**Atomics**

A mutex might be a relatively expensive operation in both necessary resources and latency of the exclusive access. So, instead of using mutexes and lock, it might be worth using atomics instead.

The high-level interface of atomics, provides atomic operations using the default guarantee regarding the order of memory access. This default guarantee provides sequential consistency, which means that in a thread, atomic operations are guaranteed to happen in the order as programmed. Thus, problems of reordered statements do not apply.

C++ standard library does not distinguish between a high-level (aka normal or strong) and a low-level (aka weak or relaxed) atomics interface. The term low-level was introduced by Hans Boehm, one of the authors of the library.

# Example of Using Atomics

#include <atomic> // for atomic types

...

std::atomic<bool> readyFlag(false);

void thread1() {

          // do something thread2 needs as preparation

          ...

          readyFlag.store(true);

}

void thread2() {

          // wait until readyFlag is true (thread1 is done)

          while (!readyFlag.load()) {

                   std::this\_thread::sleep\_for(std::chrono::milliseconds(100));

          }

          // do whatever shall happen after thread1 has prepared things

          ...

}

Note that you always should initialize atomic objects because the default constructor does not fully initialize it (it’s not that the initial value is undefined, it is that the lock is uninitialized). For static-duration atomic objects, you should use a constant to initialize them.

If only the default constructor is used, the only operation allowed next is to call a global atomic\_init() operation as follows:

std::atomic<bool> readyFlag;

...

std::atomic\_init(&readyFlag, false);

This way of initialization is provided to be able to write code that also compiles in C.

The two most important statements to deal with atomics are store() and load():

1.   **store()** assigns a new value

2.   **load()** yields the current value

The important point is that these operations are guaranteed to be atomic, so we don’t need a mutex to set the ready flag, as we had to without atomics.

Thus, in the first thread, instead of

~~{~~

~~std::lock\_guard<std::mutex> lg(readyMutex);~~

~~readyFlag = true;~~

~~} // release lock~~

we simply can program:

readyFlag.store(true);

However, when using condition variables, we still need the mutex for consuming the condition variable:

// wait until thread1 is ready (readyFlag is true)

{

          std::unique\_lock<std::mutex> l(readyMutex);

          readyCondVar.wait(l, []{ return readyFlag.load(); });

} // release lock

## Complete example using atomics

#include <atomic>   // for atomics

#include <future>    // for async() and futures

#include <thread>   // for this\_thread

#include <chrono>   // for durations

#include <iostream>

long data;

std::atomic<bool> readyFlag(false);

void provider () {

          // after reading a character

          std::cout << "<return>" << std::endl;

          std::cin.get();

          // provide some data

          data = 42;

          // and signal readiness

          readyFlag.store(true);

}

void consumer () {

          // wait for readiness and do something else

          while (!readyFlag.load()) {

                   std::cout.put(’.’).flush();

                   std::this\_thread::sleep\_for(std::chrono::milliseconds(500));

          }

          // and process provided data

          std::cout << "\nvalue : " << data << std::endl;

}

int main() {

          // start provider and consumer

          auto p = std::async(std::launch::async,provider);

          auto c = std::async(std::launch::async,consumer);

          return 0;

}

Output:

<return>

..........................Hello

value : 42

The store() operation performs a so-called release operation on the affected memory location, which by default ensures that all prior memory operations, whether atomic or not, become visible to other threads before the effect of the store operation.

The load() operation performs a so-called acquire operation on the affected memory location, which by default ensures that all following memory operations, whether atomic or not, become visible to other threads after the load operation.

As a consequence, because the setting of data happens before the provider() stores true in the readyFlag and the processing of data happens after the consumer() has loaded true as value of the readyFlag, the processing of data is guaranteed to happen after the data was provided.

This guarantee is provided because in all atomic operations, we use a default memory order named memory\_order\_seq\_cst, which stands for sequential consistent memory order.

# Atomics and Their High-Level Interface

Class template std::atomic<> provides the general abilities of atomic data types.

Specializations are provided for bool, all integral types, and pointers:

template<typename T> struct atomic;          // primary class template

template<> struct atomic<bool>;                // explicit specializations

template<> struct atomic<int>;

...

