

C++ Memory Model

Since C++11, C++ has a memory model. It is the foundation for multithreading. Without it, multithreading is not well defined.

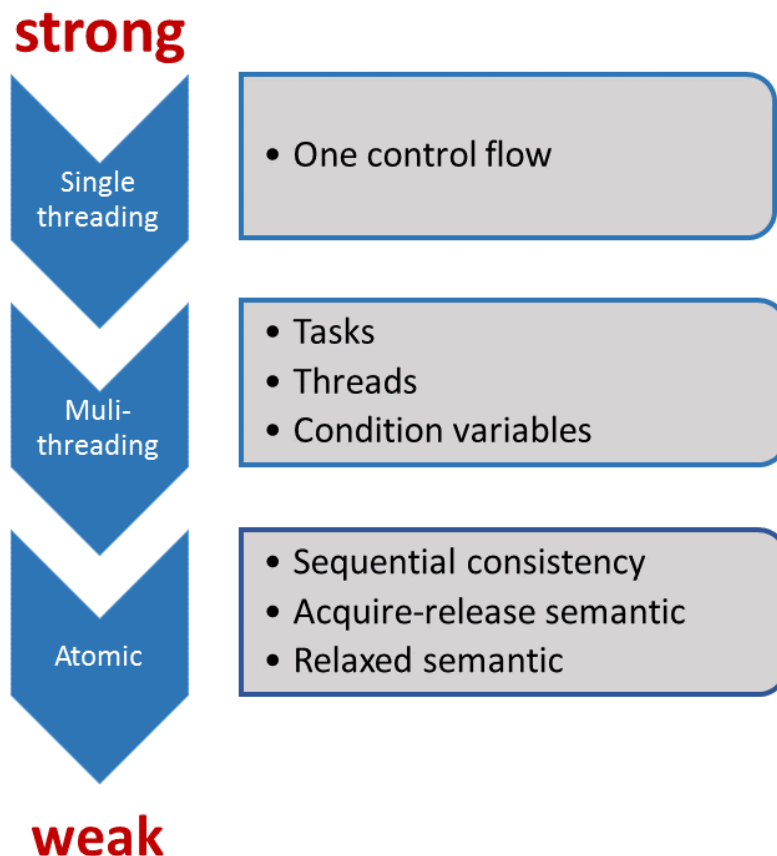
The C++ memory model consists of two aspects. On one hand, there is the enormous complexity of the memory model, which often contradicts our intuition. On the other hand, the memory model helps a lot to get a deeper insight into the multithreading challenges.

The contract

In the first approach, the C++ memory model defines a contract. This contract is established between the programmer and the system. The system consists of the compiler, which compiles the program into assembler instructions, the processor, which performs the assembler instructions and the different caches, which stores the state of the program. The contract requires from the programmer to obey certain rules and gives the system the full power to optimise the program as far as no rules are broken. The result is - in the good case - a well-defined program, that is maximal optimised. Precisely spoken, there is not only a single contract, but a fine-grained set of contracts. Or to say it differently. The weaker the rules are the programmer has to follow, the more potential is there for the system to generate a highly optimised executable.

The rule of thumb is quite easy. The stronger the contract, the fewer liberties for the system to generate an optimised executable. Sadly, the other way around will not work. In case the programmer uses an extremely weak contract or memory model, there are a lot of optimisation choices. But the program is only manageable by a few worldwide known experts.

There are three levels of the contract in C++11.



Before C++11 there was only one contract. C++ was not aware of the existence of multithreading or atomics. The system only knows about one control flow and therefore there were only restricted opportunities to optimise the executable. The key point of the system was it, to keep the illusion for the programmer, that the observed behaviour of the program corresponds to the sequence of the instructions in the source code. Of course, there was no memory model. Instead of that, there was the concept of a sequence point. Sequence points are points in the program, at which the effects of all instructions before must be observable. The start or the end of the execution of a function are sequence points. But in case you invoke a function with two arguments, the

C++ standard makes no guarantee, which arguments will be evaluated at first. So the behaviour is unspecified. The reason is straightforward. The comma operator is no sequence point. That will not change in C++11.

But with C++ all will change. C++11 is the first time aware of multiple threads. The reason for the well-defined behaviour of threads is the C++ memory model. The C++ memory model is inspired by the Java memory model, but the C++ one goes - as ever - a few steps further. But that will be a topic of the next posts. So the programmer has to obey to a few rules in dealing with shared variables to get a well-defined program. The program is undefined if there exists at least one data race. As I already mentioned, you have to be aware of data races, if your threads share mutable data. So tasks are a lot easier to use than threads or condition variables.

With atomics, we enter the domain of the experts. This will become more evident, the further we weaken the C++ memory model. Often, we speak about lock-free programming, when we use atomics. I spoke in the posts about the weak and strong rules. Indeed, the sequential consistency is called strong memory model, the relaxed semantic weak memory model.

