System Programming Project #3: Device Driver Report

Our group, unfortunately, could not complete the project. Therefore, the report will have two parts: what is done and what should have been done.

What Our Group Did

We were able to send a message to the driver and read the message without the recipient. The commands, data structures and functions are explained below.

Commands

By executing the following commands, we could write to and read from the device.

```
echo to User: Hello user!! > /dev/msgbox
cat /dev/msgbox
Hello user!!
```

The driver would parse the input and extract the recipient and store it in the relevant data structure which will be explained later in this document. However, we were unable to extract the sender information.

To compile and create the device the following commands were used.

Makefile

```
obj-m := msgbox.o
```

Shell commands:

```
make –C /lib/modules/3.8.13.6-custom/build M=~/Desktop/msgbox/ modules
sudo insmod ./msgbox.ko
sudo mknod /dev/msgbox c 61 0
sudo chmod 666 /dev/msgbox
```

`make` command compiles the c file and outputs the kernel module file (.ko). `insmod` command inserts the kernel module to the system. `mknod` command creates the device with the parameters c (char device), 61 (major number), 0 (minor number). `chmod` command sets the rwx permissions of the device. 666 (110 110 110) means read and write but no execution permissions are given to user, group and all others.

At this point the device is ready to be used, i.e. the `echo` and `cat` commands may be used as illustrated above.

Data Structures & Functions

There are 3 structs used for the device: a file operations struct to represent file operations associated with the device, a message struct to represent messages that are sent and kept in the message box, and a message box device struct to represent the message box which stores messages. Since the whole project could not be completed, only the variables that are used during the execution are displayed. There would be more variables in each struct normally, which will be the case in the “What should have been done” sub-topic.

```
struct file_operations msgbox_fops = {
      read: msgbox_read,
      write: msgbox_write,
      open: msgbox_open,
      release: msgbox_release
};

struct message {
      char *to, *data;
};

struct msgbox_dev {
      struct message* messages;
      unsigned msg_count;
      struct cdev cdev;
};
```

File operations represent simple read, write, open and release options. `message` struct keeps the recipient and the body of the text. `msgbox_dev` keeps the array of messages that reside in the message box, total number of messages and the device variable.

In the `msgbox_init()` function, the device is registered to the system using `register_chrdev()` function and all of the pointers are set to null after the memory is allocated for them while in the `msgbox_exit()` function, the device is unregistered and all the allocated memory are freed.

`msgbox_open` and `msgbox_release` functions do not perform any operation. This is the reason why we could not access to the elements of the variable of type `msgbox_dev*` but only could access to the variable of type `message*`, thus could only write one message to device and read the message from the device. We needed to set the file pointers private data to the `cdev` variable of the `messagebox` struct.

```
ssize_t msgbox_write(struct file *filp, char *buf, size_t count, loff_t
*f_pos){
      int i = 0, begin, size = 0;
      if(*f_pos >= count)
            goto out;

      /* extract the name from buf */
      /*
       * parsing and extraction code here
       */

      copy_from_user(msg_buffer->data, &buf[begin+size+1],
                       count-(begin+size+1));

      *f_pos += count;
      out:
      return 1;
}
```

This simplified version of the write function works as follows: the function first checks if the current position is in the boundary of the message that is going to be written into device coming from user space via buffer. If not, the function terminates. Then it parses the incoming message and extracts the recipient information. Note that the format of the incoming message should be fixed: as in "to USER: message". Otherwise the program would crash. After the parsing, the program copies the buffer to the `message* msg_buffer` variable, which is the buffer variable used for transferring data from kernel space to user space. Finally the file position is incremented and the function returns.

For the sake of simplicity (we avoided dynamic memory allocation in kernel space in the beginning), the message box's message array was initialized with fixed size [3]. After successfully writing message to and reading from device, we decided to save the message in the `msgbox_dev* messagebox`'s `messages` array. However, after initializing the `messagebox`, any attempt to access any element of the `messagebox` caused segmentation fault. The reason is explained above (empty open/close function).

```
ssize_t msgbox_read(struct file *filp, char *buf, size_t count, loff_t
*f_pos){
      ssize_t retval = 0;
      int i;

      if (*f_pos + count > 100)    // 100 -> max size of a msg
        count = 100 - *f_pos;
      if(msg_buffer->data == NULL){
            printk("buffer null\n");
            goto out;
      }

      if (copy_to_user(buf, msg_buffer->data, count)) {
            retval = -EFAULT;
            goto out;
      }

      *f_pos += count;
      retval = count;

      out:
      return retval;
}
```

msgbox_read function makes sure the position pointer is in the boundaries of the message being read and the message data is not null. After that the message residing in the device is transferred from kernel space to user space (only the text of the message, not the recipient). Position pointer is updated to the correct position and the function returns.

What Should Have Been Done

Obviously, the open function should have been written for the msgbox device to work. Provided that we can save messages into the `messagebox` variable successfully, we should looped through the `messages` array of `messagebox` to display the messages. The data structures would look like the following:

```
struct file_operations msgbox_fops = {
      read: msgbox_read,
      write: msgbox_write,
      open: msgbox_open,
      release: msgbox_release,
      io_ctl: msgbox_ioctl
};

struct message {
      char *to, *data;
      char *from;
      int read;  //0 or 1
};

struct msgbox_dev {
      struct message* messages;
      unsigned msg_count;
      struct cdev cdev;
      int mode;   //0: EXCLUDE_READ | 1: INCLUDE_READ
};
```

According to the mode of the `messagebox`, the read function would display only unread messages (EXCLUDE_READ) by checking the `mode` variable or would display every message in the `messagebox` (INCLUDE_READ) to the relevant user by comparing the current username with the `message`'s `from` variable.

To change the mode of the `messagebox`, the (super)user would call the `msgbox_ioctl` function. Among the completion of permission check in the `msgbox_ioctl()` function, either the mode would be changed or an error message is displayed (when non-superuser tries to change the mode).