xFS Message System Design

Doug Doucette

This document is the design for the message system module of IRIX to be used by xFS.

1.0 Introduction

A message system is provided for use by the filesystem implementation. The message system is used to communicate between certain modules of the implementation, especially over interfaces that will be remote in the future distributed implementation. We will also use the message system (along with the kernel threads implementation) if we can gain additional performance by doing so.

We expect to use messages from the vnode layer for remote files to the file system implementation vnode layers, for the distributed version of the filesystem. The volume managers in a distributed system will use messages to communicate with each other. The administrative interfaces are implemented using messages, so that they can be remote from the administered objects. The name manager needs messages to communicate with remote filesystems when crossing mount points. If remote devices are supported, then messages will need to be used in the specfs layer to implement that.

2.0 Overview

The main requirement is speed and simplicity, especially in the local case. We want local messages to be not much more costly to use than procedure calls, so that the mechanism will be used in the local case, and when the remote system is implemented, there will be less work to do.

The model is that there are three kinds of entities: messages, queues, and threads. Messages are delivered to (placed in) queues by threads. Messages are removed from queues by threads. We require that multiple threads can remove messages from a queue, so messages are not delivered to threads. We require that multiple threads can send messages to a single queue, as well. That is, message transmission is many-to-many.

The queue is a “place” for messages to wait when there are no message consumers, and for threads to wait when there are no message producers. It is a very simple data structure (pair of linked lists) with a “name”. For the moment we will assume that the name of a queue is a 128-bit unique, opaque identifier; these will be discussed in a separate paper.

What we have said so far implies an implementation for the local case, but even there a few questions remain. For instance:

- Is the position of a thread or a message in a queue affected by any priority scheme? Messages could have a priority associated with them, and threads certainly will.

- How does memory allocation work for messages: is the message system or its callers responsible for message allocation?

In the interests of simplicity we will make tentative answers for these questions, as follows:

- No priority scheme, isolate that to the thread mechanism. High-priority messages can be sent to separate queues and read by high-priority threads. A variable priority scheme which caused reading a message to set thread priority is possible but probably overkill for us.
- We will make the message system responsible for memory allocation for messages, and propose the details of the message format that make that practical.

For now, we will ignore the effect on the interfaces of making these messages work in a distributed system.

3.0 Operations

3.1 Message Queue Operations

The following queue operations are needed: create, destroy, and find. We will propose the following as interfaces (don't take the names too seriously):

- q = q_create(identifier);
- q_destroy(q);
- q = q_address(identifier);

Only one queue can exist per identifier, so **q_create** fails (returns NULL) if a queue already exists. **q_destroy** fails if the queue is active, or perhaps there is a force flag which sends the waiters and queued messages away. **q_address** translates from identifier to queue structure address (returning the value that **q_create** originally returned); it's not clear if we need this as an external interface. We may also need information operations to get back the identifier, message count, and thread count associated with a queue.

3.2 Message Operations

The following message operations are needed: allocate, free, send, receive. Also some setup operations may be supplied to initialize common portions of the message.

This is the basic, minimum set of operations. We should consider if there are additional operations we wish to specify that are useful beyond the minimal set. For instance, some form of send-and-wait-for-reply is useful.

Before we can specify interfaces, we need to know what common information is in the messages. We will assert the following here, and discuss it further below: message type, response identifier, sending thread, target identifier, and enough information to deduce the message components' sizes. We will give each message a type field which defines which structure type describes it; there is a reserved type value which indicates that the message is empty. Each message structure

will contain a message header structure at its start, containing all the common fields. The remainder of the message structure will contain fields specific to the type. Some fields may be pointers; the header must describe this so that the remote case will work.

We will propose the following interfaces (again, names have not been checked for conflicts):

- m = m_allocate(type);
- m_init(m, target id, response id);
- m_free(m);
- m_send(m);
- m2 = m_send_receive(m1);
- m = m_receive(receive id or q);

Messages are allocated with the target and response ids set to the null value. Allocation and initialization could be combined, but for now we'll make the primitives separate. **m_init** can be a simple macro, at least for the current information being set up.