The meat of the contract

The contract between the programmer and the system consists of three parts:

- **Atomic operations:** Operations, which will be executed without interruption.
- **The partial order of operations:** Sequence of operations, which can not be changed.
- **Visible effects of operations:** Guarantees, when an operation on shared variables will be visible in another thread.

The foundation of the contract are operations on atomics. These operations have two characteristics. They are atomic and they create synchronisation and order constraints on the program execution. These synchronisations and order constraints will often also hold for not atomic operations. At one hand an atomic operation is always atomic, but on the other hand, you can tailor the synchronisations and order constraints to your needs.

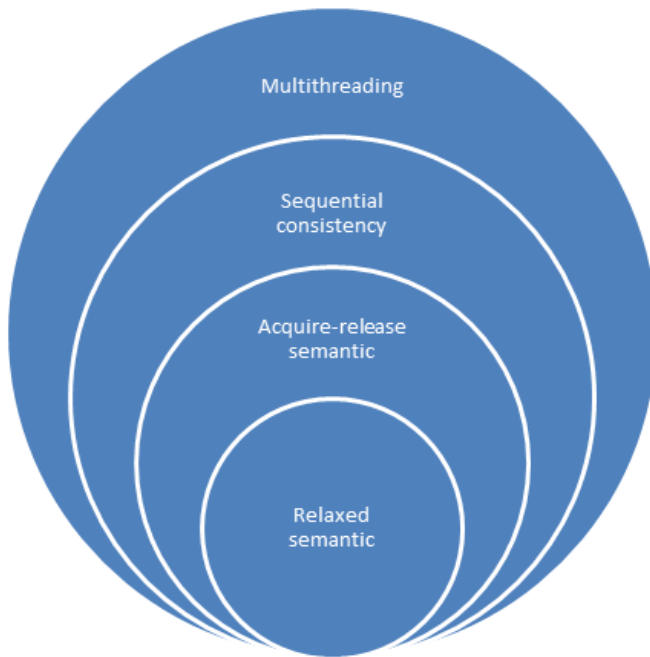
Back to the big picture

The more we weaken the memory model, the more our focus will change.

- More optimisation potential for the system
- The number of control flows of the program increases exponential
- Domain for the experts
- Break of the intuition
- Area for micro optimisation

To make multithreading, we should be an expert. In case we want to deal with atomics (sequential consistency), we should open the door to the next expertise level. And you know, what will happen when we talk about the acquire-release or relaxed semantic? We'll go each time one step higher to the next expertise level.

Expert levels



Sequential Consistency

The atomics are the base of the C++ memory model. Per default, sequential consistency is applied.

The strong C++ memory model

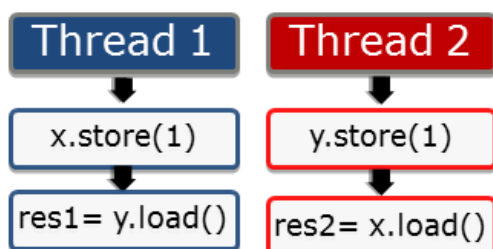
In 2004 Java 5.0 gets its current [memory model](#), in 2011 C++. Before that, Java had an erroneous memory model, C++ had no memory model. Who thinks, that this is the endpoint of a long process, is totally wrong. The foundations of multithreading programming are 40 to 50 years old. So [Leslie Lamport](#) defined 1979 the concept of sequential consistency.

Sequential consistency provides two guarantees.

1. The instructions of a program are executed in source code order.
2. There is a global order of all operations on all threads.

Before I look deeper in these two guarantees, I explicitly emphasise. The statements only hold for **atomics** but influence **non-atomics**.

The simple graphic displays two threads. Each thread is storing its variable `x` or `y`, loads the other variable `y` and `x` and stores them in the variable `res1` or `res2`.

