Scalable Synchronous Queues

Nakul Chaudhari

June 25, 2012

Background

1. Need for Concurrent systems
2. Need for Multiprocessor systems - individual processors reaching a limit of clock speed
3. Using all the parallel processing power we have
4. Concurrent data structures to communicate or synchronize between them
5. Concurrent Queues, Synchronous Asynchronous Queues

Motivation

1. Performance
2. OSX job scheduler grand central
3. Java job scheduler
4. Increase in performance by use of Java 6 implementation

Background

1. Producer and consumers
2. Put and take
3. Producer and consumer problem - do a put in a full buffer, take from an empty buffer

Background

1. Wait-free, lock-free and obstruction free
2. In wait-free algorithms all operations are guaranteed to complete its methods call within a bound of its own execution steps.
3. Lock-free guarantee progress of at least some operation and thus progress of the overall program.
4. Obstruction-free guarantees progress of some thread in absence of contention, and this has a weaker condition then the previous two types.

Naive Synchronous Queue


```
public class NaiveSQ<E> {
  boolean putting = false;
  E item = null;

  public synchronized E take() {
    while (item == null)
      wait();
    E e = item;
    item = null;
    notifyAll();
    return e;
  }

  public synchronized void put (E e) {
    if (e == null) return;
    while (putting)
      wait();
    putting = true;
    item = e;
    notifyAll();
    while (item != null)
      wait();
    putting = false;
    notifyAll();
  }
}
```

1. Keywords -
   1. Synchronized
   2. wait()
   3. notifyAll()
2. Disadvantage - Quadratic wake-ups
3. Why notifyAll() and not notify()

Naive Synchronous Queue

```
00 public class NaiveSQ<E> {
01   boolean putting = false;
02   E item = null;
03
04   public synchronized E take() {
05     while (item == null)
06       wait();
07     E e = item;
08     item = null;
09     notifyAll();
10     return e;
11   }
12
13   public synchronized void put (E e) {
14     if (e == null) return;
15     while (putting)
16       wait();
17     putting = true;
18     item = e;
19     notifyAll();
20     while (item != null)
21       wait();
22     putting = false;
23     notifyAll();
24   }
25 }
```

1. using notify - P1 - 21, P2-16, C1 notifies P2, P2 sleeps again as putting is still true. P1 is still waiting and never notified. All future P and C are blocked.
2. Putting flag - P2 might come and set data to not null before P1 can check it and move ahead. Also if C1 sets it to false, P2 could get inside change data again before P1 checks it.

Hanson's Synchronous Queue

```
public class HansonSQ<E> {
  E item = null;
  Semaphore sync = new Semaphore(0);
  Semaphore send = new Semaphore(1);
  Semaphore recv = new Semaphore(0);

  Public E take() {
    recv.acquire();
    E x = item;
    sync.release();
    send.release();
    return x;
  }

  public void put(E x) {
    send.acquire();
    item = x;
    recv.release();
    sync.acquire();
  }
}
```

1. Keywords -
   1. Semaphore
   2. release()
   3. acquire()
2. Wake-ups to only single consumer or producer
3. total 6 synchronizations per handoff

Java 5 Synchronous Queue


```
public class Java5SQ<E> {
  ReentrantLock qlock = new ReentrantLock();
  Queue waitingProducers = new Queue();
  Queue waitingConsumers = new Queue();

  static class Node
    extends AbstractQueuedSynchronizer {
    E item;
    Node next;

    Node(Object x) { item = x; }
    void waitForTake() { /* (uses AQS) */ }
    E waitForPut() { /* (uses AQS) */ }
  }

  public E take() {
    Node node;
    boolean mustWait;
    qlock.lock();
    node = waitingProducers.pop();
    if(mustWait = (node == null))
      node = waitingConsumers.push(null);
    qlock.unlock();

    if (mustWait)
      return node.waitForPut();
    else
      return node.item;
  }

  public void put(E e) {
    Node node;
    boolean mustWait;
    qlock.lock();
    node = waitingConsumers.pop();
    if (mustWait = (node == null))
      node = waitingProducers.push(e);
    qlock.unlock();

    if (mustWait)
      node.waitForTake();
    else
      node.item = e;
  }
}
```

