Message Passing

From OSDev Wiki

If you consider writing a Microkernel, you should layout how you'll manage the message passing. This page collects considerations on how you can do it.

Contents

- 1 Reliable vs Unreliable Messaging
- 2 Synchronous vs Asynchronous
- 3 The "Port" abstraction
- 4 Variable or Fixed Messages Size
- 5 See Also
  - 5.1 Articles
  - 5.2 Threads
  - 5.3 External Links

Reliable vs Unreliable Messaging

reliable

Messages are always delivered unless the recipient does not exist. On failure, the sender is notified. In order to provide reliability, the message passing system ensures that messages have not been altered or corrupted between transmission and reception. Out-of-order message blocks are reordered and duplicate message blocks are removed. If an error occurs, the system will attempt to retransmit the message automatically. Reliable messaging, however, usually comes with noticeable overhead.

unreliable

Messages may/may not be delivered to the recipient. If a message *is* delivered, its contents may be corrupted, out of order, or duplicated. Also, the sender may not receive any acknowledgment of a successful delivery. This may sound entirely useless, but unreliable messaging is typically simpler and faster than reliable messaging. By putting a reliable message protocol on top of unreliable messaging, reliable messaging can be achieved (very much like the TCP protocol[reliable] with IP packets[unreliable]). Such a protocol might utilize "ACK" messages and timeouts to support reliable messages.

Reliable messaging is used when data *must* be delivered correctly bit-by-bit (as in transferring a file). Unreliable messaging is used when data must be sent quickly or when it doesn't really matter if some messages are corrupted or lost (like periodic "I'm alive" messages or when streaming audio/video).

Synchronous vs Asynchronous

synchronous

Process A delivers the message directly to process B. If process B is not ready to receive at the moment of sending, Process A is suspended and queued upon Process B's 'sending' queue, until Process B is receiving. Using this method, you don't need to care for message queues - but you enqueue processes.

asynchronous
: Process A sends the message to Process B. The message is copied to a dedicated region in kernel space and attached to Process B's message queue (lets call it the Post box). The next time, Process B looks into its postbox, it will retrieve the message. You can have processes block for receiving a message or just stroll by and check whether there is something to fetch. One could call this method "store and forward message passing". The main benefit of asynchronous messaging is that it allows a single thread/process to wait for the results of many unrelated "requests" at the same time.

We may note that in the synchronous approach, A has the guarantee that B received the message when the deliver call terminates, which makes it especially interesting for local RPC. It should be noted that it can be very difficult to implement standard C library functions without this guarantee.

It isn't a bad idea to wrap some brains around the following problem: What, if Process A expects a message from Process C. It needs a possibility to communicate this to the message routing subsystem. If Process B sends a message to Process A, Process A shall not be woken up for it wants something from Process C. Process C sends a message to Process A. Then and only then shall the message routing subsystem wake up Process A.

What primitives will you need to get synchronous message passing running? Two: request(message) and respond(message).

What primitives will you need to get asynchronous message passing running? Two: send(message) and receive(message).

What primitives will you need to get both asynchronous and synchronous message passing running? Three: send(message), receive(message) and request(message), as the send(message) primitive is/can be the same as the respond(message) primitive.

It is possible to get synchronous message passing via asynchronous message passing primitives and vice-versa. Message forwarders along with acknowledgment messages and messaging protocols can make this possible. Unfortunately, doing it this way is not efficient.

The "Port" abstraction

Many microkernels use the notion of port, which is a 'point for receiving messages', or a 'one-way message communication channel'. The idea of ports is that you no longer send a message to a thread but to a port (which is linked to a unique thread).

Ports can be allocated by servers at will and may restrict who can send and who cannot send on it. Using ports, the 'A listen to C, B sends to A' problem above can be solved by creating a new port p that A will open to C only (and sending a message to C to tell p's port number). A can now wait for C's message by receiving on p only. B's attempt to send a message on port q of A will not interfere.

To solve situations when a server is willing to receive messages from different ports (and synchronous messaging is used), we need a select-like interface that will allow reception of a message from any port p in a given set of ports.

Variable or Fixed Messages Size

A message typically includes the ID of the emitting thread (or a replying port), a message code and a few arguments. Message queuing and dispatching code can be greatly simplified if message size is fixed.

When messages longer than the fixed-size are required, one will need to provide a way to describe the long message in the small structure. If copying a 4-word message forth and back doesn't imply excessive processing cost, it will be very different for the 1MB of data you send to a disk server. In that case, it is suggested to toy with paging in order to map the real data from the emitter to the receiver's address space.

See Also

- Remote Procedure Call (RPC)
- Message Passing Tutorial

Articles

Threads

- passing messages through registers
- RPC message size, handling oversized messages

External Links

- Message Passing on Wikipedia