Operating Systems (0368216208)

Message Slot Kernel Module Assignment

Due date (via moodle): **May 31th, 23:59**

Individual work policy

The work you submit in this course is required to be the result of your individual effort only. You may discuss concepts and ideas with others, but **you must program individually**. **You should never observe another student's code**, from this or previous semesters.

Students violating this policy will receive a **course grade of 250** ("did not complete course requirements").

1 Introduction

The goal of this assignment is to gain experience with kernel programming and, particularly, a better understanding on the design and implementation of inter-process communication (IPC), kernel modules, and drivers.

In this assignment, you will implement a kernel module that provides a new IPC mechanism, called a *message slot*. A message slot is a character device file through which processes communicate. A message slot device has multiple *message channels* active concurrently, which can be used by multiple processes. After opening a message slot device file, a process uses `ioctl()` to specify the id of the message channel it wants to use. It subsequently uses `read()`/`write()` to receive/send messages on the channel. In contrast to pipes, **a message channel preserves a message until it is overwritten, so the same message can be read multiple times**.

2 Message slot specification

A message slot appears in the system as a character device file. This device file is managed by the message slot device driver, which you will implement. A message slot is a *pseudo* device: it doesn't correspond to a physical hardware device, and therefore all its functionality is provided by the device driver.

Message slot device files. A device file has a *major* number and a *minor* number. The major number tells the kernel which driver is associated with the device file. The minor number is used internally by the driver for its own purposes.

In our case: There can be several message slot files, which correspond to different message slots. All of these files are managed by your driver, i.e., **they all have the same major number, which is hard-coded to 235**. However, different message slot files will have different minor numbers, allowing your driver to distinguish between them.

Device files are created with the `mknod` command, which takes as arguments the file's major and minor numbers. For example:

```
mknod /dev/slot0 c 235 0
```

The above command creates a character device file `/dev/slot0` with major number 235 (i.e., a message slot) and minor number `0`. Then, a subsequent command creates another message slot file:

```
mknod /dev/slot1 c 235 1
```

2.1 Semantics of message slot file operations

The message slot driver has special semantics for the `ioctl`, `write`, and `read` file operations, as described below. Note that the description is given in the style of a manual page, and specifies the behavior from the perspective of the processes using the driver. One of your tasks is to figure out how to implement the module so that it provides the specified behavior.

2.1.1 `ioctl()`

A message slot supports two `ioctl` commands:

1. `MSG_SLOT_CHANNEL`: Takes a single `unsigned int` parameter that specifies a **non-zero** channel id. Invoking this `ioctl()` sets the file descriptor's channel id. Subsequent reads/writes on this file descriptor will receive/send messages on the specified channel.
2. `MSG_SLOT_SET_CEN`: Takes a single `unsigned int` parameter that specifies the censorship mode (0 for disabled, 1 for enabled). When censorship is enabled, messages written to the channel will be censored using a simple algorithm: every 3rd character in the message is replaced with '#'. The censorship state is maintained per file descriptor.

Error cases:

- If the passed command is not `MSG_SLOT_CHANNEL` or `MSG_SLOT_SET_CEN`, the `ioctl()` returns -1 and `errno` is set to `EINVAL`.
- If the passed channel id is 0 (for `MSG_SLOT_CHANNEL`), the `ioctl()` returns -1 and `errno` is set to `EINVAL`.

2.1.2 `write()`

Writes a non-empty message of up to 128 bytes from the user's buffer to the channel. Returns the number of bytes written, unless an error occurs. (Note that the message can contain any sequence of bytes, it is **not** necessarily a C string.)

Error cases:

- If no channel has been set on the file descriptor, returns -1 and `errno` is set to `EINVAL`.
- If the passed message length is 0 or more than 128, returns -1 and `errno` is set to `EMSGSIZE`.
- In any other error case (for example, failing to allocate memory), returns -1 and `errno` is set appropriately (you are free to choose the exact value).

