Scalable Synchronous Queues

By William N. Scherer III, Doug Lea, and Michael L. Scott.
Summary on the paper by Nakul Chaudhari
for Seminar : Advanced Seminar Course Verification of Concurrent Programs

Introduction

Since many years modern computers have increasingly moved towards multicore architectures. To harness the computing power of these processors concurrent processing is needed, and to benefit the whole system working towards one purpose, synchronization between these concurrent process is needed. Concurrent data structures resident in shared memory make such synchronization possible.

Individual processes or threads may face different kind of delays, from short (e.g. cache misses) to long due to getting descheduled. Such descheduled threads may cause priority inversion of important threads or even deadlocks in system. Such descheduling generally appears due to use of locks. Locks are important part and of concurrent programs, making shared data processing mutually exclusive for threads. These critical sections guarded by locks if made enough smaller can be executed by nonblocking or lock-free algorithms using modern techniques like CAS (compare and set). But at the same time this makes designing such algorithms complex and a difficult task. This paper introduces a new nonblocking algorithm to a concurrent data structure called ‘Synchronous Queues’ (SQ’s). We see how its working yields significantly better results over previous blocking algorithms for the same.

Various mechanisms exist to transfer data between threads of concurrent systems. We can imagine threads as producers and consumers pairing up to transfer data. To get a steady flow of data we can buffer up unconsumed data ‘put’ by producers and consumers can then do a ‘take’ from this pool of data. This is a asymmetric fashion of transfer. In synchronous methods producers and consumers can arrive at any desired time, but they leave together. Thus they do a handoff by waiting for each other and leaving in pairs. Example usage can be : if a producer thread has tasks to assign and a consumer thread has computing power to supply for such tasks, then producers and consumers can be paired, and the producer is guaranteed that a consumer has taken his task as the handoff is symmetric. Unlike in an asymmetric assignment of task, the producer has no idea if the task was successfully assigned or not.

In the following pages we will take a look at the various implementations of Synchronous Queues. We will take a look in detail at the most simple implementation and the latest nonblocking implementation. We will also see how the various algorithms have made progress over the previous ones and what disadvantages they still have. In the end we will briefly take a look at performance of the latest implementation.

Naive Synchronous Queue

It is perhaps the most simple implementation of a synchronous queue, using popular Java constructs for concurrent programming. This implementation uses a simple monitor for serializing access to the members of the synchronous queue. The Queue in fact contains placeholder for just one data item and a flag called putting. Putting indicates to another incoming producer whether a previous producer has already input data. The Java constructs used are :

```
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 }
```

fig.1

synchronized - synchronizes access to methods of this class's objects.

The next three are inherited from the Java Object superclass, and every object in Java has them:

wait - causes current thread to wait until notify or notifyAll are called by other threads on the object (they are inherited from the Java Object superclass, and every object in Java has them),

notify - wakes up any single thread waiting on this objects monitor,

notifyAll - wakes up all the threads waiting on this objects monitor.

The implementation uses notifyAll instead of notify, as using notify the wrong waiting thread can be woken up, resulting in a deadlock. A simple e.g. let P1,P2 be the producers, C1 be the consumer. P1 enters to put and waits at line 21. P2 enters and waits at line 16. C1 enters, takes data, sets it to null and notifies P2. P2 wakes up but sleeps again as putting is still set to true. P1 is still waiting and is never notified. So putting is never set to false and any new producer or P2 can never put new data and all future consumers are also made to wait as no new data is available. Unfortunately using notifyAll creates quadratic wake up calls of the n producers and consumers. Also using a monitor based approach we incur heavy costs of blocking and unblocking threads accessing the object.

Usage of 'putting' flag is must in this case. If putting would not have been used then another producer would have had the ability to set the data, and an already old producer (who has put some data) who is waiting for the data to be set to null would never get out of its while loop. Also the same case would arise if the putting flag is set to false by the take method instead of the put method. Hence it is imperative to use the putting flag and also to set (to true or false) only in the put method.

Hanson's implementation improves upon the naive SQ by waking up only the corresponding consumer or producer. Still it uses three semaphores which cause blocking and hence six synchronization operations per handoff between a producer and a consumer. Java 5 improves upon this by using only three synchronization operations per handoff, but still uses a blocking algorithm.

Java 6 Synchronous Queue

A major difference between Java 6 and Java 5 implementation is use of a nonblocking algorithm. Previous implementation used a single lock to serialize access to both producer and consumer queues. Here concurrency is maintained by using atomic machine instructions like the CAS. Also, waiting (for e.g. producer waiting for consumer to take data) is achieved by spinning e.g. in a while loop until desired event occurs. Using a nonblocking algorithm higher scalability is achievable.

```
00 class Node { E data; Node next;...}
01
02 void enqueue(E e) {
03     Node offer = new Node(e, Data);
04
05     while (true) {
06        Node t = tail;
07        Node h = head;
08        if (h == t || !t.isRequest()) {
09            Node n = t.next;
10            if (t == tail) {
11                if (null != n) {
12                  casTail(t, n);
13                } else if(t.casNext(n, offer)) {
14                    casTail(t, offer);
15                    while (offer.data == e)
16                        /* spin */;
17                    h = head;
18                    if (offer == h.next)
19                    casHead(h, offer);
20                    return;
21                }
22            }
23        } else {
24            Node n = h.next;
25            if (t != tail || h != head || n == null)
26                continue; // inconsistent snapshot
27            boolean success = n.casData(null, e);
28            casHead(h, n);
29            if (success)
30                return;
31        }
32     }
33 }
```

fig.2

As shown in fig.2 a enqueue method is used by producers to put data, and analogous dequeue method (not in figure) by consumers to take data. A single queue implemented by a linked list data structure holds both requests or data nodes, but only one kind at a time. The algorithms uses many CAS methods in the form cas*field(old,new).*

The enqueue code is separated into two cases depending on the existing nodes in the linked list - either there are zero or more existing data nodes put by previous producers or one or more request nodes added by existing consumers. In the first case the algorithm checks whether read values are consistent (line 10). If the queue's tail pointer is current (line 13) it tries to insert a data node in the queue. When inserted it waits until a consumer takes it and sets the inserted node's data pointer to null (line 15-16). It advances the head pointer of linked list if needed and returns. In the second case, the linked list already contains request nodes. The algorithm tries to supply data to the request node immediately following the head. If successful it advances the head node and returns. If not, even then it advances the head node (it helps out other threads supplying data) and tries to supply data to the node following the node it just checked.

