Languages » C / C++ Language » Templates **Beginner** License: The Code Project Open License (CPOL) C++, C#, Windows, Dev

Posted : **30 Jan 2008**
Updated : **24 Feb 2008**
Views : **6,096**

Lock Free Queue implementation in C++ and C#

By Idaho Edokpayi

Lock Free Queue implementation in C++ and C#

Note: This is an unedited reader contribution

6 votes for this Article.
Popularity: 2.59 Rating: **3.33** out of 5

Download C++ Source - 6.22 KB

Download C# Source- 35.31 KB

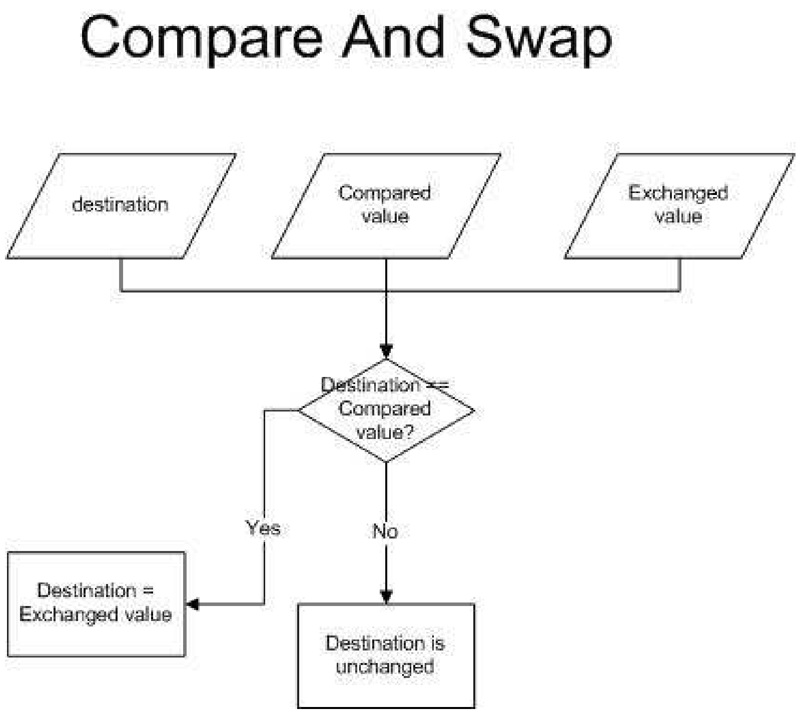
Introduction

This article demonstrates implementation of a "lock free" queue in C# and C++. Lock free queues are typically used in multi-threaded architectures to communicate between threads without fear of data corruption or performance loss due to threads waiting to use shared memory. The goal of the article is to familiarize readers with lock free queues and provide a starting point to writing wait-free production architectures. It should be noted that using lock-free queues is only the beginning - a true lock free architecture use a lock free memory allocator. Implementing lock free memory allocators is beyond the scope of this article however.

Background

Recent developments in CPU architecture has necessitated a change in thinking in high performance software architecture - multithreaded software. Communication between threads in multithreaded architecture has traditionally been accomplished using mutexes, critical sections, and locks. Recent research in algorithms and changes in computer architecture has led to the introduction of "wait free", "lock free", or "non-blocking" data structures. The most popular and possibly the most important is the queue, a First In First Out (FIFO) data structure.

The key to the majority of lock free data structures is an instruction known as **Compare and Swap** (CAS). The flow chart below describes what Compare and Swap does. For the assembler coders out there the instruction is named ***CMPXCHG*** on X86 and Itanium architectures. The special thing about this instruction is that it is atomic- meaning that other threads\processes cannot interrupt until it is finished. Operating Systems use atomic operations to implement sychronization - locks, mutexes, semaphores, and critical sections.

