

C++11 AtomicLock

Thread Synchronization in C++11 using `<atomic>`

AtomicLock features

- Makes use of `<atomic>` C++11 header to provide with thread synchronization
- Internally uses a simple `std::atomic<bool>`.
- Can be used for mutual exclusion (`lock()`, `try_lock()` and `unlock()`) ...
- ... Or to wait for another task to complete (`wait()`)
- Complies with the C++ Lockable and Mutex concepts (can be used with `std::unique_lock<>`)

lock() algorithm

```
inline void AtomicLock::lock() {  
    do {  
        // Spin on atomic Load, and if the Lock is free...  
        if (ThisLock.load(std::memory_order::memory_order_acquire) == UNLOCKED) {  
            auto _Unlocked = UNLOCKED;  
            // Try to lock it using a CAS operation... If successful, leave the function.  
            if (ThisLock.compare_exchange_strong(_Unlocked, LOCKED,  
                                                std::memory_order_acquire,  
                                                std::memory_order_relaxed))  
                return;  
        }  
        // yield() call to allocate this thread's remaining timeslice for other running threads.  
        std::this_thread::yield();  
        // Repeat while the Lock is not freed or the CAS operation was unsuccessful (that happens when the Lock  
        // has been acquired in the meanwhile).  
    } while (true);  
}
```

try_lock() algorithm

```
inline bool AtomicLock::try_lock() {  
    auto _Unlocked = UNLOCKED;  
    // First check the Lock status before trying to acquire it using a CAS operation.  
    return (ThisLock.load(std::memory_order_acquire) == UNLOCKED &&  
            ThisLock.compare_exchange_strong(_Unlocked, LOCKED,  
                                             std::memory_order_acquire,  
                                             std::memory_order_relaxed));  
}
```

- No spinning like `lock()`, but rather a single check
- Returns true if the `AtomicLock` has been successfully acquired
- Or false if the `AtomicLock` was already acquired or if it has been acquired between the `load` and the `compare_exchange_strong` operations

wait(), unlock() algorithms

```
inline void AtomicLock::wait() {  
    // Spin atomically while the AtomicLock is acquired.  
    while (ThisLock.load(std::memory_order::memory_order_acquire) == LOCKED) {  
        // Yield the calling thread.  
        std::this_thread::yield();  
    }  
}  
  
inline void AtomicLock::unlock() {  
    ThisLock.store(UNLOCKED, std::memory_order_release);  
}
```

AtomicLock performance: Contended case

- 4 threads sharing an unique AtomicLock, a `std::mutex` or a `CRITICAL_SECTION`
- 10 million Acquires and Releases on each thread
- Benchmarks repeated 50 times (SD = ~ 0.03 s)
- Configuration: Microsoft® Windows™ 10 64 bits, Visual Studio™ 2015, Release Mode x86, CPU Intel® Core™ i3 3110M (2.4 GHz, Ivy Bridge, HT enabled)

Time in seconds (s)	AtomicLock	std::mutex	CRITICAL_SECTION
lock()/unlock()	0.62	2.41 (3.9x)	6.50 (10x)
If(try_lock())/unlock()	0.26	1.88 (7.2x)	1.87 (7.2x)

AtomicLock performance: Uncontended case

- This time, only one thread owning an `AtomicLock`, a `std::mutex` or a `CRITICAL_SECTION`
- 10 million Acquires and Releases
- Benchmarks repeated 50 times (SD = ~ 0.014 s)
- Configuration: Microsoft® Windows™ 10 64 bits, Visual Studio™ 2015, Release Mode x86, CPU Intel® Core™ i3 3110M (2.4 GHz, Ivy Bridge, HT enabled)

Time in seconds (s)	AtomicLock	std::mutex	CRITICAL_SECTION
lock()/unlock()	0.123	0.486 (4.0x)	0.304 (2.5x)
If(try_lock())/unlock()	0.122	0.490 (4.0x)	0.308 (2.5x)

AtomicLock conclusion

- A lightweight interface for synchronizing threads
- Great performance, especially in uncontended cases

This program is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program. If not, see [<http://www.gnu.org/licenses/>](http://www.gnu.org/licenses/).

Intel, the Intel Logo, and Intel Core are trademarks of Intel Corporation in the U.S. and other countries.

Microsoft, Visual Studio, Windows are trademarks of Microsoft Corporation in the U.S. and other countries.