Moroso IPC Specification

Amanda M. Watson, Carnegie Mellon University

December 16, 2014

This document defines the behavior of the IPC mechanism for the Moroso Architecture.

1 Introduction

Like many other microkernels, the Moroso userland uses Interprocess Communication (IPC) to enable communication across address spaces. In the Moroso operating system, IPC is used for the following:

Requests from Servers In Moroso, programs can request resources or services from application servers: this might be the status of a file, several pages of data, or a request from another program to relinquish a resource to them. With this in mind, IPC should allow threads to exchange both messages (statuses, requests) and resources (large amounts of memory), and make it possible for a requesting thread to block until the request or resource is complete – a requesting thread should be aware of when they alone have a resource without any ambiguity of ownership.

Synchronization The Moroso userland should provide developers with core thread synchronization primitives, such as `yield(), sleep,` and some approximation of Pebbles' `deschedule()/make_runnable()`. IPC should make it possible for these primitives to make guarantees about atomicity and synchronization between threads.

Signaling With many device drivers in userland, IPC should make it possible to forward interrupts and resources.

Performance While research suggests that IPC in modern microkernels is not the bottleneck it once was, as IPC is a core service, performance remains an important consideration. While its not clear yet how good performance should be defined, designs should attempt to not hinder it, optimizing for common cases as well as remaining configurable for the wide variety of workloads it might support.

Security IPC by nature allows one process to influence another, and has historically presented many secuirty vulnerabilities; a bad IPC system could allows one program to cause another to crash, run out of memory, or be blocked forever. Thus, ensuring that IPC enables communication without infringement of resources is paramount to our design.

While the userland may not have all of these features in place, these use cases represent the spirit of IPC, and the design is made with these in mind.

1.1 Blocking on Messages

In Moroso IPC, there are two different protocols for blocking on message exchange:

Synchronous When a message is sent synchronously, the sender blocks until a receiver either sends a reply or the request times out. A synchronous receive (note: all receives are synchronous in Moroso IPC) blocks until a sender completes a message transfer, or the request times out.

Asynchronous When a message is sent asynchronously, the sender is not blocked on a reply: instead, if no receiver is waiting, it returns as soon as the message is made available to the receiver: otherwise, it performs the transfer directly, but returns without a reply.

1.2 Types of Messages

To enable the exchange of both information and memory resources, IPC can send two different types of messages: long and short. Since each has different abilities and limitations, they are dealt with differently.

Long Message A long IPC message is a message that is larger than one page. To avoid the overhead of making copies of so much data or creating additional storage, long messages are specified by the page and transferred directly between virtual memory spaces: as such, all long messages are transferred synchronously.

Short Message a short IPC message is a message smaller than a page in size – this is what might be used for an exchange of requests, acknowledgements, or other short status information. As short messages are small enough to be copied into the kernel, they can be sent synchronously or asynchronously. Note that performance is less of a concern with asynchronous sends, and will almost certainly be slower.

Descriptor Descriptors are unsigned integer values that map to mailboxes; in order to send to a mailbox, a process needs an associated descriptor. A process can receive a descriptor by being "forwarded one" via IPC. When one process sends a descriptor, the kernel creates an entry for the mailbox it maps to in the recipient's descriptor table, and sends the recipiant the new descriptor. Note that, in order to forward a descriptor, the receiver must be denote that they are open to receiving a new mailbox.

1.3 Additional Notes:

"Mailboxes" All IPC messages are passed through mailboxes. Mailboxes are created by threads and interacted with through "descriptors", an unsigned integer that maps to a mailbox kernelside for all threads in the process. For more information, see "The Moroso Userland", or the mailbox interface below.

Timeouts It's important to note that synchronous sends and receives have timeouts associated with them – timeouts can range from 0 (essentially polling) to infinite (no timeout). When the timeout is reached, the message expires, and the invoking thread returns `IPC_ERR_TIMEOUT`.

2 Interface

2.1 IPC

`ipc_send` and `ipc_recv` are passed their arguments through a struct called `ipc_args`. Its declaration is the following:

```
struct buffer_t {
    addr: *u8,
    len: u32,
}

struct ipc_args {
    dest_id: u32,
    long_src: buffer_t,
    long_dest: buffer_t,
    short_src: buffer_t,
    short_dest: buffer_t,
    desc_src: i32,
    desc_dest: i32,
    timeout: u32,
    options: u32,
}
```

The interfaces for the IPC functions are the following:
`fn ipc_send(args: *ipc_args) -> i32`

Sends a long and/or short message to a thread receiving on the mailbox that maps to the descriptor `args->dest_id`. Messages are received in the order in which they are sent.

