std::mem_order_consume

This lesson introduces std::mem_order_consume which is used for concurrency in C++.

Introduction

That is for two reasons that `std::memory_order_consume` is the most legendary of the six memory models: first, `std::memory_order_consume` is extremely hard to understand, and second - which may change in the future - no compiler currently supports it. With C++17 the situation gets even worse. Here is the official wording: “The specification of release-consume ordering is being revised, and the use of memory_order_consume is temporarily discouraged.”

How can it be that a compiler that implements the C++11 standard doesn’t support the memory model `std::memory_order_consume`? The answer is that the compiler maps `std::memory_order_consume` to `std::memory_order_acquire`. This is acceptable because both are load or acquire operations; `std::memory_order_consume` requires weaker synchronization and ordering constraints than `std::memory_order_acquire`. Therefore, the release-acquire ordering is potentially slower than the release-consume ordering, but - and this is the key point - it’s *well-defined*.

To get an understanding of the release-consume ordering, it is a good idea to compare it with the release-acquire ordering. I speak in the following subsection explicitly about the release-acquire ordering (not about the acquire-release semantic) to emphasize the strong relationship of `std::memory_order_consume` and `std::memory_order_acquire`.

Release-acquire Ordering

Let’s use the following program with two threads `t1` and `t2` as a starting point. `t1` plays the role of the producer, `t2` the role of the consumer. The atomic variable `ptr` helps to synchronize the producer and consumer.

```
// acquireRelease.cpp

#include <atomic>
#include <thread>
#include <iostream>
#include <string>

using namespace std;

atomic<string*> ptr;
int data;
atomic<int> atoData;

void producer(){
    string* p  = new string("C++11");
    data = 2011;
    atoData.store(2014, memory_order_relaxed);
    ptr.store(p, memory_order_release);
}

void consumer(){
    string* p2;
    while (!(p2 = ptr.load(memory_order_acquire)));
    cout << "*p2: " << *p2 << endl;
    cout << "data: " << data << endl;
    cout << "atoData: " << atoData.load(memory_order_relaxed) << endl;
}

int main(){

    cout << endl;

    thread t1(producer);
    thread t2(consumer);

    t1.join();
    t2.join();

    cout << endl;

}
```

Release-acquire vs. Release-consume ordering

Before analysing the program, I want to introduce a small variation. Replace the memory model `std::memory_order_acquire` in line 23 with `std::memory_order_consume`.

```
// acquireConsume.cpp

#include <atomic>
#include <thread>
#include <iostream>
#include <string>

using namespace std;

atomic<string*> ptr;
int data;
atomic<int> atoData;

void producer(){
    string* p  = new string("C++11");
    data = 2011;
    atoData.store(2014,memory_order_relaxed);
    ptr.store(p, memory_order_release);
}

void consumer(){
    string* p2;
    while (!(p2 = ptr.load(memory_order_consume)));
    cout << "*p2: " << *p2 << endl;
    cout << "data: " << data << endl;
    cout << "atoData: " << atoData.load(memory_order_relaxed) << endl;
}

int main(){

    cout << endl;

    thread t1(producer);
    thread t2(consumer);

    t1.join();
    t2.join();

    cout << endl;

}
```

Now the program has undefined behavior. This statement is very hypothetical because my GCC 5.4 compiler implements `std::memory_order_consume` using `std::memory_order_acquire`. Under the hood, both programs actually do the same thing.

The outputs of the programs are identical. At the risk of repeating myself, I want to add a few words explaining why the first program `acquireRelease.cpp` is well-defined.

The store operation in line 17 *synchronizes-with* the load operation in line 23. The reason is that the store operation uses `std::memory_order_release` and the load operation uses `std::memory_order_acquire`. This is the synchronization. What are the ordering constraints for the release-acquire operations? The release-acquire ordering guarantees that the results of all operations before the store operation (line 17) are available after the load operation (line 23). So, in addition, the release-acquire operation orders the access on the non-atomic variable `data` (line 11) and the atomic variable `atoData` (line 12). That holds, although `atoData` uses the `std::memory_order_relaxed` memory model.

Here's the key question: what happens if I replace `std::memory_order_acquire` with `std::memory_order_consume`?