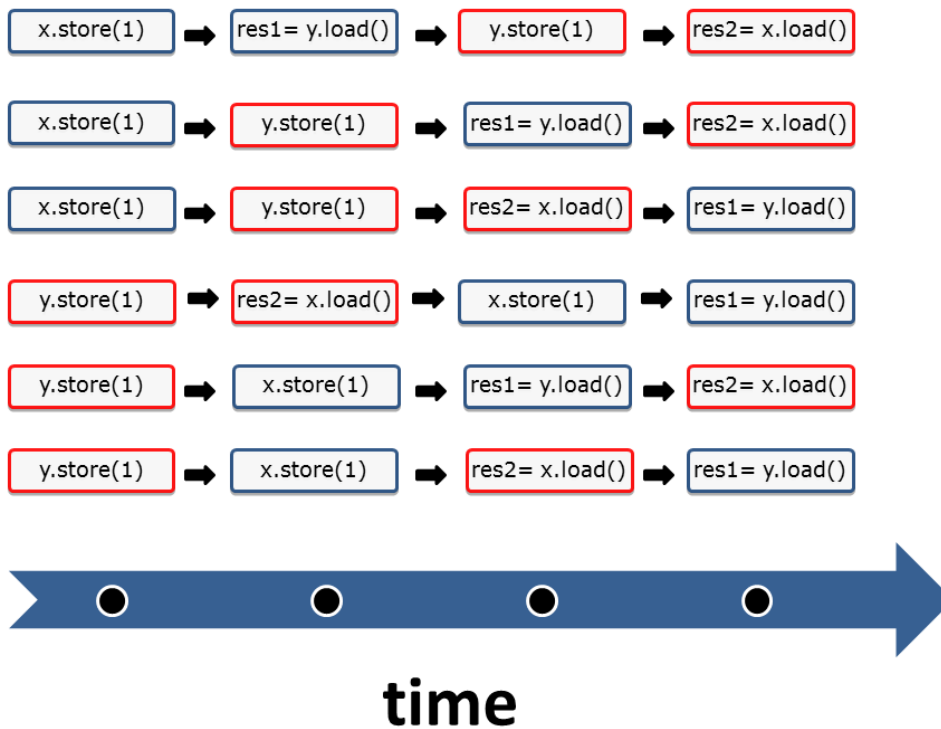


The operations take place on atomics and so, they are atomic. By default, sequential consistency applies. But the question is. In which order can the statements take place?

The first guarantee of the sequential consistency is that the instruction will be executed in the order of the source code. That is easy. No store operation can overtake a load operation.

The second guarantee of the sequential consistency is, that all instructions of all threads have to follow a global order. That means in that concrete case, that thread 2 sees the operations of thread 1 in the same order, in which thread 1 executes them. This is the key observation. **Thread 2 sees all operations of thread 1 in the source code order of thread 1.** The same holds from the perspective of thread 1. So you can think about characteristic 2 as a global counter, which all threads have to obey. The global counter is the global order.

We're not done with our riddle right now. What is still missing, is to look at the different interleaving executions of the two threads. So the following six interleavings of the two threads are possible.



That was easy. Or?

From the strong to the weak memory model

I want once more refer to the [picture](#) of the contract between the programmer and the system.

The programmer uses atomics in this particular example. So he obeys his part of the contract by using them in the right way. The system guarantees him a well-defined program behaviour without data races. In addition to that, the system can execute the four operations in each combination. In case the programmer uses the relaxed semantic, the pillars of the contract dramatically changes. On one hand, it is a lot more difficult for the programmer to apply the contract in the right way. On the other hand, the system has a lot more optimisation possibilities. With the relaxed semantic - also called weak memory model - there are a lot more combinations of the four operations possible. The counter-intuitive behaviour is, that the thread 1 can see the operations of thread 2 in a different order. So there is no picture of a global counter. From the perspective of thread 1 it is possible, that the operation `res = y.load()` overtake `x.store()`.

Between the sequential consistency and the relaxed-semantic, there are a few more models. The most important one is the acquire-release semantic. I think you already guess it. With acquire-release semantic, the programmer has to obey weaker rules than with sequential consistency. But the system has more optimisation possibilities. The acquire-release semantic is the key for a deeper understanding of the multithreading programming because the threads will be synchronised at specific synchronisation points in the code. Without these synchronisation points, there is no well-defined behaviour of threads, tasks or condition variables possible. More about that in the following post.

The Atomic Flag

Atomics guarantee two characteristics. At one hand, they are atomic, at the other hand, they provide synchronisation and order constraints on the program execution.

I introduced in the last post the sequential consistency as the default behaviour of atomic operations. But, what does that mean? You can specify for each atomic operation the memory order. If not specified, `std::memory_order_seq_cst` is used.

So this piece of code

```
x.store(1);
res= x.load();
```

is equivalent to the following piece of code.

```
x.store(1, std::memory_order_seq_cst);
res= x.load(std::memory_order_seq_cst);
```

For simplicity reasons, I will use the first spelling in this post.

std::atomic_flag

`std::atomic_flag` has a simple interface. Its method `clear` enables you to set its value to `false`, with `test_and_set` back to `true`. In case you use `test_and_set` you get the old value back. To use `std::atomic_flag` it must be initialized to `false` with the constant `ATOMIC_FLAG_INIT`. That is not so thrilling. But `std::atomic_flag` has two very interesting properties.

`std::atomic_flag` is

- the only lock-free atomic.
- the building block for higher thread abstractions.

The only lock-free atomic? The remaining more powerful atomics can provide their functionality by using a [mutex](#). That is according to the C++ standard. So these atomics have a method `is_lock_free` to check if the atomic uses internally a mutex. On the popular platforms, I always get the answer `false`. But you should be aware of that.

