# Operating Systems. Homework #3.

Submission – May 14, 23:59.

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*. A message slot device can have multiple channels active concurrently, which can be used by different processes.

Once a message slot device file is opened, a process uses `ioctl()` to either specify the id of the message channel it wants to use, or to change the write mode (append/overwrite). 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 by a write using the 'overwrite' write mode**; so the same message can be read multiple times.

**Cheating Policy**

Part of the course requirements is to program assignments by yourself. You can discuss concepts and ideas with others, but looking at other people's code, looking at code from previous semesters, sharing your code with others, coding together, etc. are all not allowed. Students caught violating the requirement to program individually will receive a "250" grade in the course and will have to repeat it. *Even if you're under pressure, ask for help or even don't submit, rather than cheat and risk having to repeat the course.*

If you put your work in your TAU home directory, set file and directory permissions to be accessible only by you.

**Late Submission Policy**

You have 5 grace days throughout the semester, you can use all of them, some of them or none of them for this exercise. A 10 point deduction will be applied for every late day past the grace days.

## Message Slot Specification

Message slots are implemented with a character device driver. Each message slot is a character device file managed by this driver. 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 message slot.

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

- `ioctl()`: A message slot supports two commands:
  - The first command is `MSG_SLOT_CHANNEL`. This command takes a single `unsigned int` parameter that specifies a non-zero channel id. Invoking the ioctl with this command 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 channel id is 0, returns -1 and sets `errno` to `EINVAL`.
  - The second command is `MSG_SLOT_WRITE_MODE`. This command takes a single `unsigned int` parameter that specifies 0 (overwrite mode) or 1 (append mode). Invoking the ioctl with this command sets the device file write mode. Subsequent writes on this device file (on any channel) will be performed according to the last set write mode.
    - The default write mode of a device is 0 (overwrite mode).
    - If the passed write mode is different than 0 or 1 , returns -1 and sets `errno` to `EINVAL`.
  - If the passed command is not `MSG_SLOT_CHANNEL` or `MSG_SLOT_WRITE_MODE`, returns -1 and sets `errno` to `EINVAL`.
- `write()`: Writes a non-empty message from the user's buffer to the channel according to the write mode. If the write mode is set to 0 (overwrite mode), the new message will replace the old message in the channel (if such). Otherwise, the new message will be appended to the previous content in the channel. The overall size of the message (in both write modes) can be up to 128 bytes. `write()` returns the number of bytes written, unless an error occurs:
  - If no channel has been set, returns -1 and sets `errno` to `EINVAL`.
  - In 'overwrite' mode: If the passed message length is 0 or more than 128 bytes, returns -1 and sets `errno` to `EMSGSIZE`.
  - In 'append' mode: If the passed message length is 0 or the total message size (original message + new message) is more than 128 bytes, returns -1 and sets `errno` to `EMSGSIZE`.
  - In 'append' mode: if no previous message exists, the message should be written at the beginning of the channel.
  - In any other error case (for example, failing to allocate memory), returns -1 and sets `errno` appropriately (you are free to choose the exact value).
  - In case of a write error, the previous message state is not guaranteed.
- `read()`*:* Reads the last message written on the channel into the user's buffer, and returns the number of bytes read, unless an error occurs:
  - 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`.
  - In any other error case (for example, failing to allocate memory), returns -1 and sets `errno` appropriately (you are free to choose the exact value).
- Message 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_sende*r: user space program to send a message.
3. *message_reader*: user space program to read a message.

**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);
```

- If module initialization fails, print an error message using `printk(KERN_ERR …)`.
- 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 (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 commands, as shown in the recitation.
- 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 write mode. Assume 0 or 1.
  3. `argv[3]` – the target message channel id. Assume a non-negative integer.
  4. `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 write mode of the device.
  3. Set the channel id to the id specified on the command line.
  4. Write the specified message to the file.
  5. Close the device.
  6. 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 -Wall -std=gnu99`.

**Message Reader**

- Command line argument:
  1. `argv[1]` – message slot file path.
  2. `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 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 -Wall -std=gnu99`.

## 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. Change the message slot file's permissions to make it readable and writable.
5. Invoke `message_sender` to send a message.

6. Invoke `message_reader` to receive the message.
7. Execute steps #5 and #6 several times, for different channels, with different write modes in different sequences.

## Submission Guidelines

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.