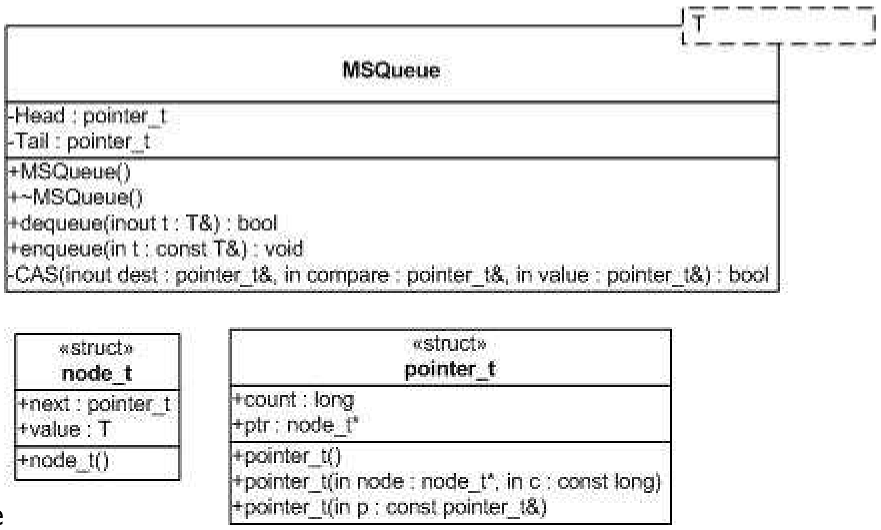


My code draws on research by Maged M. Michael and Michael L. Scott on non-blocking and blocking concurrent queue algorithms. In fact, an implementation of their queue is now part of the Java concurrency library. Their paper demonstrates why the queue is "linearizable" and "lock free". An implementation of the code in C is available here in tar.gz format. The idea is that pointers are reference counted and checked for consistency in a loop. The reference count is meant to prevent what is referred to the "ABA problem" - if a process or threads reads a value 'A' then attempts a **CAS** operation, the **CAS** operation might succeed incorrectly if a second thread or process changes value 'A' to 'B' and then back again to 'A'. If the "ABA" problem never occurs the code is safe because:

1. The linked list is always connected.
2. Nodes are only inserted at the end of the linked list and only inserted to a node that has a NULL *next* pointer.
3. Nodes are deleted from the beginning of the list because they are only deleted from the head pointer and head always points to the first element of the list.
4. Head always points to the first element of the list and only changes it's value atomically.

If **CAS** or similar instructions are not available I suggest using the STL queue or a similar queue with traditional sychronization primitives. Michael and Scott also present a "two lock" queue data structure.

Using the code


T
MSQueue
-Head : pointer_t
-Tail : pointer_t
+MSQueue()
+~MSQueue()
+dequeue(inout t : T&) : bool
+enqueue(in t : const T&) : void
-CAS(inout dest : pointer_t&, in compare : pointer_t&, in value : pointer_t&) : bool
«struct» node_t
+next : pointer_t
+value : T
+node_t()
«struct» pointer_t
+count : long
+ptr : node_t*
+pointer_t()
+pointer_t(in node : node_t*, in c : const long)
+pointer_t(in p : const pointer_t&)
UML Diagram of source

Using the code provided with this article is simple.

- C++ class declaration `template< class T > class MSQueue`
  - Include the "lockfree.h" file. `#include "LockFreeQ.h"`
  - Declare C++ queues like this: `MSQueue< int > Q;`
  - Add items to the queue: `Q.enqueue(i);`
  - Remove items from the queue: `bool bIsQEmpty = Q.dequeue(i);` dequeue returns false if the queue is empty and the value of i would be undefined.
- C# class declaration `namespace Lockfreeq { public class MSQueue {`
  - Include the Lock Free Queue DLL: `using Lockfreeq;`
  - Declare a C# queue: `MSQueue< int > Q = new MSQueue< int >();`
  - Add items to the queue: `Q.enqueue(i);`
  - Remove items from the queue: `bool bIsQEmpty = Q.dequeue(ref i);` dequeue returns false if the queue is empty and the value of i would be undefined.

Points of Interest

Did you know that Valve Software (makers of Half-Life computer game) have switched to a wait free architecture?

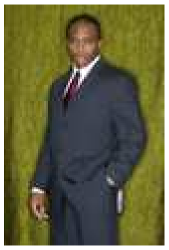
History

- This is my first article! (30.1.2008)
- Revised 21:00 30.1.2008 (Thanks to the early commenters for spotting those mistakes!)
- Noted need for memory allocator. 31.1.2008
- Revision to correct inaccuracy with regards to Java Concurrency library.

License

About the Author

Idaho Edokpayi



Idaho Edokpayi is a Consultant working in EMC's Microsoft Practice specializing in WSS 3.0, MOSS 07, .NET, and ASP.NET technology. He also likes to write C++, Win32/64 API, and DirectX code in his spare time.

Occupation:	Web Developer
Company:	EMC Corporation
Location:	🇺🇸 United States

Discussions and Feedback

23 messages have been posted for this article. Visit **http://www.codeproject.com/KB/cpp/lockfreeq.aspx** to post and view comments on this article, or click **here** to get a print view with messages.