Parameters:
`args->dest_id`: the mailbox descriptor being sent to. function returns `IPC_ERR_DEST` if `dest_id` is not a valid descriptor, and `IPC_ERR_DEAD` if the mailbox is destroyed while the invoking thread waits on it.

`args->long_src` specifies the long message to be passed to the recipiant. A maximum of `long_src->len` bytes will be copied to the recipiant. `long_src->addr` is ignored if `long_src->len` is 0. returns success (but with no copy) if pages are not mapped at the time of the copy. returns `IPC_ERR_ARGS` if `long_src.len` is not page aligned. Ignored if message is async. Returns `IPC_ERR_ARGS` if `args->long_src.addr` does not correspond to a userland address, but `args->long_src.len` is nonzero; however, no error is thrown if a long message is not mapped at the time of transfer; the message is merely truncated. Returns `IPC_ERR_ARGS` is `args->long_src.addr` is not page-aligned, or `args->long_src.len` is not a multiple of `PAGE_SIZE`

`args->long_dest`: specifies a buffer for a long reply from receiver. `long_dest.len` gets altered to be the final length of the message copied. ignored if message is async or doesn't expect a reply. Returns `IPC_ERR_ARGS` if `args->long_dest.addr` does not correspond to a userland address, but `args->long_dest.len` is nonzero; however, no error is thrown if a long message is not mapped at the time of transfer; the message is merely truncated. Returns `IPC_ERR_ARGS` is `args->long_dest.addr` is not page-aligned, or `args->long_dest.len` is not a multiple of `PAGE_SIZE`

`args->short_src`: specifies a short message to be passed to the recipiant. A maximum of `args->short_src.len` bytes will be copied to the recipiant. `args->short_src.addr` is ignored if `args->short_src.len` is 0. returns `ERR_ARGS` if `args->short_src.len` is geq `PAGE_SIZE`. returns `IPC_ERR_MAP` if message is not mapped into memory. if `args.short_src.len` is larger than `SHORT_BUFFER_MAX`, it is placed in a dynamically allocated buffer; returns `IPC_ERR_INTERNAL` if buffer allocation fails. Note that this is the only way a valid synchronous transfer can fail due to memory errors. Returns `IPC_ERR_ARGS` if `args->short_dest.addr` does not correspond to a userland address, but `args->shot_dest.len` is nonzero
`args->short_dest`: specifies a buffer for a short reply from receiver. Is ignored if send doesn't expect a reply. A maximum of `short_dest.len` bytes is copied; `short_dest.len` is altered to specify number of bytes copied to `short_dest.addr`. Returns `IPC_ERR_MAP` if specifed message is not correctly mapped into memory by the time the transfer occurs.

`args->desc_src`: descriptor being sent to the receiver. Ignored if `args->desc_src` is a value lest than zero. returns `MB_ERR_DESC` if receiver was unable to transfer a descriptor into the table (most likely due to running out of space)

`args->desc_dest`: buffer for descriptor from receiver. If `args->desc_dest` is positive, receiver can place a mailbox in the invoking thread's descriptor table and pass the resultant descriptor to `args->desc_dest`. if `args->desc_dest` is negative, no descriptor transfer occurs. This is done so that the sender if able to consent to whether a new mailbox is added to its table

`args->timeout`: how many clock cycles a message should wait for receipt/reply until it expires. message is unable to timeout while being modified by `ipc_recv` or `ipc_reply`. a timeout of -1 means the message does not expire. Returns `IPC_ERR_TIMEOUT` if timeout occurs before successful transfer.

`args->options`: specifies what kind of message and whether long message is shared

Return Value:
`ipc_send()` returns 0 on success, a negative error code on error. `args->short_dest` and `args->long_dest` are altered to denote how much was copied to the specified buffers. However, no indication is given a to how much of `args->short_dest` and `args->long_dest` were copied to the destination if the transfer was successful. Note that message transfers might still be successful, even if a negative error code is returned

`fn ipc_recv(args: *ipc_args) -> i32`
Blocks until it receives a long and/or short message from the next available sender on the mailbox that maps to `args->dest_id`, or the receive times out. Only one sender is received per invocation. Note that, when `ipc_recv` returns from a synchronous send, the sender will still be blocked until `ipc_reply` is called.

Parameters:
`args->dest_id`: the mailbox descriptor being received on. function returns `IPC_ERR_DEST` if `args->dest_id` is not a valid descriptor, `IPC_ERR_DEAD` if the mailbox is destroyed while the invoking thread waits on it, and `IPC_ERR_OWNER` if the invoking thread does not belong to the proc that owns the mailbox.

