NMQUEUE

Julian Schutsch

6. May, 2013

Contents

1 Introduction

This is to describe the implementation decisions for a multi sender and multi receiver queue for communication between threads in a shared memory machine. The second chapter describes the basic architecture, while the third chapter offers proofs for a number of issues. The last chapter will discuss measurements.

2 Design

The message queue is implemented as a bounded ring buffer queue with a single lock for both sending and receiving and two conditional variables for signaling changes in the ring buffer. As an overview the petri net used in the next section can be used, see 3.

2.1 Ring buffer

There is a choice between bounded and unbounded versions of a queue. The unbounded versions require a memory manager to allocate new messages while the bounded version must allocate a fixed amount at startup. The unbounded version has the advantage of a non blocking send, assuming enough memory is available.

The decision was against unbounded ring buffers because memory is usually limited. If the receiving threads cannot keep up with the sending threads, memory requirement will increase to infinity.

If sudden burts of messages are expected, but the amount can be handled by the receivers over time, either one has to use an unbounded version or a large enough buffer.

2.2 Two direction signals

With a bounded buffer, both sending and receiving threads can be blocked. To wake them up again, either signals or semaphores can be used. The decision goes against semaphores because,

- a possible limit smaller than the buffer size
- no reliable broadcast

For a proper shutdown, all threads need to be unblocked. They can then determine if they should terminate.

2.3 Single lock

On first glance a lock for reading and a lock for writing could be used in combination with volatile positions for reading and writing. But for safe use of signals, the lock of the opposite side must be hold.

Also a single lock avoids potential memory barrier issues, like changing the order of writting the message and incrementing the pointer.

3 Proof

Synchronisation issues cannot be tested (as in software test) for absence, since they may only happen in very rare cases. Therefore an attempt to proof correctness of the approach is presented using a petri network, shown in fig. 3 as starting point. The petri net is divided into three parts, where the top part shows a sending thread, the bottom part a receiving thread and the places in the middle are shared among all threads. For all proofs it must be acknowledged that there are more sending and receing threads.

Also there are a few necessary assumptions,

- Correct interpretation and implementation of the POSIX functions in the operating systems
- Ideal hardware, e.g. no random bit flipping, no fire
- Reasonable software conditions, e.g. no lack of resources

The system presented should neither dead lock, nor leave the queue in an inconsistent state. Therefore a number of questions are discussed which can cause either situation. A complete accessibility graph would go beyond the space available in this document.

3.1 Are sending and receiving exclusiv?

Inconsistency through race conditions can be avoided by locks. It must be shown, that the locks are freed upon completion and only held by one thread at the same time.

