

# An Efficient Wait-free Queue Implementation

Peter Seymour  
peteralanseymour@hotmail.com

November 19, 2007

## 1 Introduction

Interest in lock-free and wait-free algorithms for use in concurrent systems has recently been gaining in popularity. Lock-free is the property that at any time at least one process will be making progress (although no guarantees of fairness are assumed). Wait-free is the property that any process will complete in a finite amount of time. Most papers on concurrent algorithms focus on complex structures such as maps with multiple processes simultaneously accessing and updating the state. However, all these algorithms either use locks which are not only expensive to implement but lead to structures that are hard to reason about or employ some form of RCU (read-compare-update) which gives non-deterministic behaviour and potential starvation scenarios. Other problems arise such as memory management since without consensus between processes it's hard to determine whether memory is in use or not. These algorithms make the assumption that concurrency is implemented using threads and shared data structures, however, the actor model can handle this situation more elegantly. Each structure is represented as an actor receiving and sending messages asynchronously. To access a map, send it a message and wait for the response. To update a map, send it a message. This still requires a queue to handle the message passing implementation and it must work with concurrent processes but at least the model is becoming simpler. The consumer of messages in this queue is a single process, an actor, but there could be multiple producers. Again the above techniques are needed to handle the contention of the multiple producers. This can be avoided in numerous ways.

*Dedicated queues* Each actor pair has a dedicated message queue. However, this is only practical for a small fixed number of processes as it scales exponentially.

*Mailer process* Each actor communicates on a dedicated queue to a mailer process that then forwards the message on another dedicated queue to the consumer. This scales linearly but imposes a huge burden on the single mailer process.

*Tree approach* A tree of mailers would be able to forward messages in logarithmic time but only be subjected to linear increases in workload. A runtime of lightweight threads such as in Erlang does not suffer these problems as it is inherently a single path of execution application but communication between it and other instances or external devices can use dedicated pipes in a tree configuration.

The hybrid approach outlined above scales as well as can be expected by any scheme. Where a single path of execution runtime manages multiple processes no concurrency issues arise. At the medium level are dedicated queues until scaling problems occur. At higher levels logarithmic performance is accepted using a tree of mailer nodes. The higher level includes unbounded numbers of processes distributed globally as used by the Internet.

This all leads to a model that requires a very fast and simple queue that has a single producer process and a single consumer process. These requirements can be met using no more sophisticated primitives other than atomic reads and writes. It is shown in [1] that not even 2-consensus can be achieved with atomic registers. Consensus is roughly defined as peer processes all using the same algorithm coming to consensus on one of their input values in a finite amount of time. Is this a problem? Not really since a master process can be added to arbitrate decisions. In the real world the same problem occurs. If two people cannot agree then either no consensus is reached or a third-party must act. This is a natural way to structure software with a hierarchy of processes. Hopefully the case has been made that possibly the simplest of structures can solve the vast majority of concurrency issues when shared state is replaced by more elegant constructs. There will always be exceptions but the following implementation is so efficient it lends itself to wide usage.

## 2 The Implementation

This section details the implementation in a loose version of C++. The structure of the message is assumed to be of a fixed size but where that is a problem the message can be a pointer to a variable length message structure. Each message is contained in a `Packet` structure which also contains a possibly null pointer to another packet forming a linked list. Both the producer and consumer are modelled as agents which allows the common functions to be shared.

### 2.1 Agents

There are in fact two linked list between both agents. One is used by the producer to append new messages for the consumer to read and the other for the consumer to send them back for re-use. The producer is only allowed to create a fixed number of packets so relies on the consumer to send them back. An agent contains three pointers to packets and a counter:

```
Packet* _write;
```

```

Packet* _read;
const Packet* _end;
int _credit;

```

The `_write` pointer indicates the last packet on the outgoing queue of an agent. The `_read` pointer indicates the head of the incoming queue and `_end` points to the packet after the last element in the queue that can be safely read or used. The counter `_credit` keeps track of difference between the number of packets sent and received. The key functions are `scan`, `extract`, `insert` and `empty`.

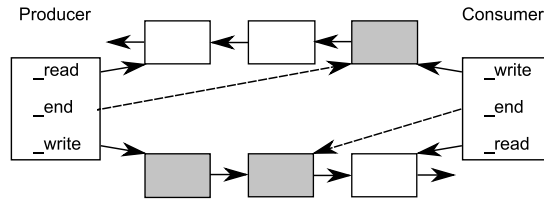


Figure 1: Producer and Consumer connected by two linked lists

The function `scan` advances the `_end` pointer over all the packets in the incoming queue until it settles on the final one which will have a null next pointer. This final one cannot be extracted as there must always be at least one packet in each queue. Initially both queues are given one packet to link the two agents together.

```

void scan()
{
    const Packet* pkt = atomic_read( &_end->next );

    while( pkt != 0 )
    {
        _end = pkt;

        smp_read_barrier_depends();

        pkt = atomic_read< const Packet* >( &_end->next );
        ++_credit
    }
}

```

The `atomic_read` function reads a value from the address specified. All mainstream architectures ensure that aligned reads of values no larger than a word are atomic making this a no-op. The function `smp_read_barrier_depends` is both a barrier that ensures the data which can be dereferenced from the packet pointer is read in order. It seems only the Alpha architecture does not do this by default but more information is given in [2]. The `smp_` functions are as used

in the Linux kernel [3]. They ensure firstly that the compiler will not move respective reads and writes over this barrier nor will the cpu reorder them across multiple processors. The `_credit` counter is incremented indicating the agent has one more packet in hand.