The interface of a `std::atomic_flag` is sufficient to build a spinlock. With a spinlock, you can protect a [critical section](#) similar to a mutex. But the spinlock will not passively wait in opposite to a mutex until it gets it mutex. It will eagerly ask for the critical section. So it saves the expensive context change in the wait state but it fully utilises the CPU:

The example shows the implementation of a spinlock with the help of `std::atomic_flag`.

```

1 // spinLock.cpp
2
3 #include <atomic>
4 #include <thread>
5
6 class Spinlock{
7     std::atomic_flag flag;
8 public:
9     Spinlock(): flag(ATOMIC_FLAG_INIT) {}
10
11     void lock(){
12         while( flag.test_and_set() );
13     }
14
15     void unlock(){
16         flag.clear();
17     }
18 };
19
20 Spinlock spin;
21
22 void workOnResource(){
23     spin.lock();
24     // shared resource
25     spin.unlock();
26 }
27
28
29 int main(){
30
31     std::thread t(workOnResource);
32     std::thread t2(workOnResource);
33
34     t.join();
35     t2.join();
36
37 }

```

The both threads `t` and `t2` (line 31 and 32) are fighting for the critical section. For simplicity reasons the critical section in line 24 consists only of a comment. How does it work? The class `Spinlock` has - similar to a mutex - the two methods `lock` and `unlock`. In addition to that the constructor of `Spinlock` initializes in line 9 the `std::atomic_flag` to `false`. In case thread `t` wants to execute the function `workOnResource` two scenarios can happen.

First, the thread `t` gets the lock. So the lock invocation was successful. The lock invocation is successful, if the initial value of the flag in line 12 is `false`. In this case thread `t` sets it in an atomic operation to `true`. That value `true` is the value, the while loop returns to the other thread `t2`, if it tries to get the lock. So thread `t2` is caught in the rat race. Thread `t2` has no possibility to set the value of the flag to `false`. So `t2` must eagerly wait until thread `t1` executes the `unlock` method and sets the flag to `false` (line 15 - 17).

Second, the thread `t` don't get the lock. So we are in scenario 1 with changed roles.

It's very interesting to compare the active waiting of a spinlock with the passive waiting of a mutex.

Spinlock versus Mutex

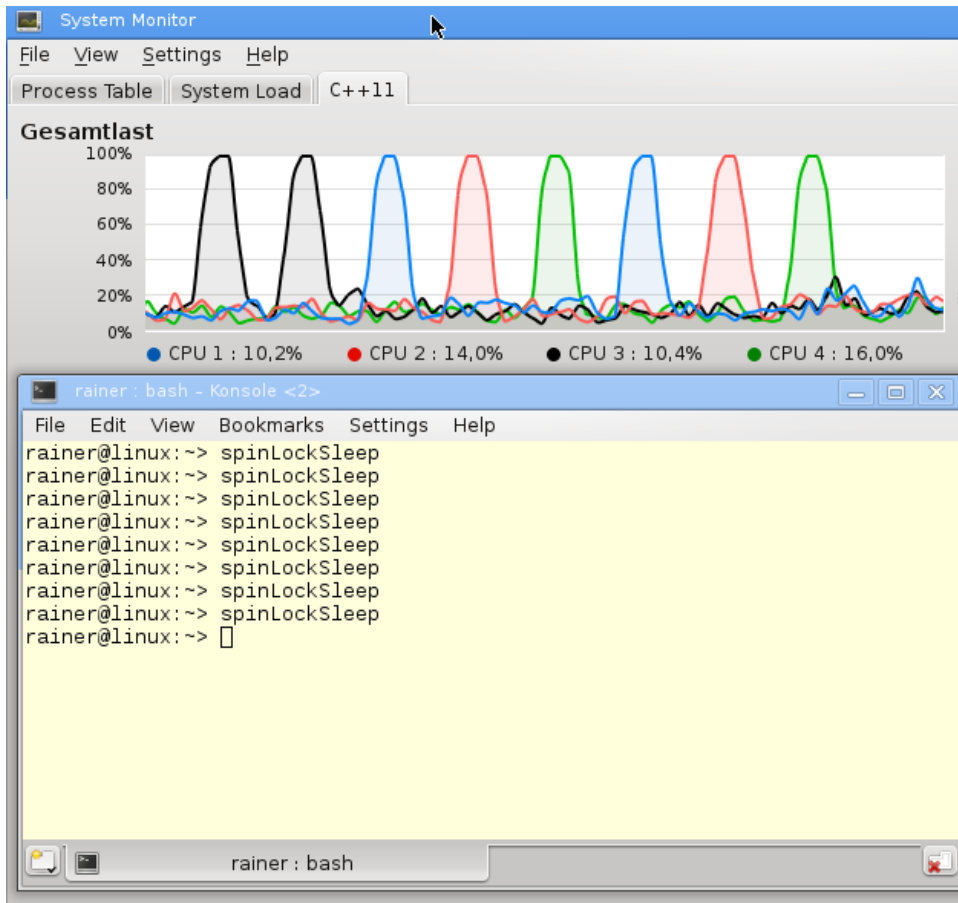
What is happening to the CPU load, if the function `workOnResource` locks the spinlock for 2 seconds (line 23 - 25)?

```

1 // spinLockSleep.cpp
2
3 #include <atomic>
4 #include <thread>
5
6 class Spinlock{
7     std::atomic_flag flag;
8 public:
9     Spinlock(): flag(ATOMIC_FLAG_INIT) {}
10
11     void lock(){
12         while( flag.test_and_set() );
13     }
14
15     void unlock(){
16         flag.clear();
17     }
18 };
19
20 Spinlock spin;
21
22 void workOnResource(){
23     spin.lock();
24     std::this_thread::sleep_for(std::chrono::milliseconds(2000));
25     spin.unlock();
26 }
27
28
29 int main(){
30
31     std::thread t(workOnResource);
32     std::thread t2(workOnResource);
33
34     t.join();
35     t2.join();
36
37 }

