A Linux `Mail Slot` device implementation

Laura Trivelloni

Course of Multi/Many-core Programming

Abstract

This report describes the implementation of a special device file that is accessible according to FIFO style semantic, posting or delivering data units atomically and in data separation, and that is configurable via *ioctl* interface.

1. Introduction

This report describes the implementation of a special device file that is accessible according to FIFO style semantic (via *open/close/read/write* services), but offering an execution semantic of *read/write* services such that any segment that is posted to the stream associated with the file is seen as an independent data unit (a message), thus being posted and delivered atomically (all or nothing) and in data separation (with respect to other segments) to the reading threads. The device file is multi-instance (by having the possibility to manage at least 256 different instances, as default configuration) so that mutiple FIFO style streams (characterized by the above semantic) can be concurrently accessed by active processes/threads.

The device file supports ioctl commands in order to define the run time behavior of any I/O session targeting it (such as whether read and/or write operations on a session need to be performed according to blocking or non-blocking rules, change driver parameters).

The device file described above is called **mailslot**.

A mailslot device manages independent data units. This is what distiguishes it from a standard `mkfifo` device, from which bytes can be read among multiple reads. At the end of this report will be showed a performance comparison between these two device types.

2. Implementation

2.1. Configuration

- maximum multi-instance level **DEVICE_FILE_SIZE** (default 256 instances)
- **policy** to define rules for I/O session (default non-blocking policy)

Email address: `laura.trivelloni@gmail.com` (Laura Trivelloni)

- maximum mailslot storage **MAXIMUM_SLOT_STORAGE** (default 256*100 bytes)
- absolute upper limit up to is possible to set **MAXIMUM_SLOT_STORAGE**
- maximum message size per data unit **MAXIMUM_MESSAGE_SIZE** (default 256 bytes)
- minimum for minor number **MINOR_MIN** (default 0)

2.2. Data unit representation

A **data unit** is a dynamically allocated message node (`maislot_msg_t struct`) described by:

- pointer to the next data unit
- message content
- message size

2.3. Mailslot representation

A **mailslot** is a statically allocated array of 256 (or maximum number of instances specified), where each entry is a (`maislot_t struct`) composed by:

- pointer to the head of the queue
- pointer to the tail of the queue
- current global size as sum of every message currently in the mailslot
- semaphore to make insert and delete operations on the mailslot atomic
- conditional wait queue for sleeping threads due to empty mailslot status
- conditional wait queue for sleeping threads due to full mailslot status

2.4. Supported operations

- **read**: get standing message, only a data unit for each request. It fails if user requests less bytes than standing message size
  - non-blocking policy: if empty mailslot, return with 0 read byte
  - blocking policy: if empty mailslot, request added to a queue waiting for a new message available
- **write**: append to the mailslot a new message. It fails if message size is more than `MAXIMUM_MESSAGE_SIZE` specified.
  - non-blocking policy: if mailslot is full, return with failure

– blocking policy: if mailslot is full, request added to a queue waiting for space where writing the new message available

- **open**: open device with a minor included between 0 and 255. It fails if it is specified an out of range minor.
- **release**: close device
- **ioctl**: interface for status and configuration control.
  – command `0`: change policy to blocking/non-blocking one
  – command `1`: change maximum message size
  – command *2*: change maximum mailslot storage tunable up to an absolute upper limit

2.5. Writers/Readers synchronization

The synchronization among threads is managed only among that ones that accesses to the same instance of the device, indexed by its minor number.

- after that a writer has written a new message, wake up all readers waiting for data unit availability
- after that a reader has read the standing message, wake up all writers waiting for storage availability
- after that a reader has failed a read request because of non-compliance with pending message size (too short read buffer), wake up all other readers waiting for data unit availability

3. Performance comparison: `mkfifo/mailslot`

In Figure 1 is shown the performance comparison between read and write operations for `mkfifo` and `mailslot`.

The times are measured through command line program `time`, getting the system time value.

The execution of this benchmark is pinned on a single 64 bits Intel(R) Core(TM) i7-3630QM CPU @ 2.40GHz with 4 cores.

The resulting times are computed as average among 4.000.000 operations of read and write, called in a concurrent way: 4 running threads (one per CPU-core) each executing 1.000.000 operations.

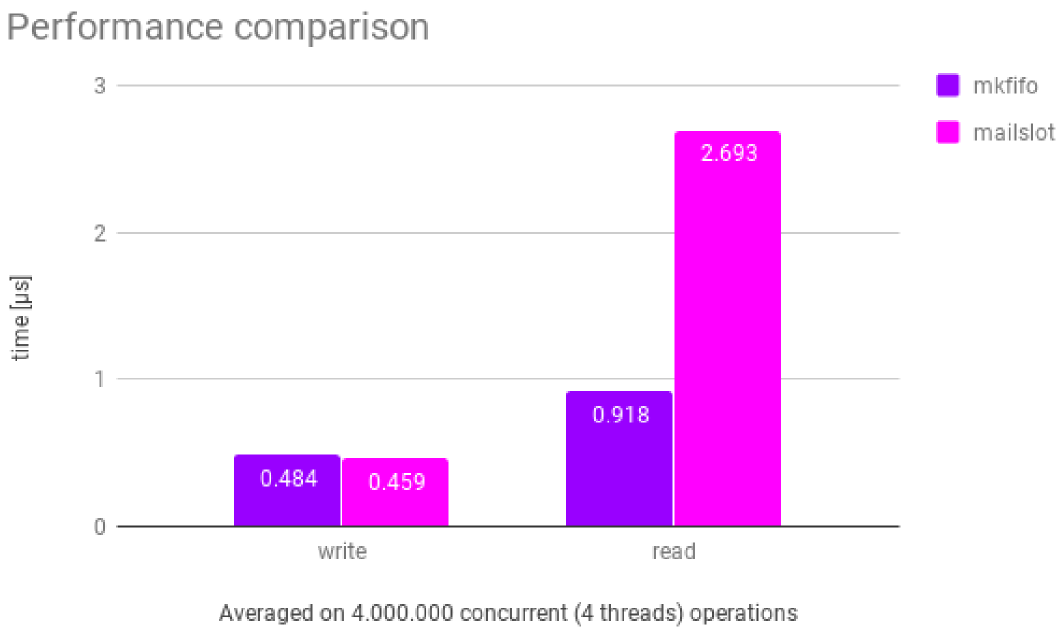

Figure 1: Averaged read and write execution times (μs) with 4 concurrent running threads