_msg(file_desc);
94      if (ret_val)
95          goto error;
96
97      close(file_desc);
98      return 0;
99  error:
100     close(file_desc);
101     exit(EXIT_FAILURE);
102 }
```

2.2.2 chardev.h

```
/*
 * chardev.h - the header file with the ioctl definitions.
 *
 * The declarations here have to be in a header file, because they need
 * to be known both to the kernel module (in chardev2.c) and the process
 * calling ioctl() (in userspace_ioctl.c).
 */

#ifndef CHARDEV_H
#define CHARDEV_H

#include <linux/ioctl.h>

/* The major device number. We can not rely on dynamic registration
 * any more, because ioctls need to know it.
 */
#define MAJOR_NUM 100

/* Set the message of the device driver */
#define IOCTL_SET_MSG _IOW(MAJOR_NUM, 0, char *)
/* _IOW means that we are creating an ioctl command number for passing
 * information from a user process to the kernel module.
 *
 * The first arguments, MAJOR_NUM, is the major device number we are using.
 *
 * The second argument is the number of the command (there could be several
 * with different meanings).
 *
 * The third argument is the type we want to get from the process to the
 * kernel.
 */

/* Get the message of the device driver */
#define IOCTL_GET_MSG _IOR(MAJOR_NUM, 1, char *)
/* This IOCTL is used for output, to get the message of the device driver.
 * However, we still need the buffer to place the message in to be input,
 * as it is allocated by the process.
 */

/* Get the n'th byte of the message */
#define IOCTL_GET_NTH_BYTE _IOWR(MAJOR_NUM, 2, int)
/* The IOCTL is used for both input and output. It receives from the user
 * a number, n, and returns message[n].
 */

/* The name of the device file */
#define DEVICE_FILE_NAME "char_dev"
#define DEVICE_PATH "/dev/char_dev"

#endif
```

3 Output of `sudo cat /proc/devices`

```
Character devices:
1 mem
4 /dev/vc/0
4 tty
4 ttyS
5 /dev/tty
5 /dev/console
5 /dev/ptmx
7 vcs
10 misc
13 input
29 fb
100 char_dev
116 alsa
128 ptm
136 pts
180 usb
188 ttyUSB
189 usb_device
202 cpu/msr
203 cpu/cpuid
226 drm
236 binder
237 hidraw
238 wwan_port
239 nvme-generic
240 nvme
241 aux
242 bsg
243 watchdog
244 remoteproc
245 ptp
246 pps
247 cec
248 lirc
249 rtc
250 dma_heap
251 dax
252 dimmctl
253 ndctl
254 gpiochip

Block devices:
8 sd
65 sd
66 sd
67 sd
68 sd
69 sd
70 sd
71 sd
128 sd
129 sd
130 sd
131 sd
132 sd
133 sd
134 sd
135 sd
259 blkext
```

4 Output of running the `userspace_ioctl.c` code

```
get_nth_byte message:Message passed by ioctl
get_msg message:Message passed by ioctl
```

5 Output of `sudo dmesg -T -l info | tail -5`

```
[Mon Oct 31 11:57:15 2022] Device created on /dev/char_dev
[Mon Oct 31 11:57:16 2022] device_open(00000000d7150407)
[Mon Oct 31 11:57:16 2022] device_write(00000000d7150407,000000001ed918c3,25)
[Mon Oct 31 11:57:16 2022] Read 24 bytes, 75 left
[Mon Oct 31 11:57:16 2022] device_release(0000000005e64dfb,00000000d7150407)
```