Censorship: When censorship is enabled for the file descriptor (set via `MSG_SLOT_SET_CEN`), the message is censored before being stored in the channel. The censorship algorithm replaces every 3rd character (positions 2, 5, 8, etc., using 0-based indexing) with '#'. The censored message is what gets stored and will be returned by subsequent reads.

2.1.3 `read()`

Reads the last message written on the channel into the user's buffer. Returns the number of bytes read, unless an error occurs:

Error cases:

- If no channel has been set on the file descriptor, returns -1 and `errno` is set to `EINVAL`.
- If no message exists on the channel, returns -1 and `errno` is set to `EWOULDBLOCK`.
- If the provided buffer length is too small to hold the last message written on the channel, returns -1 and `errno` is set to `ENOSPC`.
- In any other error case (for example, failing to allocate memory), returns -1 and `errno` is set appropriately (you are free to choose the exact value).

Censorship: Reading from a channel always returns the stored message exactly as it was written. If a message was written with censorship enabled, read() will return the censored version (with '#' characters). If a message was written without censorship, read() will return the original message.

IMPORTANT: Message slot reads/write should be atomic: they should always read/write the entire passed message and not parts of it. So a successful `write()` always returns the number of bytes in the supplied message and a successful `read()` returns the number of bytes in the last message written on the channel.

3 Assignment description

Implement the following:

1. `message_slot`: A kernel module implementing the message slot IPC mechanism.
2. `message_sender`: A user space program to send a message.
3. `message_reader`: A user space program to read a message.

3.1 Message slot kernel module (device driver)

Implement the module in files named `message_slot.c` and `message_slot.h`:

1. The module should use the hard-coded major number **235**. (The proper way to implement a character device driver is to dynamically allocate a major number, as seen in the recitation, but we will use a hard-coded major number for simplicity.)
2. If module initialization fails, print an error message using `printk(KERN_ERR ...)`.
3. The module should implement the file operations needed to provide the message slot interface: `device_open`, `device_ioctl`, `device_read`, and `device_write`. Implement these operations any way you like, as long as the module provides the message slot interface specified above. You might find these suggestions useful:

- You'll need a data structure to describe individual message slots (device files with different minor numbers). In `device_open()`, the module can check if it has already created a data structure for the file being opened, and create one if not. You can get the opened file's minor number using the `iminor()` kernel function (applied to the `struct inode*` argument of `device_open()`).
- `device_ioctl()` needs to associate the passed channel id with the file descriptor it was invoked on. You can use the `void* private_data` field in the `file` structure parameter for this purpose. For example: `file->private_data = (void*) 3`. Check `<linux/fs.h>` for the details on `struct file`.

4. Bounds on number of messages channels and message slots:
   - For each message slot file, assume that no more than 2^{20} message channels will be used. This does **not** mean that the channel ids will be smaller than 2^{20}, just that you need to support at most 2^{20} different ids.
   - If you use `register_chrdev()` to register your device, you can assume that minor numbers are in the range $[0, 255]$ (i.e., there can be at most 256 different message slots device files). (This occurs because `register_chrdev()` limits the registered device to 256 minor number.) Otherwise, you can assume that minor numbers do not exceed 2^{20}, because Linux uses 20 bits to represent minor numbers.
5. In the module's `struct file_operations`, include the initialization
   `.owner = THIS_MODULE,`
   This will prevent the module from being unloaded while it is being used.
6. You need to define both `MSG_SLOT_CHANNEL` and `MSG_SLOT_SET_CEN` ioctl commands using the appropriate macros (e.g., `_IOW`).
7. The censorship state should be maintained per file descriptor (you can store it in `file->private_data` along with the channel id).
8. Censorship only affects write operations. Read operations always return the stored message as-is, regardless of the current censorship setting of the file descriptor.
9. You are responsible for defining the driver's `ioctl` command, as shown in the recitation.
10. Allocate memory using `kmalloc()` with `GFP_KERNEL` flag. (It is declared in `<linux/slab.h>`.)
11. While your module runs, the total amount of memory allocated by it should be $O(C \cdot M + N)$, where C is the total number of channels that were used (across all device files), M is the size of the largest message held and N is the number of times `MSG_SLOT_SET_CEN` ioctl command is called.
12. When unloaded, the module should free all memory that it allocated.
13. Remember that processes aren't trusted. Verify arguments to file operations and return -1 with errno set to `EINVAL` if the arguments are invalid. In particular, check the validity of user space buffers.
14. You can assume that any invocation of the module's operations (including loading/unloading) will run alone; i.e., there will not be concurrent system call invocations. This **does not** mean that there can't be several processes that have the same message slot open or using the same channel; it just means that they won't access the device concurrently.