The function `extract` detaches a packet from the incoming queue. It should only be called if `_read != _end` as that indicates the packet has been scanned. Note that scanned packets are *safe* since the dependencies have been ensured.

```
Packet* extract()
{
    Packet* pkt = _read;
    _read = _read->next;
    --_credit;

    return pkt;
}
```

The function `insert` inserts a packet into the outgoing queue.

```
void insert( Packet* pkt )
{
    smp_write_barrier();

    atomic_write< Packet* >( &_write->next, pkt );
    _write = pkt;
}
```

The write barrier ensures that the content of the packet will appear correctly written prior to the packet being linked onto the list. Note that the barrier in `scan` ensures it is read correctly. Again writing the pointer should be an atomic operation. Once completed the packet is available for the other agent to read it. In effect the other agent now *owns* that packet.

Finally to detect if the incoming queue is empty is the function `empty`. If this returns false it is safe to call `extract`. It also contains a call to `scan` so that repeated polling on the queue will advance once new packets are available.

```
bool empty()
{
    scan();
    return _read == _end;
}
```

## 2.2 Producer and Consumer

Now that the common functionality is covered the final producer and consumer specialisations can be constructed. The producer is slightly more complicated as it must create new packets when none are free. It has a single `produce`

function that may fail and will return false. The only way it can fail is if it has used up all its available rights to create new packets. This is modelled with a counter `_overdraft` that is initially set with the maximum queue size. A greater allowance can be given to the producer allowing the queue to extend indefinitely.

```
bool produce( const Message& message )
{
    if( empty() )
        if( !create() )
            return false;

    Packet* pkt = extract();

    pkt->next = 0;
    pkt->message = message;

    insert( pkt );

    return true;
}
```

The function `create` generates a new packet and adds it to the list of scanned packets as follows:

```
bool create()
{
    if( _overdraft == 0 )
        return false;

    Packet* pkt = new Packet;
    pkt->next = _read;
    _read = pkt;

    --_overdraft;
    ++_credit;
    return true;
}
```

The process controlling the producer can check what proportion of the queue is in use and adjust its behaviour. This is achieved with a function `spare` indicating how many productions will succeed if at worst no more packets are returned. The producer could throttle its behaviour if this falls below a certain threshold enabling the producer to dynamically adjust its production rate for maximum flow.

```
size_t spare()
```

```

{
    scan();
    return _overdraft + _credit;
}

```

The consumer is simpler and enables the most recent packet to be looked at through `consume`. If none is available a null pointer is returned.

```

const Message* consume()
{
    if( empty() )
        return 0;

    Packet* pkt = extract();

    pkt->next = 0;

    insert( pkt );

    return &_read->message;
}

```

Notice that although the packet is extracted and placed on the outgoing queue it will not be reused by the producer. The producer will scan upto this packet at most since it must always leave one in the queue. This means the message contents can be safely used by the consumer process until another packet is consumed..

### 2.3 Producer and Consumer timeouts

Both `produce` and `consume` may fail so a further variant is to include an overload that keeps trying for a fixed period of time. It is best for neither agent to enter a busy-wait state so these functions judiciously yield and sleep and the processes/threads. On a single cpu machine having either agent busy-wait prevents the other agent unblocking the situation. This will eventually resolve itself but is similar to a deadlock.

```

bool produce( const Message& message, size_t period )
{
    Ticker ticker;

    if( produce( message ) )
        return true;

    yield();

    for( size_t t = 1; ticker.period() < period; t *= 2 )

```

```

    {
        if( produce( message ) )
            return true;

        sleep( t );
    }

    return false;
}

```

The `Ticker` instance counts milliseconds since its construction. The process (thread) yields first to allow a fast switch to the consumer if the agents have their execution interleaved. If this fails then it backs off for at most double the period specified which is the true timeout. Yielding only makes sense in threaded environments and would be removed for inter-cpu communication. Similarly for the consumer.

```

const Message* consume( size_t period )
{
    Ticker ticker;

    const Message* msg;
    if( ( msg = consume() ) != 0 )
        return msg;

    yield();

    for( size_t t = 1; ticker.period() < period; t *= 2 )
    {
        if( ( msg = consume() ) != 0 )
            return msg;

        sleep( t );
    }

    return 0;
}

```

### 3 Conclusion

The queue implementation detailed in this paper is very efficient since the synchronisation primitives will translate into either no-ops or single machine code instructions to adjust the memory bus. There is no requirement to call OS level functions, speculatively lock structures or flush caches. The queues also provide a high-level interface to send messages with efficient backoff. Using largish timeouts does not affect performance but enables the two agents to run at

synchronised speeds without the application paying too much attention to this matter.

## References

- [1] “Wait-Free Synchronization”, M. Herlihy, ACM Transactions on Programming Languages and Systems, 1991.
- [2] “Memory Ordering in Modern Microprocessors”, P. McKenney, Linux Journal, 2005.
- [3] “Linux Cross Reference (asm-xxx/system.h)”.