template<typename T> struct atomic<T\*>; // partial specialization for pointers

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Operation** | **triv** | **int** | **ptr** | **Effect** |
| **atomic a = val** | Yes | Yes | Yes | Initializes a with val (not an atomic operation) |
| **atomic a; atomic\_init(&a,val)** | Yes | Yes | Yes | Ditto (without atomic\_init(), a is not initialized) |
| **a.is\_lock\_free()** | Yes | Yes | Yes | true if type internally does not use locks |
| **a.store(val)** | Yes | Yes | Yes | Assigns val (returns void) |
| **a.load()** | Yes | Yes | Yes | Returns copy of the value of a |
| **a.exchange(val)** | Yes | Yes | Yes | Assigns val and returns copy of old value of a |
| **a.compare\_exchange\_strong(exp, des)** | Yes | Yes | Yes | CAS operation (see below) |
| **a.compare\_exchange\_weak(exp,  des)** | Yes | Yes | Yes | Weak CAS operation |
| **a = val** | Yes | Yes | Yes | Assigns and returns copy of val |
| **a.operator atomic()** | Yes | Yes | Yes | Returns copy of the value of a |
| **a.fetch\_add(val)** |  | Yes | Yes | Atomic t+=val (returns copy of new value) |
| **a.fetch\_sub(val)** |  | Yes | Yes | Atomic t-=val (returns copy of new value) |
| **a += val** |  | Yes | Yes | Same as t.fetch\_add(val) |
| **a -= val** |  | Yes | Yes | Same as t.fetch\_sub(val) |
| **++a, a++** |  | Yes | Yes | Calls t.fetch\_add(1) and returns copy of a or a+1 |
| **--a, a--** |  | Yes | Yes | Calls t.fetch\_sub(1) and returns copy of a or a-1 |
| **a.fetch\_and(val)** |  | Yes |  | Atomic a&=val (returns copy of new value) |
| **a.fetch\_or(val)** |  | Yes |  | Atomic a|=val (returns copy of new value) |
| **a.fetch\_xor(val)** |  | Yes |  | Atomic a^=val (returns copy of new value) |
| **a &= val** |  | Yes |  | Same as a.fetch\_and(val) |
| **a |= val** |  | Yes |  | Same as a.fetch\_or(val) |
| **a ^= val** |  | Yes |  | Same as a.fetch\_xor(val) |

**Note:**

·         In general, operations yield copies rather than references.

·         The default constructor does not initialize a variable/object completely.

The only legal operation after default construction is calling atomic\_init() to initialize the object.

·         The constructor for a value of the corresponding type is not atomic.

·         All functions except constructors are overloaded for volatile and non-volatile.

For atomic<int>, the following assignment operations are declared:

namespace std {    // specialization of std::atomic<> for int:

          template<> struct atomic<int> {

          public:

                   // ordinary assignment operators are not provided:

                   atomic& operator=(const atomic&) = delete;

                   atomic& operator=(const atomic&) volatile = delete;

                   // but assignment of an int is provided, which yields the passed argument:

                   int operator= (int) volatile noexcept;

                   int operator= (int) noexcept;

                   ...

          };

}

## Compare and Swap (CAS) Operations

Both compare\_exchange\_strong() and compare\_exchange\_weak() are so-called compare and swap (CAS) operations.

CPUs often provide this atomic operation to compare the contents of a memory location to a given value and, only if they are the same, modify the contents of that memory location to a given new value. This guarantees that the new value is calculated based on up-to-date information. The effect is something like the following pseudocode:

bool compare\_exchange\_strong (T& expected, T desired) {

          if (this->load() == expected) {

                   this->store(desired);

                   return true;

          }

          else {

                   expected = this->load();

                   return false;

          }

}

Thus, if the value had been updated by another thread in the meantime, it returns false with the new value in expected.

The weak form may spuriously fail so that it returns false even when the expected value is present. But the weak form is sometimes more efficient than the strong version.

# The C-Style Interface of Atomics

The whole atomic interface has a C-style equivalent, which also was proposed as an extension to the C standard.

For example, you can also declare an atomic<bool> as atomic\_bool, and instead of store() and load(), you can use global functions, which use a pointer to the object:

std::atomic\_bool ab;                 // equivalent to: std::atomic<bool> ab

std::atomic\_init(&ab, false);     //

...

std::atomic\_store(&ab,true);    // equivalent to: ab.store(true)

...

if (std::atomic\_load(&ab)) {      // equivalent to: if (ab.load())

...

}

Most important atomic type names are:

|  |  |
| --- | --- |
| **Named Type** | **Corresponding Type** |
| **atomic\_bool** | atomic<bool> |
| **atomic\_char** | atomic<char> |
| **atomic\_schar** | atomic<signed char> |
| **atomic\_uchar** | atomic<unsigned char> |
| **atomic\_short** | atomic<short> |
| **atomic\_ushort** | atomic<unsigned short> |
| **atomic\_int** | atomic<int> |
| **atomic\_uint** | atomic<unsigned int> |
| **atomic\_long** | atomic<long> |
| **atomic\_ulong** | atomic<unsigned long> |
| **atomic\_llong** | atomic<long long> |
| **atomic\_ullong** | atomic<unsigned long long> |
| **atomic\_char16\_t** | atomic<char16\_t> |
| **atomic\_char32\_t** | atomic<char32\_t> |
| **atomic\_wchar\_t** | atomic<wchar\_t> |
| **atomic\_intptr\_t** | atomic<intptr\_t> |
| **atomic\_uintptr\_t** | atomic<uintptr\_t> |
| **atomic\_size\_t** | atomic<size\_t> |
| **atomic\_ptrdiff\_t** | atomic<ptrdiff\_t> |
| **atomic\_intmax\_t** | atomic<intmax\_t> |
| **atomic\_uintmax\_t** | atomic<uintmax\_t> |

