Implement of thread-safe Queue

## Siyuan Zhu, Henjun Zhan, Yinghao Wu

Stevens Institute of Tenhnology, NY

[ywu](mailto:fmichael%2Cscottg@cs.rochester.edu)100@stevens.edu

### Abstract

Drawing ideas from previous authors, we present a new non-blocking concurrent queue algorithm and a new two- lock queue algorithm in which one enqueue and one de- queue can proceed concurrently. Both algorithms are sim- ple, fast, and practical; we were surprised not to find them in the literature. Experiments on a 12-node SGI Challenge multiprocessor indicate that the new non-blocking queue consistently outperforms the best known alternatives; it is the clear algorithm of choice for machines that provide a universal atomic primitive (e.g. compare and swap or load linked/store conditional). The two-lock concurrent queue outperforms a single lock when several processes are competing simultaneously for access; it ap- pears to be the algorithm of choice for busy queues on ma- chines with non-universal atomic primitives (e.g. test and set). Since much of the motivation for non-blocking algorithms is rooted in their immunity to large, unpre- dictable delays in process execution, we report experimental results both for systems with dedicated processors and for systems with several processes multiprogrammed on each processor.

**Keywords:** concurrent queue, lock-free, non-blocking,

compare and swap, multiprogramming.

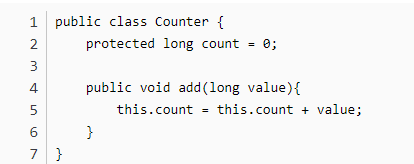
This work was supported in part by NSF grants nos. CDA–94–01142 and CCR–93–19445, and by ONR research grant no. N00014–92–J–1801 (in conjunction with the DARPA Research in Information Science and Technology—High Performance Computing, Software Science and Tech- nology program, ARPA Order no. 8930).

# Introduction to thread safety

Thread safety is an issue in the multithreaded world. Thread safety can be simply understood as a method or an instance that can be used in a multithreaded environment without problems.

## Causes of thread insecurity

Running multiple threads in the same program does not itself cause a problem; the problem is that multiple threads are accessing the same resource.For example, the same memory area (variable, array, or object), system (database, Web services, etc.), or file.In fact, these problems are only possible if one or more threads have written to these resources, and it is safe for multiple threads to read the same resource as long as the resource has not changed It is possible for multiple threads to execute the following code simultaneously.



Imagine threads A and B executing the add() method on the same Counter object at the same time, and we have no way of knowing when the operating system will switch between the two threads.Instead of treating this code as a single instruction, the JVM executes it in the following order:

Get the value of this.count from memory into the register

Increases the value in the register by value

Writes the value of the register back to memory

Observe what happens when threads A and B are interleaved:

This. Count = 0;

A: Read this.count to A register (0)

B: Read this.count into a register (0)

B: Add 2 to the register

B: Writes back register value (2) to memory. This.count is now equal to 2

A: Add 3 to the value of the register

A: Writes back register value (3) to memory. This.count is now equal to 3

The two threads add 2 and 3 to the count variable, and the value of the count variable should be equal to 5 when the two threads finish executing.However, since the two threads are executing across each other, both threads are reading an initial value of 0 from memory.Then add 2 and 3, respectively, and write them back to memory.The final value is not the expected value of 5, but the value of the thread that last wrote back to memory, in the example above, is thread A, but in reality it could be thread B.Without proper synchronization mechanisms in place, cross execution between threads is unpredictable.

## Race condition & critical region

When two threads are competing for the same resource, a race condition is said to exist if the order of access to the resource is sensitive.The area of code that causes a race condition to occur is called the critical area.In the example above, the add() method is a critical section that generates race conditions.Race conditions can be avoided by using proper synchronization in the critical region.

## Shared resources

Code that allows multiple threads to execute simultaneously is called thread-safe code.Thread-safe code does not contain race conditions.Race conditions are raised when multiple threads update a shared resource simultaneously.Therefore, it is important to understand what resources are shared when a Java thread executes.

## A local variable

Local variables are stored on the thread's own stack.That is, local variables are never shared by multiple threads.So, local variables of the underlying type are thread-safe.Here is an example of a local variable of the underlying type:

# Implementation methods for thread safety

Synchronization is when multiple threads concurrently access shared data, ensuring that the shared data is only used by one thread (or some, when using semaphores).

Mutually exclusive synchronization

Mutual exclusion is a means to achieve synchronization

1) Synchronized keyword

After compilation, two bytecode instructions, monitorenter and Monitorexit, are formed before and after the synchronization block.

In the JVM specification, when you execute the Monitorenter directive, you first try to get the lock for an object.If the object is not already locked, or if the current thread only has an object lock, the lock counter is incremented by 1.

When the Monitorexit instruction is executed, the lock counter is reduced by 1, and when the counter changes to 0, the lock is released.If the object lock fails, the current thread will block and wait.

2) The reentry lock ReentrantLock in the Java.util.concurrent package

ReentrantLock has some advanced features compared to synchronized: wait interruptible, fair lock, and lock binding to multiple conditions.

Wait interruptible: Threads holding a lock do not release the lock for a long time, and threads waiting for a lock can give up waiting and do something else

Fair locking: The synchronized lock is unfair, and ReentrantLock, by default, is also unfair, but ReentrantLock achieves fair locking

Locks bind multiple conditions: A ReentrantLock binds multiple Condition objects

* + 1. Non-blocking synchronization

The main problem with mutex synchronization is the performance problem caused by thread blocking and thread awakening. This synchronization is called blocking synchronization.

Mutex synchronization is a pessimistic locking strategy. If the task does not do the correct synchronization measures, it will definitely cause problems. Lock should be added regardless of whether there is a competition for shared data.

With the development of hardware instruction sets, an optimistic locking strategy can be used, that is, the operation is executed first, and if there is no thread contention, the operation is successful;If there is thread contention, try again and again until you succeed.

3. No synchronization scheme

Methods that do not involve sharing data, and do not require any synchronization measures to ensure correctness, are inherently thread-safe

1) Reentrant code

Code reentrant rule of thumb: If a method enters the same data, it always returns the same result.

2) Thread local storage

Code that shares data is executed in the same thread, limiting the visibility of the shared data to the same thread

The ThreadLocal class in Java implements ThreadLocal storage.

Example: In the classic Web interaction model, a request corresponds to a server thread, and the server can solve thread-safety problems by using thread-local storage.

# Two kinds of thread-safe queue

## Blockingqueue

1Compare and swap, introduced on the IBM System 370, takes as arguments the address of a shared memory location, an expected value, and a new value. If the shared location currently holds the expected value, it is assigned the new value atomically. A Boolean return value indicates whether the replacement occurred.

2A *wait-free* algorithm is both non-blocking and starvation free: it guarantees that every active process will make progress within a bounded number of time steps.

# References

1. T. E. Anderson. The Performance of Spin Lock Alter- natives for Shared-Memory Multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):6–16, January 1990.
2. G. Barnes. A Method for Implementing Lock-Free Data Structures. In *Proceedings of the Fifth Annual ACM Symposium on Parallel Algorithms and Archi- tectures*, Velen, Germany, June – July 1993.
3. A. Gottlieb, B. D. Lubachevsky, and L. Rudolph. Basic Techniques for the Efficient Coordination of Very Large Numbers of Cooperating Sequential Pro- cessors. *ACM Transactions on Programming Lan- guages and Systems*, 5(2):164–189, April 1983.
4. M. P. Herlihy and J. M. Wing. Axions for Concurrent Objects. In *Proceedings of the 14th ACM Symposium on Principles of Programming Languages*, pages 13– 26, January 1987.