Operating Systems. Homework #3.

Submission – December 16, 23:55.

In this exercise, we implement a new mechanism for inter-process communication – *Message Slot*. A message slot is a character device file through which processes communicate using multiple *message channels*. Once a message slot device file is opened, 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 slot preserves a message until it is overwritten; so the same message can be read multiple times.

Message Slot Specification

Message slots are implemented with a character device driver. Each message slot is a character device file managed by this driver. In general, different device files managed by the same driver are distinguished by a *minor* number. (The major number tells the kernel which device driver to use, and the minor number is used by the driver to distinguish different files it manages.) The minor number is specified as the second argument to `mknod`. Therefore, message slots are created using the `mknod` command, for example: `mknod /dev/myslot c M 0` (where M is the driver's major number) creates a message slot, and a subsequent `mknod /dev/fooslot c M 42` creates a different one.

This following specifies the semantics of file operations on a message slot file.

- `ioctl()`: A message slot supports a single `MSG_SLOT_CHANNEL` command. This command takes a single `unsigned int` parameter that specifies a channel id. Invoking the ioctl sets the file descriptor's channel id. Subsequent reads/writes on this file descriptor will receive/send messages on the specified channel.
    - If the passed command is not `MSG_SLOT_CHANNEL`, returns -1 and sets `errno` to `EINVAL`.
- `write()`: Writes a message to the channel. A message is a string, at most 128 bytes long.
    - If no channel has been set, returns -1 and sets `errno` to `EINVAL`.
    - If the length of the message is more than 128 bytes, returns -1 and sets `errno` to `EINVAL`.
- `read()`: Returns the last message written on the channel.
    - If no channel has been set, returns -1 and sets `errno` to `EINVAL`.
    - If no message exists on the channel, returns -1 and sets `errno` to `EWOULDBLOCK`.
    - If the provided buffer is too small to hold the message, returns -1 and sets `errno` to `ENOSPC`.
- Messsage slot reads/write are atomic: they read/write the entire passed message and not parts it. So, for example, a `write()` always returns the number of bytes in the supplied message (unless an error occurred).

Deliverables

You need to implement the following:

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

4. **Message Slot Kernel Module (Device Driver)**

- When loaded, the module should acquire a major number and print it as follows:

```
printk(KERN_INFO "message_slot: registered major number %d\n", majorNumber);
```

- The module should implement the file operations needed to provide the message slot interface: `device_open`, `device_ioctl`, `device_read`, `device_write`, and `device_release`. 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 (different device files). In `device_open()`, the module can check if it has already created such a data structure for this file, 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 set 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`.
- You are responsible for defining the driver's ioctl command, as shown in the recitation. You may use whatever major number works in your system; just make sure to hard-code it into the kernel module.
- General requirements:
  - Allocate memory using `kmalloc` with `GFP_KERNEL` flag.
  - When unloaded, the module should free all memory that it allocated.
  - Remember that processes aren't trusted. Verify arguments to file operations and return -1 with `errno` set to `EINVAL` if they are invalid. In particular, check the provided user space buffers.
- There's no need to handle concurrency. You can assume that any invocation of the module's operations (including loading and unloading) will run alone. This does **not** mean that there can't be several processes that have the same message slot open or use the same channel; it just means that they won't access the device concurrently.

Message Sender

- Command line arguments:
  1. `argv[1]` – message slot file path
  2. `argv[2]` – the target message channel id. Assume an integer.
  3. `argv[3]` – the message to pass. Assume maximal string length be 128.
- The flow:
  1. Open the specified message slot device file.
  2. Set the channel id to the id specified on the command line.
  3. Write the specified message to the file.
  4. Close the device.
  5. Print a status message.
- Exit value should be 0 on success and a non-zero value on error.
- Should compile without warnings or errors using `gcc -O3 -std=c99`.

Message Reader

- Command line argument:
    1. `argv[1]` – message slot file path
    2. `argv[2]` – the target message channel id. Assume an integer.
- 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 device to a buffer.
    4. Close the device.
    5. Print the message and a status message.
- Exit value should be 0 on success and a non-zero value on error.
- Should compile without warnings or errors using `gcc -O3 -std=c99`.

Example Session

1. Load (`insmod`) the `message_slot.ko` module.
2. Check the system log for the acquired major number.
3. Create a message slot file managed by `message_slot.ko` using `mknod`.
4. Invoke `message_sender` to send a message.
5. Invoke `message_reader` to receive the message.
6. Execute steps #4 and #5 several times, for different channels, in different sequences.

Submission

Submit five files: `message_slot.c`, `message_slot.h`, `message_sender.c`, `message_reader.c` and a `Makefile` that builds the module. Submit the files in a ZIP file named `ex3_012345678.zip`, where `012345678` is your ID.

Mac users! Don't submit hidden folders!