`args->long_src`: ignored

`args->long_dest`: specifies a buffer for a long reply from sender. `args->long_dest.len` gets altered to be the final length of the message copied. ignored if message is async or doesn't expect a reply. Returns `IPC_ERR_ARGS` if `args->long_dest.addr` does not correspond to a userland address, but `args->long_dest.len` is nonzero; however, no error is thrown if a long message is not mapped at the time of transfer; the message is merely truncated. Returns `IPC_ERR_ARGS` if `args->long_dest.addr` is not page-aligned, or `args->long_dest.len` is not a multiple of `PAGE_SIZE`

`args->short_src`: ignored

`args->short_dest`: specifies a buffer for a short reply from sender. A maximum of `args->short_dest.len` bytes is copied; `short_dest.len` is altered to specify number of bytes copied to `args->short_dest.addr`. Returns `IPC_ERR_MAP` if specifed message is not correctly mapped into memory by the time the transfer occurs.

`args->desc_src`: ignored

`args->desc_dest`: buffer for descriptor from sender. `args->desc_dest` is positive, sender can place a mailbox in the invoking thread's descriptor table and pass the resultant descriptor to `args->desc_dest`. if `args->desc_dest` is negative, no descriptor transfer occurs. This is done so that the receiver if able to consent to whether a new mailbox is added to its table

`args->timeout`: how many clock cycles a message should wait for a sender until it expires. message is unable to timeout while being modified by `ipc_send`. a timeout of -1 means the message does not expire. Returns `IPC_ERR_TIMEOUT` if timeout occurs before successful transfer.

`args->options`: ignored

Return Value:
Returns 0 on success, a negative error code on error. the dest fields in args are altered to denote how much was copied to the specified buffers. Note that message transfers might still be successful, even if a negative error code is returned. returns `IPC_ERR_REPLY` if invoked before last message received by the invoking thread with option `IPC_SYNC_REPLY` has timed out or been replied to.
`fn ipc_reply_all(message_long: *u8, len_long: u32, message_short: *u8, short_len: u32, desc: i32) -> i3`
reply: replies to the last sender received by invoking thread; returns with error if no active thread
Parameters:
`message_long`: address of long message to send to receiver. returns with success (but mapping fails) if address in range becomes unmapped

`long_len`: maximum length of long message transfer. `message_long` is ignored if `long_len` is 0.

`message_short`: address of short message to send to receiver.

`short_len`: maximum length of short message. `message_short` is ignored if `short_len` is 0.

`desc`: descriptor being sent to sender

Return Value:
Returns 0 if the sender was successfully replied to, a negative error code otherwise. An error is returned if the last synchronous sender has already been acknlowedged, or if the sender timed out and returned before `ipc_reply_all()` could be invoked.

2.2 Mailboxes

`fn mailbox_new() -> i32`
Creates a new mailbox. This mailbox is ”owned” by the process of the invoking thread, meaning only threads in the process are allowed to receives messages on it via `ipc_recv`.

Return Value:
On successful creation of the mailbox, returns a positive mailbox descriptor: the owner can then use this descriptor to send to the new mailbox, wait on it, or forward access to other processes. All children of the invoking process will have this value as the new mailbox's descriptor.

On error, returns a negative error code:

`MB_ERR_INTERNAL` (-8): insufficient memory to create mailbox/descriptor
`DESC_ERR_FULL` (-1): Not enough room in the descriptor table (other mailboxes will first have to be removed by calling `mailbox_remove()`, or moved by calling `mailbox_move()`)

`fn mailbox_move(old_desc: u32, new_desc: u32) -> i32`
Closes the descriptor `old_desc` and moves its mailbox to the descriptor `new_desc`; if `new_desc` already points to an active mailbox, its mailbox is removed and replaced by the mailbox at `old_desc`. The program can now communicate with the mailbox using the descriptor `new_desc`

Parameters:
`old_desc` the current descriptor of the mailbox being moved. if `mailbox_move` returns success, the program will no longer by able to communicate with the mailbox using `old_desc`. Returns `DESC_ERR_EMPTY`

if there is currently no mailbox at `old_desc`, and `ERR_ARGS` if the `old_desc` exeeds the descriptor table limit.

`new_desc` the new descriptor the mailbox is being moved to. If `mailbox_move` returns success, the program will now de able to communicate with the mailbox using `new_desc`. Returns `ERR_ARGS` if the `new_desc` exeeds the descriptor table limit.

Return Value:
On success, returns 0.

On error, returns a negative error code:
`DESC_ERR_EMPTY` (-4): `old_desc` referred to an empty descriptor
`DESC_ERR_ARGS` (-3): descriptor exceeded table limits.

`fn mailbox_remove(desc: u32)`
if `desc` is a valid descriptor, remove the mailbox it points to. If the invoking process is removing the last reference to the given mailbox, mailbox is destroyed

Parameters:
`desc` descriptor to mailbox being moved