CAS (compare and set) methods are the basic building blocks of nonblocking algorithms. We will first discuss some of their salient features, and then look at how to build nonblocking algorithms with our specific Java 6 implementation example.

CAS versus software locks

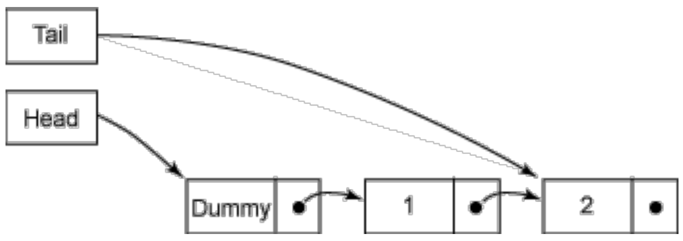
With compare and set we can atomically update shared data without locking parts of code [2]. What a CAS method does is, that it compares the value of the shared data with an expected value (e.g. just retrieved current value) and if same (not changed between retrieval and compare), sets it to a new value. These methods are atomic in hardware and easily outperform software atomic operations (e.g. those done using locks). Modern Processors provide us with such CAS methods. Also, only since Java 5.0 it was possible to develop nonblocking algorithms with help of these.

When we use locks or monitors to develop concurrent data structures we essential block threads when another thread is are performing certain modifications on the data, so that they remain consistent. With such software locks we need to suspend certain threads and reschedule them. Thus when rescheduled we have to make context switches and at higher granularity levels as we lock significant portions of code and some times even entire methods. Unlike software locks, using CAS we synchronize at finer level of granularity and those threads which lose out can retry immediately (generally inside programming loops e.g. while). Also, nonblocking algorithms generally are successful at the first attempt and even with few failed attempts they still outperform their counterparts.

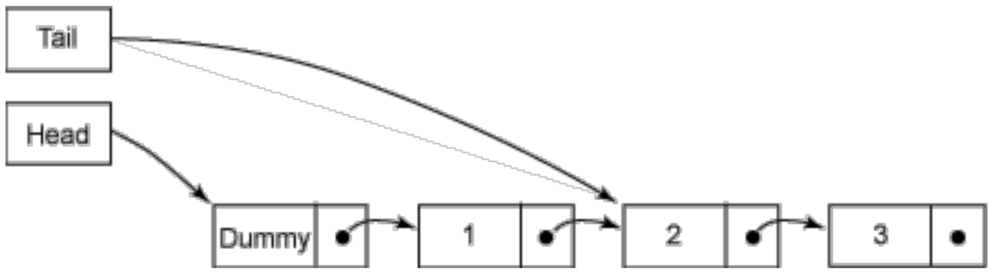
Nonblocking concurrent algorithms using CAS

Building nonblocking concurrent data structures can be complex, as CAS provides us atomic updates on single units of data but not on more at the same time. An easy guideline then can be, is to always maintain consistent data structure, even between an individual threads modifying operations start and end. Other threads should also be able to tell whether a thread currently modifying the data structure has finished modification and if not what operations will it need to complete its operation.

For e.g. in our Synchronous queue example we have a linked list. There can be two possible cases - normal state where the linked list is stable and up to date i.e. the tail pointer points to the last element and the next pointer of tail points to null.



The other state is intermediate stage where maybe a certain thread has inserted new data by linking a new node to the tail node, but yet to update the tail pointer.



So in our case when one thread has inserted the new node, and still has to update the tail pointer, another new thread coming sees that the list in inconsistent (and doesn't perform its own insertion, instead), it helps out in making it consistent by updating the tail pointer, and then itself retrying again for insertion. When the first thread realizes that someone else has updated the tail pointer it just fails its CAS of tail and return.

There are three basic types of guarantees a nonblocking algorithms can make about its progress. In wait-free algorithms all operations are guaranteed to complete its methods call within a bound of its own execution steps. Lock-free guarantee progress of at least some operation and thus progress of the overall program. Thus wait-free have a stronger condition and all wait-free are lock-free but not vice versa. Obstruction-free guarantees progress of some thread in absence of contention, and this has a weaker condition then the previous two types. Our algorithm of SQ is a lock-free algorithm.

Performance summary

From the various tests performed by the authors it is evident that blocking and contention between threads are the major roadblocks on our path to achieving high scalability. The latest implementation makes significant progress in both areas by implementing a nonblocking algorithm. This Java 6 implementation significantly outperforms previous implementations. One area where they suggest we can make improvement is by reducing the contention faced by thread at the tail end of the linked list.

References:

1) Most of the content is based on the paper ‘Scalable Synchronous Queues’ By William N. Scherer III, Doug Lea, and Michael L. Scott.

2) The explanation on CAS is based on the article at http://www.ibm.com/developerworks/java/library/j-jtp04186/index.html titled ‘Java theory and practice: Introduction to nonblocking algorithms’ by Brian Goetz.