```

In case the theory is right, one of the four cores of my PC must be fully utilized. Exactly that you can see in the screenshot.



The screenshot shows in a nice way, that the load of one core gets 100%. But my PC is fair. Each time a different core has to

perform the busy waiting.

I use in the next concise program a mutex instead of a spinlock.

```
#include <mutex>
#include <thread>

std::mutex mut;

void workOnResource() {
    mut.lock();
    std::this_thread::sleep_for(std::chrono::milliseconds(5000));
    mut.unlock();
}

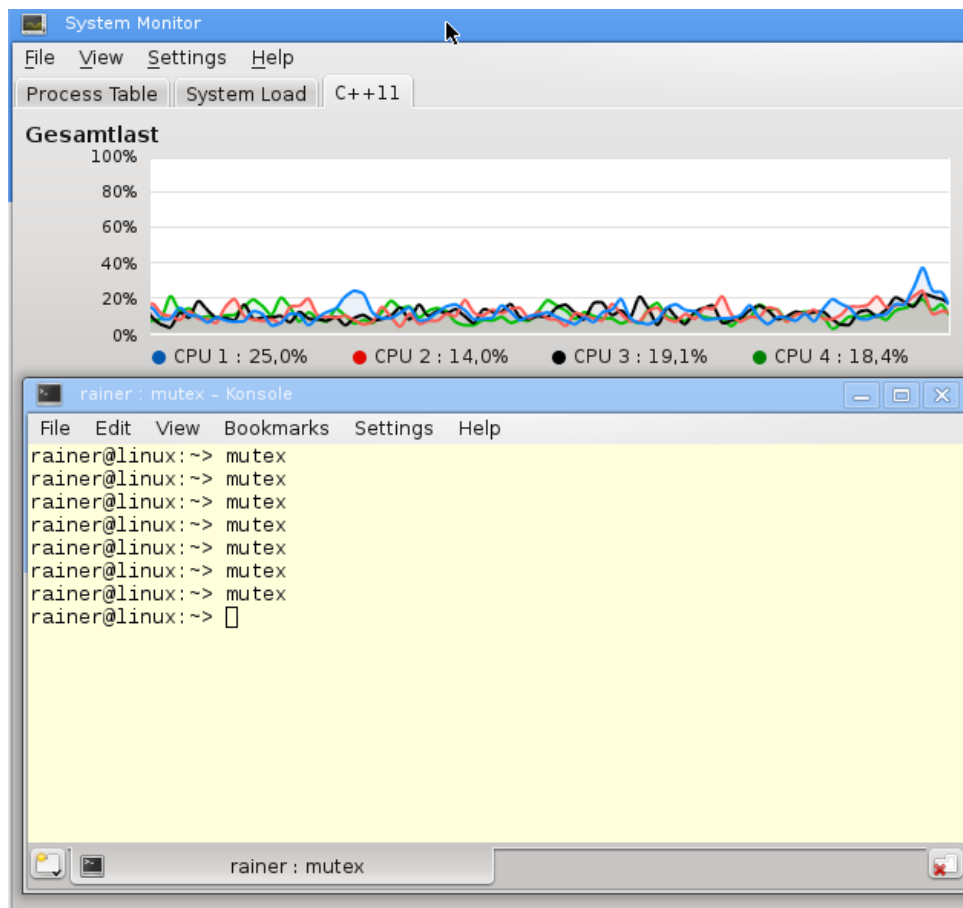
int main() {

    std::thread t(workOnResource);
    std::thread t2(workOnResource);

    t.join();
    t2.join();

}
```

Although I execute the program several times, I can not observe a higher load of the cores.



The Atomic Boolean

The remaining atomics - in contrast to `std::atomic_flag` - are partial or full specialisations of the class template `std::atomic`. Let's start with `std::atomic<bool>`.

std::atomic<bool>

std::atomic<bool> has a lot more to offer than std::atomic_flag. It can explicitly be set to true or false. That's enough to synchronise two threads. So I can simulate [condition variables](#) with atomic variables.