The send and receive interfaces work solely with queue identifiers. While this can be made efficient, it will always be slower than using addresses. This can be repaired by adding "hint" fields to the messages, which contain the results of calling **q_address** on the identifier (one for response id, one for target id). We can also assume that it is actually ok for the **m_receive** call to take a message queue instead of a receive id as argument. We put the target id (and address hint) in the message instead of as arguments to **m_send**, since these will have to be in the remote messages anyway.

We need to decide whether a message receive call can be interrupted. It is clear we can add a non-blocking (polling) version of **m_receive**, but how do we abort a thread waiting for a message on a particular queue, and why would we be doing that? We could assert the existence of a **q_flush** interface, which blocked further messages from being sent to a queue and then sent "empty" messages to all waiters. This is probably sufficient, and we don't have to discuss thread semantics to describe it.

4.0 Allocation and Structures

4.1 Message Queue Allocation and Structures

The message queue structures are allocated from ordinary kernel memory by the interfaces described in Section 20.1 on page 21. Each queue structure includes the following:

- queue identifier
- pointer to head of list of message structures
- pointer to head of list of thread structures
- lock control information

- current and maximum length of message list
- current and maximum length of thread list
- if **q_flush** exists, a state flag for the queue
- a string naming the queue for debugging and statistics printing

Also needed are hashing or indexing structures for the translation of unique identifiers to queue structure addresses. For the moment we will assume a simple hashing scheme, and a sibling pointer in the queue structure to point to the next same-hash-value queue.

4.2 Message Allocation and Structures

As described in Section 20.2 on page 21, the messages are divided into a header (the same for all message types) and a body (different for all message types). The header includes the following:

- message type [small integer]
- target queue identifier [unique id]
- target queue address hint [queue structure pointer] (and target host identifier hint)
- source thread (and source host identifier), for debugging
- response queue identifier [unique id]
- response queue address hint [queue structure pointer] (and response host identifier hint)
- pointer to link the message on its queue

The message type implies the value of a couple of other pieces of information: the size and layout of the message body. We will assume that each body is composed of a data portion and a pointer portion. The size of the data portion is known based on the message type, as is the number of pointers. Question: do the pointers point to fixed or variable-sized data? If fixed, the pointer section is just pointers, and the length of data pointed to is known based on the message type. If variable, each length must be stored in the structure along with the pointer. Note that the size must be transmitted, in the remote case, for variable sizes; if the sizes are fixed then they can be asserted to be known on both sides.

Messages must be stored in global (not procedure-local) memory to avoid the necessity of message copying. To simplify this the **m_allocate** and **m_free** interfaces are supplied. The model is as follows:

A message is allocated with **m_allocate**, then initialized for sending with **m_init**. It is then sent to the destination queue with **m_send** (for instance). This links it onto the tail of the list of messages in the queue structure; the receiver removes the message from the head of the list of messages in the queue structure. The receiver examines and processes the message, and then calls **m_free** on it.

In the remote message-passing case, the operations are the same as far as the message producers and consumers are concerned. The pointer fields, if any, in the message, are transformed into arrays of bytes of data (with lengths). The message is freed on the sending side once it has been

successfully delivered (? we haven't discussed what kind of protocol this is ?). As the message is received a new message structure is allocated to hold it, and the new structure is put on the queue at that end. The arrays of bytes corresponding to the pointer section of the message are turned back into pointers to data on the receiving end. Since the original pointed-at objects are on the sending side, these are copies and can be allocated along with the other message data.

It is possible that the free operation will actually be a callback routine, defaulting to the behavior given above. This would allow more flexibility per subsystem on how message memory is allocated. The callback would have to be registered for the message type.

5.0 Distributed Message System Design

[To be supplied later.]