3.2 Message sender

Implement the program in a file named `message_sender.c`.

Command line arguments:

- `argv[1]`: message slot file path.
- `argv[2]`: the target message channel id. Assume a non-negative integer.
- `argv[3]`: censorship mode (0 for disabled, 1 for enabled).
- `argv[4]`: the message to pass.

You should validate that the correct number of command line arguments is passed.

The flow:

1. Open the specified message slot device file.
2. Set the censorship mode to the value specified on the command line (0 to disable censorship, 1 to enable censorship).
3. Set the channel id to the id specified on the command line.
4. Write the specified message to the message slot file. Don't include the terminating null character of the C string as part of the message.
5. Close the device.
6. Exit the program with exit value 0.

If an error occurs in any of the above steps, print an appropriate error message (using `strerror()` or `perror()`) and exit the program with exit value 1.

3.3 Message reader

Implement the program in a file named `message_reader.c`.

Command line arguments:

- `argv[1]`: message slot file path.
- `argv[2]`: the target message channel id. Assume a non-negative integer.

You should validate that the correct number of command line arguments is passed.

The flow:

1. Open the specified message slot device file.
2. Set the channel id to the id specified on the command line.
3. Read a message from the message slot file to a buffer.
4. Close the device.

5. Print the message to standard output (using the `write()` system call). Print **only** the message, without **any** additional text.

6. Exit the program with exit value 0.

If an error occurs in any of the above steps, print an appropriate error message (using `strerror()` or `perror()`) and exit the program with exit value 1.

3.4 Example session

1. As root (e.g., with `sudo`): Load (`insmod`) the `message_slot.ko` module.

2. As root: Create a message slot file using `mknod`.

3. As root: Change the message slot file's permissions to make it readable and writable by your user.

4. Invoke `message_sender` to send a message on some channel without censorship.

5. Invoke `message_reader` to read the message on the same channel.

6. Invoke `message_sender` to send a message on some channel with censorship.

7. Invoke `message_reader` to read the message on the same channel.

8. Execute steps #4 to #7 several times, for different channels, in different sequences.

4 General guidelines

1. Because your message sender takes the messages from the command line, it will only be able to send a C string as a message. This does **not** mean that the kernel module or message reader should only work with C string messages. The kernel module provides the general interface specified in Section 2, and any program—not necessarily yours—can use it.

2. There's no requirement for the message sender/reader to exit "cleanly" on error. These programs may terminate without freeing memory and closing file descriptors.

5 Submission instructions

1. Submit a ZIP file containing five files: `message_slot.c`, `message_slot.h`, `message_sender.c`, `message_reader.c`, and a `Makefile` that builds the module. The ZIP file should be named `ex3_XXXXXXXXX.zip`, where `XXXXXXXXX` is your ID #.

2. The message sender and reader programs must compile cleanly on the course VM (no errors or warnings) when the following command is run in a directory containing the source code file and the `message_slot.h` file:

   `gcc -O3 -Wall -std=c11 message_sender.c` (or `message_reader.c`)