Let's first have a look at condition variables.

```
1 // conditionVariable.cpp
2
3 #include <condition_variable>
4 #include <iostream>
5 #include <thread>
6 #include <vector>
7
8 std::vector<int> mySharedWork;
9 std::mutex mutex_;
10 std::condition_variable condVar;
11
12 bool dataReady;
13
14 void waitingForWork() {
15     std::cout << "Waiting " << std::endl;
16     std::unique_lock<std::mutex> lck(mutex_);
17     condVar.wait(lck, [] {return dataReady;});
18     mySharedWork[1] = 2;
19     std::cout << "Work done " << std::endl;
20 }
21
22 void setDataReady() {
23     mySharedWork = {1, 0, 3};
24     {
25         std::lock_guard<std::mutex> lck(mutex_);
26         dataReady = true;
27     }
28     std::cout << "Data prepared" << std::endl;
29     condVar.notify_one();
30 }
31
32 int main() {
33
34     std::cout << std::endl;
35
36     std::thread t1(waitingForWork);
37     std::thread t2(setDataReady);
38
39     t1.join();
40     t2.join();
41
42     for (auto v: mySharedWork) {
43         std::cout << v << " ";
44     }
45
46
47     std::cout << "\n\n";
48
49 }
```

And now the pendant with atomic booleans.

```

1 // atomicCondition.cpp
2
3 #include <atomic>
4 #include <chrono>
5 #include <iostream>
6 #include <thread>
7 #include <vector>
8
9 std::vector<int> mySharedWork;
10 std::atomic<bool> dataReady(false);
11
12 void waitingForWork() {
13     std::cout << "Waiting " << std::endl;
14     while ( !dataReady.load() ) { // (3)
15         std::this_thread::sleep_for(std::chrono::milliseconds(5));
16     }
17     mySharedWork[1] = 2; // (4)
18     std::cout << "Work done " << std::endl;
19 }
20
21 void setDataReady() {
22     mySharedWork = {1, 0, 3}; // (1)
23     dataReady = true; // (2)
24     std::cout << "Data prepared" << std::endl;
25 }
26
27 int main() {
28
29     std::cout << std::endl;
30
31     std::thread t1(waitingForWork);
32     std::thread t2(setDataReady);
33
34     t1.join();
35     t2.join();
36
37     for (auto v: mySharedWork) {
38         std::cout << v << " ";
39     }
40
41
42     std::cout << "\n\n";
43
44 }

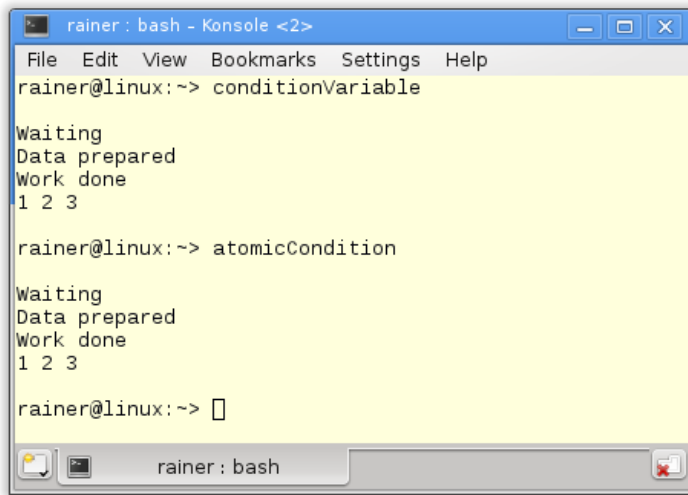
```

What guarantees, that line 17 will be executed after the line 14? Or to say it more general, that the thread t1 will execute `mySharedWork[1] = 2` (line 17) after thread t2 had executed `mySharedWork = {1, 0, 3}` (line 22). Now it gets more formal.

- Line 22 (1) *happens-before* line 23 (2)
- Line 14 (3) *happens-before* line 17 (4)
- Line 23 (2) *synchronizes-with* line 14 (3)
- Because *happens-before* is transitive, it follows: `mySharedWork = {1, 0, 3}` (1) *happens-before* `mySharedWork[1] = 2` (4)

In want to explicitly mention one point. Because of the condition variable `condVar` or the atomic `dataReady`, the access to the shared variable `mySharedWork` is synchronised. This holds although `mySharedWork` is not protected by a lock or itself an atomic.

Both programs produce the same result for `mySharedWork`.



```
rainer: bash - Konsole <2>
File Edit View Bookmarks Settings Help
rainer@linux:~> conditionVariable

Waiting
Data prepared
Work done
1 2 3

rainer@linux:~> atomicCondition

Waiting
Data prepared
Work done
1 2 3

rainer@linux:~> []
```

Push versus pull principle

Obviously, I cheated a little. There is one difference between the synchronisation of the threads with the condition variable and the atomic boolean. The condition variable notifies the waiting thread (`condVar.notify()`), that it should proceed with its work. But the waiting thread with the atomic boolean checks, if the sender is done with its work (`dataRead= true`).

