Send/Receive/Reply#

Note that this document is under construction. At this point I have just started with the introduction and structure of the document. It's a work in progress, people! Give me a chance!

Introduction#

In this document I will describe the design and implementation of the MsgSend and related functions.

The messaging mechanism is **the** key to Neutrino performance: Almost every interaction that a user program has with the kernel or with other services happens through a message send/receive/reply.

For a functional description of the Neutrino messaging mechanism, refer to the QNX Neutrino IPC section of the System Architecture Guide.

Core Functions#

The core functions of the QNX Neutrino messaging mechanism are MsgSendv, MsgReceivev and MsgReplyv. Also of importance are MsgError, MsgReadv and MsgWritev.

Note that the MsgXxxxv functions are each used for the implementation of a set of public APIs.

For example, MsgSendv is used to implement the public functions MsgSend, MsgSend_r, MsgSendnc, MsgSendnc_r MsgSendsv, MsgSendsv_r, MsgSendsvnc, MsgSendsvnc_r, MsgSendv, MsgSendvnc, MsgSendvnc_r, MsgSendvs, MsgSendvs_r, MsgSendvsnc, and MsgSendvsnc_r. And I probably missed some.

All of these are referred to as the *MsgSend family* of functions. They are simply parameter variations of the MsgSendv function. In fact, there is only one system call (for MsgSendv) -- all the rest of the functions are implemented in the C library by massaging the parameters to match the requirements of MsgSendv and then invoking the MsgSendv system call.

Throughout the rest of this document, MsgXxxx will be used to refer to any or all of the MsgXxxx family of functions.

Synchronous Local Short Messages Only#

The QNX Neutrino messaging mechanism supports messaging to both local and remote threads through the same interface. As might be expected, the implementation differs somewhat between local and remote messaging. For the moment we will only talk about local messaging; remote messaging will be described in a later version of this document.

Further, in the initial version of this document we will only look closely at short messages. The message transfer mechanism for longer messages is quite complex, and we will only look at it form a high level at this point; long message transfer will be fully documented at some point in the not-too-distant future.

Finally, the asynchronous message passing mechanism shares some implementation with the synchronous messaging mechanism. We are only going to look at the synchronous message mechanism for now.

Let's Dive In...#

In a typical client/server model, the server has a thread waiting blocked in the MsgReceive function (state **RECEIVE_BLOCKED**) for requests from clients. The client sends a message using MsgSend and blocks waiting for a reply (state **REPLY_BLOCKED**). The server thread receives the message and begins to run. The server thread does whatever it needs to do with the message, and when it is done, sends a reply using MsgReply. The reply is given to the client thread which is allowed to run while the server thread goes back to the **RECEIVE_BLOCKED** state.

The key difficulty in implementing the Send/Receive/Reply mechanism is that the client and server threads are (usually) in different address spaces. This means that the data is the message and the reply have to be copied (a gut reaction to copying data in a message send is usually: **this is bad!** - every copy costs CPU cycles. But it's not completely a bad thing -- the alternative is to allow the server to mess around in the client's data structures, or vise versa, and that's dangerous). Copying data between address spaces is difficult.

As we'll see, the key complexity is the messaging mechanism comes in copying the data. In general, you will find the implementation of a system call names FooBar in a function called ker_foo_bar, usually implemented in services/system/ker/ker_xxxxx.c. This holds true here: we'll start by looking at ker_msg_receivev and ker_msg_sendv, which you can find in services/system/ker/ker_fastmsg.c.

I'm going to assume you're looking at the code the same time you're reading this, so best get your favourite code browser going or a lot of this isn't going to make sense.

MsgReceive#

The ker_msg_receivev function starts with validating the parameters. Refer to act and kap to help understand the system call parameters.

The chid (channel id) that is passed in is resolved to a kernel channel object, and if the receive info parameter is given it is validated for writability.

Next we test for special cases (flags & (_NTO_CHF_ASYNC | _NTO_CHF_GLOBAL)) which we'll ignore...

We validate the incoming IOVs. Here we see a trick used in supporting both IOVs and buffer/length parameter lists through one system call. Refer to IOVs and buffer length if you want to know how the rparts parameter can be less than zero.

We get into the real work when we do "thp = chp->send_queue;". The *chp* variable is the channel object we found earlier (by resolving the chid given to us as a parameter). The *send_queue* field of the channel is a pointer to the list of threads that are send-blocked on the channel. So, this statement gives us a pointer to the first send-blocked thread.

