A Lock Free Bounded Queue Implementation

1. Overview

Currently, lockless queue is widely used in multi-processor concurrency scenario. Classical lock free queue/list based on pointer and memory manager(allocator/garbage collector) presented in [1] has been implemented in variable libraries, such as java concurrency lib[2] and boost lockfree lib[3]. DPDK[4] has provided a concurrent queue which supports batch enqueue/dequeue, while it is not real lock less. This document presents a lock free bounded queue based on array and primitive index manipulations that enables shared memory based implementation.

2. Algorithm

This section gives details implementation in C++.

```
atomic_ulong list[LISTSIZE];
atomic_ulong head;
atomic_ulong tail;
ulong mask;

bool isEmpty() {return head == tail;}
bool isFull() { return (tail - head) >= LISTSIZE; }
```

void init()

```
{
	head = 0;
	tail = 0;
	for(int i = 0; i < LISTSIZE; i ++)
	{
		list[i] = FREESLOT;
	}
}

bool pushMul(uint item)
{
	bool rc = false;

	do
	{
		if(isFull())
		{
			return false;
		}
```

```
            ulong localTail = tail;
1.          int tailIndex = localTail & mask;
2.          ulong oldValue = list[tailIndex];

3.          if(localTail != tail)
            {
                  continue;
            }

4.          if(!isEmptySlot(oldValue))
            {
                  continue;
            }

5.          ulong newValue = (NextCounter(oldValue)) | item;
6.          rc = list[tailIndex].CAS(oldValue, newValue);

            ulong nextTail = localTail + 1;
7.          tail.CAS(localTail, nextTail);
      }while(!rc);

      return true;
}

bool popMul(uint& item)
{
      bool rc = false;

      do
      {
            if(isEmpty())
            {
                  return false;
            }

            ulong localHead = head;
            int headIndex = localHead & mask;
            ulong oldValue = list[headIndex];

            if(localHead != head)
            {
                  continue;
            }
            if(isEmptySlot(oldValue))
            {
                  continue;
            }

            ulong newValue = (NextCounter(oldValue)) | FREESLOT;
            item = oldValue & DATAMASK;

            rc = list[headIndex].CAS(oldValue, newValue);
            ulong nextHead = localHead + 1;
            head.CAS(localHead, nextHead);

      }while(!rc);
```

```
        return true;
}
//CAS = Compare and Set
```

```
This implementation is based on algorithm described in [1]. We take a bounded
array work as storage; helper mechanism to guarantee lock free; a counter embedded
in value to solve ABA problem; and index instead of pointer manipulation to
enabled Shared Memory usage.
```

3. Properties Prove

Obviously, the list[i].CAS is the linearization point when enqueue/dequeue succeeded.

3.1 Exclusion

Lemma1: When a slot had been filled in, it can not be modified by another en-queue.

The sequence of enqueue is:

R(tailIndex) → R(oldValue) → Assert oldValue is Empty → slot[tailIndex].CAS(oldValue, newValue)

If the slot is modified by another enqueue, there should have been slot[tailIndex] == EMPTYSLOT; that is conflict to the assumption that the slot had been enqueued. ■

Lemma2: When a slot had been dequeued, it can not be modified by another de-queue

Similar to Lemma1. ■

Lemma3: A slot would be dequeued only when the slot had been enqueued. That is, there is no elimination.

The sequence of dequeue is:

R(headIndex) → R(oldValue) → Assert oldValue is NOT Empty → slot[headIndex].CAS(oldValue, newValue)

The dequeue requires that the slot's value is not empty. The only chance that a slot could be set as NOT empty is in step 6 in enqueue, and as declared above, this step is the linearization point of enqueue. So, a dequeue can only after an enqueue. ■

3.2 FIFO

Lemma4: Suppose the current slot index had been enqueued is n, the next slot would be enqueued must b (n + 1); not

case 1: n;

case 2: n + 1 + x (x > 0);

case 3: n – x (x > 0).

case 1: According to lemma1, the next slot can not be n.

case 2: Without loss of generality, set x = 1, that is suppose a thread(Ta) is going to enqueue into slot[n + 2] in this case.

Ta must had read the tail == n + 2.

← A thread (Tb) must have set the tail from n + 1 to n + 2.

← Tb must have tried slot[n + 1].CAS.

← If Tb succeeded, Tb must have enqueued the slot[n + 1], it proves the lemma.

← If Tb failed, there must be slot[n + 1] != oldValue.

← The slot[n + 1] is not empty. That proves the case.