The condition variable notifies the waiting thread (push principle). The atomic boolean repeatedly asks for the value (pull principle).

compare_exchange_strong and compare_exchange_weak

`std::atomic<bool>` and the fully or partially specialisations of `std::atomic` supports the bread and butter of all atomic operations: `compare_exchange_strong`. This function has the syntax: `bool compare_exchange_strong(T& expected, T& desired)`. Because this operation compares and exchanges in one atomic operation a value, is often called `compare_and_swap` (CAS). This kind of operation is in a lot of programming languages available. Of course, the behaviour may differ a little.

A call of `atomicValue.compare_exchange_strong(expected, desired)` obeys the following strategy. In case the atomic comparison of `atomicValue` with `expected` returns `true`, the value of `atomicValue` is set in the same atomic operation to `desired`. If the comparison returns `false`, `expected` will be set to `atomicValue`. The reason why the operation `compare_exchange_strong` is called strong is simple. There is a method `compare_exchange_weak`. This weak version can spuriously fail. That mean, although `*atomicValue == expected` holds, the weak variant returns `false`. So you have to check the condition in a loop: `while (!atomicValue.compare_exchange_weak(expected, desired))`. The reason for the weak form is performance. On some platforms, the weak is faster than the strong variant.

Atomics

In addition to booleans, there are atomics for pointers, integrals and user defined types. The rules for user-defined types are special.

Both. The atomic wrapper on a pointer `T*` `std::atomic<T*>` or on an integral type `integ` `std::atomic<integ>` enables the CAS (compare-and-swap) operations.

`std::atomic<T*>`

The atomic pointer `std::atomic<T*>` behaves like a plain pointer `T*`. So `std::atomic<T*>` supports pointer arithmetic and pre- and post-increment or pre- and post-decrement operations. Have a look at the short example.

```
int intArray[5];
std::atomic<int*> p(intArray);
p++;
assert(p.load() == &intArray[1]);
p+=1;
assert(p.load() == &intArray[2]);
--p;
assert(p.load() == &intArray[1]);
```

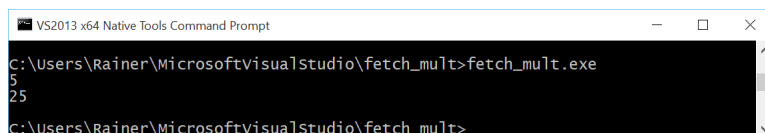
`std::atomic<integral type>`

In C++11 there are atomic types to the known integral data types. As ever you can read the whole stuff about atomic integral data types - including there operations - on the page en.cppreference.com. A `std::atomic<integral type>` allows all, what a `std::atomic_flag` or a `std::atomic<bool>` is capable of, but even more.

The composite assignment operators `+=`, `-=`, `&=`, `|=` and `^=` and there pedants `std::atomic<>::fetch_add()`, `std::atomic<>::fetch_sub()`, `std::atomic<>::fetch_and()`, `std::atomic<>::fetch_or()` and `std::atomic<>::fetch_xor()` are the most interesting ones. There is a little difference in the atomic read and write operations. The composite assignment operators return the new value, the fetch variations the old value. A deeper look gives more insight. There is no multiplication, division and shift operation in an atomic way. But that is not that big restriction. Because these operations are relatively seldom needed and can easily be implemented. How? Look at the example.

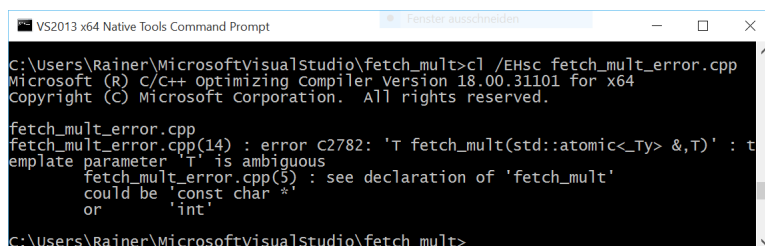
```
1 // fetch_mult.cpp
2
3 #include <atomic>
4 #include <iostream>
5
6 template <typename T>
7 T fetch_mult(std::atomic<T>& shared, T mult){
8     T oldValue= shared.load();
9     while (!shared.compare_exchange_strong(oldValue, oldValue * mult));
10    return oldValue;
11 }
12
13 int main(){
14     std::atomic<int> myInt{5};
15     std::cout << myInt << std::endl;
16     fetch_mult(myInt,5);
17     std::cout << myInt << std::endl;
18 }
```

I should mention one point. The addition in line 9 will only happen, if the relation `oldValue == shared` holds. So to be sure that the multiplication will always take place, I put the multiplication in a `while` loop. The result of the program is not so thrilling.



```
VS2013 x64 Native Tools Command Prompt
C:\Users\Rainer\MicrosoftVisualStudio\fetch_mult>fetch_mult.exe
25
C:\Users\Rainer\MicrosoftVisualStudio\fetch_mult>
```