In our example, the server thread is invoking MsgReceive before any client sends any message, so thp will be NULL. So we skip a lot of source code (almost 200 lines -- we'll come back and look at it in another example), all the way down to a comment "No-one waiting for a msg so block". This code deals with the circumstance that the sender needs to go RECEIVE_BLOCKED.

We set the server thread's state to be RECEIVE_BLOCKED with a call to "unready(act, STATE_RECEIVE)", note that we're blocked on the channel, store our receive parameters in our thread control block and place thread control block on the channel's receive queue.

Refer to act->args for some information about the act->args field where we store the receive parameters.

MsgSend#

fixme: discuss MsgSend code where server is already waiting for message

Extract parameters from kap including rparts,rmsg,sparts,smsg to act's arg buffer.

If src message length <=_SHORT_MSG_LEN(current is 256) then we copy src message to kernel buffer located in act's control structure. Each thread has a kernel buffer to speed up the case message passing between different processes because we don't need to map dst process' receiving buffer to act's process address space and all processes sharing kernel space so we can quickly copy this buffer to dst's process address space after we switch to dst process address space.(In my opinion we can change this array to a pointer and extend spawn to make this buffer controlled by user and default is _SHORT_MSG_LEN, because CPU and memory various you can't say map or copy which is faster). Switch address space to dst process and call xfer_cpy_diov to transfer data.

If src message is > _SHORT_MSG_LEN then we call xfermsg to transfer data to dst thp(thp has been blocking on the channel's receive_queue already). In xfermsg we have to map dst thread's buffer to act thread's process using memmgr.map_xfer(normally it is vmm_map_xfer, some CPUs have optimization). Why not act's buffer to dst thread's process, we finally will switch to dst thread's process and it is synchronas message passing? Because the message is long we will have long time to map and copy and we are real time OS, in current OS design a long message passing can be interrupted so kernel has to switch back to act's process a couple of times then we waste much time.(Of course if CPU is strong then 1024 is not long, possible never be interrupted) Next we call xferiov to copy data.

__ker_exit#

fixme:

SEND_BLOCKED#

The server thread might not be waiting when the client sends the message. In that case, the client must block (state **SEND_BLOCKED**) until the server does a MsgReceive.

MsgSend#

fixme:

MsgReceive#

fixme:

Some Details#

In this section I talk about some of the details that are worth bringing up, but that get too much into the nitty-gritty when we're going over the algorithms.

act and kap#

All system calls take two parameters: act and kap. These are set up by the c library APIs and the kernel system call mechanism.

The act parameter ('act' for 'active') is a pointer to the thread object of the thread that made the system call. The act pointer is of type *THREAD* or *struct thread_entry* -- you can find it in services/system/public/kernel/objects.h. Thread object are heavily referenced throughout most of the system calls, so if you like reading the kernel documentation you're going to become familiar with them.

The kap parameter ('kap' for 'kernel args pointer') is a pointer to a structure that contains the parameters to the system call. The type of the kap pointer varies for different system calls. You can find them all defined in services/system/ker/kerargs.h.

IOVs and buffer/length#

In some of the variants in the function families, message buffers are defined through buffer/length pairs, while in others they are defined in IOVs (list of vectors).

In the IOV case, the parameters to the system call functions consist of a pointer to the IOV table (rmsg) and the number of entries in the table (rparts). In the buffer/length case, the parameters to the system call are given as the pointer to the buffer and the negative of the length of the buffer. The code can test whether the rparts parameter is greater or less than zero to determine what the pointer means.

Yes, we could have set up a single-entry IOV to handle the buffer/length case, but this is how we did it...

act->args#

The thread control block has a field called *args*. This field is a union of several different structure types. When we need to store some parameter information when we are about to block a thread, we store it here.

The act->args.ri structure is used when we're receive-blocked. The act->args.ms structure is used when we're send-blocked or when we're transferring a message from the sender to the receiver.

Obviously since act->args is a union, we need to be careful to make sure that we carefully control what fields we use in what states.

Leftovers#

There is some baggage in the Neutrino source code from previous versions of the messaging implementation. For the most part they are irrelevant to local messaging (they might still play a part in remote messaging), but they do show up in a couple of locations, so they're worth mentioning.

fixme: talk about _NTO_TF_SHORT_MSG and _NTO_TF_RCVINFO here.