← The counter changed. When the counter changed, there must be some operation after Tb read the old value, enqueue or dequeue, whichever. As we suppose the value of tail is always increasing, step7 must fail. That is, Tb can not forward the index into n + 2.

case 3: According to case2, if Ta is going to enqueue into slot[n – x], it must be the sequence:

Ta.R(tailIndex = n – x) → Tb.enqueue/dequeue → Ta continues enqueue.

The sequence of enqueue is:

Ta.R(localTail) → Ta.R(tailIndex = n – x) → Ta.R(oldValue) → Ta.Assert (localTail == tail) → Ta.Assert(oldValue == EMPTY) → Ta.CAS(slot[tailIndex], oldValue, newValue).

If Ta succeeded in step6 → oldValue == EMPTY.

The sequence must be:

Ta.R(localTail) → Ta.R(tailIndex = n – x) → Tb.Enqueue → Tb.Increase(tail) → Tc.Dequeue → Ta.R(oldValue) → Ta.Assert(localTail == tail) → Failure.

In fact, this is the same case as the second scenario of case 2. ■

3.3 Wait Free

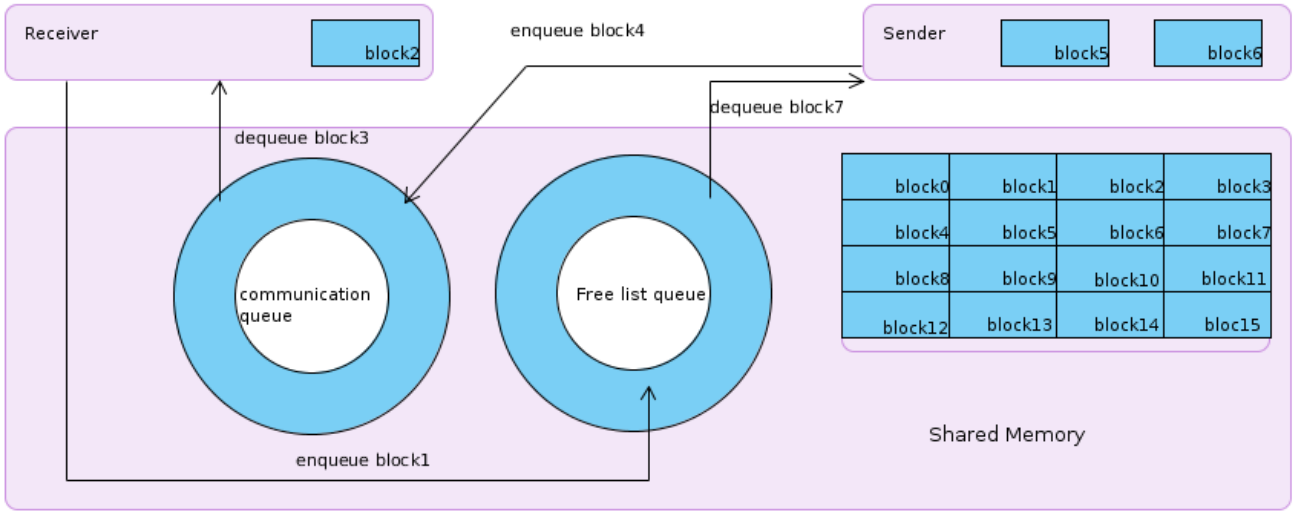
According to [1], as long as there are one enqueue thread and one dequeue thread, the algorithm is wait free. While in fact, it is not really wait free if all enqueue threads dead as the dequeue thread can not help enqueue. While the gap is slim.

Lemma5: If all enqueue threads dead, there is no more than one message that can not be dequeued.

According to the FIFO property, when the current enqueued slot is n, a thread can not enqueue into slot[n + 2] before slot[n + 1] enqueued. When all enqueue threads dead, the only slot with undetermined state would be n + 1. ■

4. Scenario

Following figure presents an IPC solution based on our algorithm. Memory, including memory for our queues, were allocated in shared memory in advance. Memory management is implemented by free list queue, communication is implemented by communication queues.



The solution depends merely on manipulations on primitives, which enables the solution for scenarios such as IPC between KVM VMs , or between VM and host, and also the ordinary IPC use cases. Besides, this is a zero-copy solution.

5. Benchmark

6. Further improvement

As the algorithm is based on xchg primitive, it has inherited shortcoming in scalability, especially when it is used for memory management. In cases when batch enqueue/dequeue is required, a layered list could be adapted.

Also, garbage collector would be required in production environment.

7. Bibliography

[1]. Maurice Herlihy, Nir Shavit. The Art of Multiprocessor Programming. Chapter 10.
[2]. Java libraries
[3]. Boost libraries
[4]. http://www.dpdk.eu/