

C++11 Concurrency Tutorial - Part 4: Atomic Types

Baptiste Wicht — 2012-07-16 09:22 — 5 Comments

In the previous article, we saw advanced techniques about mutexes. In this post, we will continue to work on mutexes with more advanced techniques. We will also study another concurrency technique of the C++11 Concurrency Library: Atomic Types

Atomic Types

We will take, the example of a Counter:

```
struct Counter {  
    int value;  
  
    void increment(){  
        ++value;  
    }  
  
    void decrement(){  
        --value;  
    }  
  
    int get(){  
        return value  
    }  
};
```

We already saw that this class was not safe at all to use in multithreaded environment. We also saw how to make it safe using mutexes. This time, we will see how to make it safe using atomic types. The main advantage of this technique is its performance. Indeed, in most cases, the `std::atomic` operations are implemented with lock-free operations that are much faster than locks.

The C++11 Concurrency Library introduces Atomic Types as a template class: `std::atomic`. You can use any Type you want with that template and the operations on that variable will be atomic and so thread-safe. It has to be taken into account that it is up to the library implementation to choose which synchronization mechanism is used to make the operations on that type atomic. On standard platforms for integral types like `int`, `long`, `float`, ... it will be some lock-free technique. If you want to make a big type (let's say 2MB storage), you can use `std::atomic` as well, but mutexes will be used. In this case, there will be no performance advantage.

The main functions that `std::atomic` provide are the store and load functions that atomically set and get the contents of the `std::atomic`. Another interesting function is `exchange`, that sets the atomic to a new value and returns the value held previously. Finally, there are also two functions `compare_exchange_weak` and `compare_exchange_strong` that performs atomic exchange only if the value is equal to the provided expected value. These two last functions can be used to implement lock-free algorithms.

`std::atomic` is specialized for all integral types to provide member functions specific to integral (like operators `++`, `--`, `fetch_add`, `fetch_sub`, ...).

It is fairly easy to make the counter safe with `std::atomic`:

```
#include <atomic>

struct AtomicCounter {
    std::atomic<int> value;

    void increment(){
        ++value;
    }

    void decrement(){
        --value;
    }

    int get(){
        return value.load();
    }
};
```

If you test this counter, you will see that the value is always the expected one.

Wrap-Up


In this article we saw a very elegant technique to perform atomic operations on any type. I advice you to use `std::atomic` any time you need to make atomic operations on a type, especially integral types.

The source code for this article can be found on Github
(<https://github.com/wichtounet/articles/tree/master/src/threads/part4>).

Next

In the next post of this series, we will see how to use the Futures facility to perform asynchronous task.

C++ C++11 Concurrency Tutorial Concurrency Performances

Contents © 2017 Baptiste Wicht (mailto:baptistewicht@gmail.com) - Powered by Nikola
(<http://getnikola.com>) - License:  (<http://creativecommons.org/licenses/by/4.0/>)

Source (c11-concurrency-tutorial-part-4-atomic-type.wp)