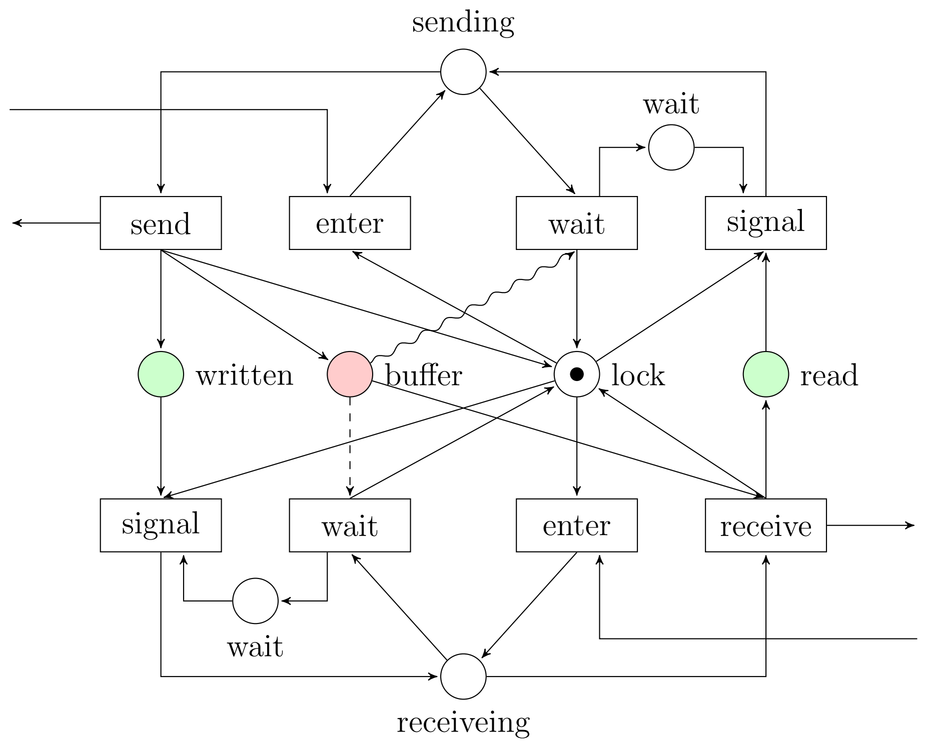

Figure 1: Petri network of the multi receiver and multi sender queue. Calls are received at the enter transitions and return after a successfull send or receive. The green places keeps many bullet as long as any target wait is active, but loose their bullets otherwise. The buffer node (red) can take a limited number of bullets (packages). The dashed arrow suppresses transition if the source place has any bullets. The snake arrow suppresses transition if the source place is not completely filled. Both special arrows do not consume bullets.

The situation is symmetric with only one lock bullet in the initial state. For entering either sender or receiver, the lock bullet is consumed. From this only two transitions are possible,

- Send/Receive
- Wait

which both give up the lock. If a wait state is active, the signal transition can only happen if the the lock is available. Therefore no two threads can enter sending or receiving at the same time.

3.2 Is it possible to be stuck in receive wait?

Receive wait can only be entered if the buffer is empty. Any send changing this will emit a "written" signal, which will not be lost since a receive wait state is active and cannot be lost without the lock which is only available once the signal is emitted.

3.3 Is it possible to be stuck in send wait?

Send wait can only be entered if the buffer is full. Any receve changing this will emit a "read" signal, which will not be lost since the send wait state is active. It cannot be lost for the same reason in receive wait.

3.4 Is it possible to be stuck in sending state?

Either the buffer is not full, then send is available. If the buffer is full, wait is available.

3.5 Is it possible to be stuck in receiving state?

Either the buffer is empty, then wait is available. If the buffer is not empty, receive is possible.

4 Measurements

Two measurements have been made. The first being the total time T required to send a number of messages. The second being the time required for a message to arrive at any receiving thread, called Δq. The average time required for each message is calculated by,

$$s = \frac{T}{N} \tag{1}$$

with N as the total number of messages send.

Still the number of possible combinations is very large with any number of sending and receiving threads and associated loads.

Here only two very simple setups are tested,

1. One sending thread, one receiving thread
2. Two sending threads, two receiving threads

where each sending thread sends 1 million messages without delay. For the second measurement, only the Δq for messages of send by the first thread are used.

For all measurements, `CLOCK_MONOTONIC` is used.

Results are shown in fig.4, fig.4 and fig.4.

Setup	T/s	$\Delta q/\text{s}$	s/s
1	3839609	968 ± 1356	3,8
2	6048373	1451 ± 1250	3,0

Table 1: VirtualBox 4.1.22 (4 cores assigned) running amd64 debian linux. Numbers behind $\pm$ are the standard deviates.

Setup	T/s	$\Delta q/\mathrm{s}$	s/s
1	9586533	1797 ± 2413	9,59
2	28430649	5219 ± 4508	14,22

Table 2: VMWare Player 5.0.2 (4 core assigned) running x86 QNX Neutrino 6.5.0. Numbers behind $\pm$ are the standard deviates.

Setup	T	$\Delta q/\mathrm{s}$	s/s
1	7757389	95 ± 536	7,76
2	14833631	620 ± 1988	7,42

Table 3: FreeBSD 9.1., Numbers behind $\pm$ are the standard deviates.

5 Conclusion

The implementation should be compared to different implementations to judge its speed and latency. This could include different signal approaches,

- Sending signals only once buffers are empty or full
- Comparison against unbounded message buffers
- Using Posix message passing
- Using Neutrino native message passing
- Comparison to pipes

Also results among the different plattforms vary, and they are not well explained. There should be a deeper analysis why there is a very short Δq for FreeBSD 9.1.

The implementation has been tested for correctness with `testdelivery`, it therefore should deliver the expected results.