Note that for shared pointers special atomic operations are provided. The reason is that a declaration, such as atomic<shared\_ptr<T>>, is not possible, because a shared pointer is not trivially copyable. The atomic operations follow the naming conventions of the C-style interface.

# The Low-Level Interface of Atomics

The low-level interface - using the atomic operations in a way that we have no guaranteed sequential consistency. Thus, compilers and hardware might (partially) reorder access on atomics.

**Example**

the providing thread calls:

data = 42;

readyFlag = true;

and the consuming thread calls:

while (!readyFlag) { // loop until data is ready

;

}

foo(data);

The compiler and/or the hardware might reorder the statements so that effectively the following gets called:

readyFlag = true;

data = 42;

In general, such a reordering is allowed due to the rules of C++, which requires only that the observable behavior inside a thread of the generated code be correct.

**An Example for the Low-Level Interface of Atomics**

We declared an atomic flag to control access to some data:

long data;

std::atomic<bool> readyFlag(false);

and a thread providing the data:

data = 42; // provide some data

readyFlag.store(true); // and signal readiness

and a thread consuming the data:

while (!readyFlag.load()) { // loop until ready

...

}

std::cout << data << std::endl; // and process provided data

Because we use the default memory order, which guarantees sequential consistency, this works fine.

In fact, what we really call is:

data = 42;

readyFlag.store(true,std::memory\_order\_seq\_cst);

and

while (!readyFlag.load(std::memory\_order\_seq\_cst)) {

...

}

std::cout << data << std::endl;

Thus, each operation has an optional argument to pass the memory order, which by default is std::memory\_order\_seq\_cst (sequential consistent memory order).

By passing other values as memory order, we can weaken the order guarantees.

In our case, it is, for example, enough to require that the provider not delay operations past the atomic store and that the consumer not bring forward operations following the atomic load:

data = 42;

readyFlag.store(true, std::memory\_order\_release);

and

while (!readyFlag.load(std::memory\_order\_acquire)) {

...

}

std::cout << data << std::endl;

However, relaxing all constraints on the order of atomic operations would result in undefined behavior.

The reason is that std::memory\_order\_relaxed doesn’t guarantee that all prior memory operations become visible to other threads before the effect of the store operation. Thus, the provider might write data after setting the ready flag, so the consumer might read data while it gets written, which is a **data race**.

**Data atomic and std::memory\_order\_relaxed as memory order:**

std::atomic<long> data(0);

std::atomic<bool> readyFlag(false);

// providing thread:

data.store(42,std::memory\_order\_relaxed);

readyFlag.store(true,std::memory\_order\_relaxed);

// consuming thread:

while (!readyFlag.load(std::memory\_order\_relaxed)) {

...

}

std::cout << data.load(std::memory\_order\_relaxed) << std::endl;

This is not undefined behavior, because we **don’t have a data race**. However, this also would not work as expected, because the resulting value of data might not be 42 yet (the memory order is still not guaranteed). It’s behavior that results in data having an unspecified value.

 Using memory\_order\_relaxed would be useful only if we have atomic variables where reads and/or writes are independent of one another. An example would be a global counter, which different threads might increment or decrement and where we need only the final value after all threads ended.

|  |  |  |  |
| --- | --- | --- | --- |
| Operation | triv | int | ptr |
| a.store(val,mo) | Yes | Yes | Yes |
| a.load(mo) | Yes | Yes | Yes |
| a.exchange(val,mo) | Yes | Yes | Yes |
| a.compare\_exchange\_strong(exp,des,mo) | Yes | Yes | Yes |
| a.compare\_exchange\_strong(exp,des,mo1,mo2) | Yes | Yes | Yes |
| a.compare\_exchange\_weak(exp,des,mo) | Yes | Yes | Yes |
| a.compare\_exchange\_weak(exp,des,mo1,mo2) | Yes | Yes | Yes |
| a.fetch\_add(val,mo) |  | Yes | Yes |
| a.fetch\_sub(val,mo) |  | Yes | Yes |
| a.fetch\_and(val,mo) |  | Yes |  |
| a.fetch\_or(val,mo) |  | Yes |  |
| a.fetch\_xor(val,mo) |  | Yes |  |

# END