Operating Systems. Homework #5.

Submission - July 10, 23:55.

In this exercise, we implement a new mechanism for inter process communication – Message Slot. We develop a kernel module that accepts and stores messages in its internal buffer. In contrast to pipes, the module preserves a message until it is overwritten. Also, we provide a support for several Message Channels, so several messages can be preserved simultaneously. Four channels are supported in our implementation. Three programs shall be written: message_slot - a kernel module (similar to CHARDEV example shown in the class), and two user space programs - message_sender, message_reader.

The flow:

1. Load (insmod) the message_slot.ko, check the system log for the acquired Major Number.
2. Define a virtual device managed by message_slot.ko. (mknod)
3. Invoke message_sender to send a message.
4. Invoke message_reader to receive the message.
5. Execute steps #3 and #4 several times, for different channels, in different sequences.

Kernel module

The kernel module provides 5 file operations: device_open, device_ioctl, device_read, device_write, device_release. It also defines a custom data structure – Message Slot, which contains 4 internal buffers, each of 128 bytes long, and the index of the current channel. An instance of this data structure contains the last messages written to the channel, on a per device file basis. The module supports one ioctl command – set current channel index.

Upon open
Check if the file has an associated Message Slot data structure.
If not, then allocate Message Slot data structure (kmalloc, with GFP_KERNEL flag), and preserve it in kernel module. Hint: Use `file->f_inode->i_ino`, as a unique ID of a device file.
Initialize/allocate any other variables you need.

Upon module removal
Free the allocated memory.

Upon ioctl
Check that the correct command was received. The argument is the Message Channel index. Check that the index is in [0,3]. Return an error code if one of the tests fails. If all tests succeed then store the Message Channel index for this open file (different open files may have different Message Channel indices, at the same time) in the relevant Message Slot data structure.

For both read and write requests
Check the provided user space buffer. Return an error code if the Message Channel index is not set. If both tests succeed then proceed with read/write. If length is more than 128, ignore extra bytes.

Upon write
If both preliminary tests succeed then copy the bytes from the user buffer to the relevant internal kernel buffer. If the message content is less than 128, then fill the rest of the kernel buffer with zeros.

Upon read
If both preliminary tests succeeded then copy the number of requested bytes, from the kernel buffer to the user buffer, according to the number of the requested bytes by the user.

Message Sender

Command line arguments:

1. argv[1] – the target Message Channel index. Assume an integer.
2. argv[2] – the message to pass.

The flow:

1. Open the device managed by message_slot kernel module.
2. Invoke ioctl() on the device with the index provided.
3. Prepare a buffer with the message to pass.
4. Write the buffer to the device.
5. Close the device.
6. Print a status message.

Message Reader

Command line argument:

1. argv[1] – the source Message Channel index. Assume an integer.

The flow:

1. Open the device managed by message_slot kernel module.
2. Invoke ioctl() on the device with the index provided.
3. Prepare a buffer.
4. Read the message from the device to the buffer.
5. Close the device.
6. Print the message and a status message.

Submission

Submit a zip file named ex5_012345678.zip, where 012345678 is your ID. The archive contains 5 files: `message_slot.c`, `message_slot.h`, `Makefile`, `message_sender.c`, `message_reader.c`. *Mac users! Don't submit hidden folders!*