1. Keywords -
   1. ReentrantLock
   2. lock()
   3. unlock()
2. 3 synchronizations per handoff
3. Queue implementation allows producers to publish data items instead of having to awaken after blocking on semaphore, consumers need not wait also

Java 6 Synchronous Queue


```
class Node { E data; Node next;...}

void enqueue(E e) {
    Node offer = new Node(e, Data);

    while (true) {
        Node t = tail;
        Node h = head;
        if (h == t || !t.isRequest()) {
            Node n = t.next;
            if (t == tail) {
                if (null != n) {
                  casTail(t, n);
                } else if(t.casNext(n, offer)) {
                    casTail(t, offer);
                    while (offer.data == e)
                        /* spin */;
                    h = head;
                    if (offer == h.next)
                    casHead(h, offer);
                    return;
                }
            }
        } else {
            Node n = h.next;
            if (t != tail || h != head || n == null)
                continue; // inconsistent snapshot
            boolean success = n.casData(null, e);
            casHead(h, n);
            if (success)
                return;
        }
    }
}
```

1. Keywords -
   1. casFIELD(old,new)
2. Non blocking
3. Queue implementation allows producers to publish data items instead of having to awaken after blocking on semaphore, consumers need not wait also

Java 6 Synchronous Queue

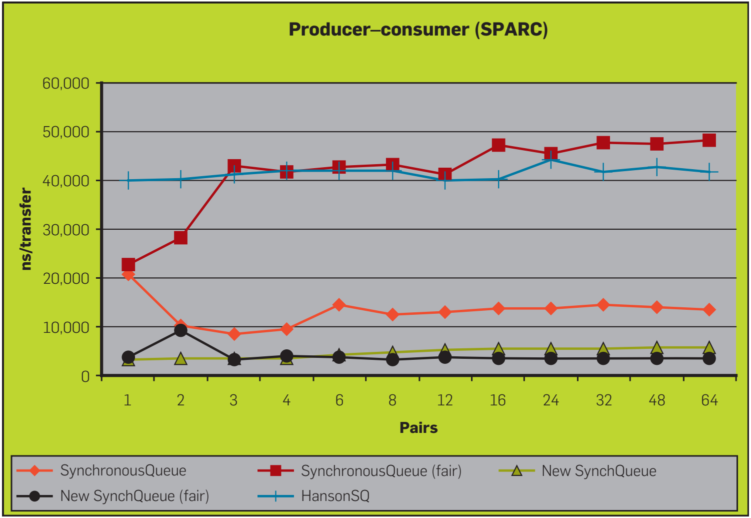
```
class Node { E data; Node next;...}

void enqueue(E e) {
    Node offer = new Node(e, Data);

    while (true) {
        Node t = tail;
        Node h = head;
        if (h == t || !t.isRequest()) {
            Node n = t.next;
            if (t == tail) {
                if (null != n) {
                  casTail(t, n);
                } else if(t.casNext(n, offer)) {
                    casTail(t, offer);
                    while (offer.data == e)
                        /* spin */;
                    h = head;
                    if (offer == h.next)
                    casHead(h, offer);
                    return;
                }
            }
        } else {
            Node n = h.next;
            if (t != tail || h != head || n == null)
                continue; // inconsistent snapshot
            boolean success = n.casData(null, e);
            casHead(h, n);
            if (success)
                return;
        }
    }
}
```

1. How CAS works
2. 2 states
3. 2 steps to get to the same state again
4. Help out

Conclusion


Producer–consumer (SPARC)
ns/transfer
60,000
50,000
40,000
30,000
20,000
10,000
0
1
2
3
4
6
8
12
16
24
32
48
64
Pairs
SynchronousQueue
SynchronousQueue (fair)
New SynchQueue
New SynchQueue (fair)
HansonSQ