The implementations of the function template `fetch_mult` is generic, too generic. So you can use it with an arbitrary type. In case I use instead of the number 5 the C-String 5, the Microsoft compilers complains that the call is ambiguous.



```
VS2013 x64 Native Tools Command Prompt
C:\Users\Rainer\MicrosoftVisualStudio\fetch_mult>cl /EHsc fetch_mult_error.cpp
Microsoft (R) C/C++ Optimizing Compiler Version 18.00.31101 for x64
Copyright (C) Microsoft Corporation. All rights reserved.

fetch_mult_error.cpp
fetch_mult_error.cpp(14) : error C2782: 'T fetch_mult(std::atomic<Ty> &,T)' : t
template parameter 'T' is ambiguous
        fetch_mult_error.cpp(5) : see declaration of 'fetch_mult'
        could be 'const char *'
        or
        'int'
C:\Users\Rainer\MicrosoftVisualStudio\fetch_mult>
```

"5" can be interpreted as a `const char*` or as an `int`. That was not my intention. The template argument should be an integral type. The right use case for concepts lite. With concepts lite, you can express constraints to the template parameter. Sad to say but they will not be part of C++17. We should hope for C++20 standard.

```
1 template <typename T>
2     requires std::is_integral<T>::value
3 T fetch_mult(std::atomic<T>& shared, T mult){
4     T oldValue= shared.load();
5     while (!shared.compare_exchange_strong(oldValue, oldValue * mult));
6     return oldValue;
7 }
```

The predicate `std::is_integral<T>::value` will be evaluated by the compiler. If T is not an integral type, the compiler will complain. `std::is_integral` is a function of the new [type-traits library](http://en.cppreference.com), which is part of C++11. The requires condition in line 2 defines the constraints on the template parameter. The compiler checks the contract at compile time.

You can define your own atomic types.

`std::atomic<user defined type>`

There are a lot of serious restrictions on a user defined type to get an atomic type `std::atomic<MyType>`. These restrictions are on the type, but these restrictions are on the available operations that `std::atomic<MyType>` can perform.

For `MyType` there are the following restrictions:

- The copy assignment operator for `MyType`, for all base classes of `MyType` and all non-static members of `MyType` must be trivial. Only an automatically by the compiler generated copy assignment operator is trivial. To say it the other way around. User defined copy assignment operators are not trivial.
- `MyType` must not have virtual methods or base classes.
- `MyType` must be bitwise comparable so that the C functions `memcpy` or `memcmp` can be applied.

You can check the constraints on `MyType` with the function `std::is_trivially_copy_constructible`, `std::is_polymorphic` and `std::is_trivial` at compile time. All the functions are part of the [type-traits library](#).

For the user defined type `std::atomic<MyType>` only a reduced set of operations is supported.

Atomic operations

To get the great picture, I displayed in the following table the atomic operations dependent on the atomic type.

Operation	<code>atomic_flag</code>	<code>atomic<bool></code>	<code>atomic<T*></code>	<code>atomic<integral></code>	<code>atomic<user defined></code>
<code>test_and_set</code>	✓				
<code>clear</code>	✓				
<code>is_lock_free</code>	✓	✓	✓	✓	✓
<code>load</code>		✓	✓	✓	✓
<code>store</code>		✓	✓	✓	✓
<code>exchange</code>		✓	✓	✓	✓
<code>compare_exchange_weak</code> <code>compare_exchange_strong</code>		✓	✓	✓	✓
<code>fetch_add, +=</code> <code>fetch_sub, -=</code>			✓	✓	
<code>fetch_or, =</code> <code>fetch_and, &=</code> <code>fetch_xor, ^=</code>				✓	
<code>++</code> <code>--</code>			✓	✓	

Free atomic functions and smart pointers

The functionality of the class templates `std::atomic` and the Flag `std::atomic_flag` can be used as a [free function](#). Because the free functions use atomic pointers instead of references they are compatible with C. The atomic free functions support the same types as the class template `std::atomic` but in addition to that the smart pointer `std::shared_ptr`. That is special because of `std::shared_ptr` is not an atomic data type. The C++ committee recognised the necessity, that instances of smart pointers that maintain under their hood the reference counters and object must be modifiable in an atomic way.

```
std::shared_ptr<MyData> p;  
std::shared_ptr<MyData> p2= std::atomic_load(&p);  
std::shared_ptr<MyData> p3(new MyData);  
std::atomic_store(&p, p3);
```

To be clear. **The atomic characteristic will only hold for the reference counter, but not for the object.** That was the reason, we get a `std::atomic_shared_ptr` in the future (I'm not sure if the future is called C++17 or C++20. I was often wrong in the past.), which is based on a `std::shared_ptr` and guarantees the atomicity of the underlying object. That will also hold for `std::weak_ptr`. `std::weak_ptr`, which is a temporary owner of the resource, helps to break cyclic dependencies of `std::shared_ptr`. The name of the new atomic `std::weak_ptr` will be `std::atomic_weak_ptr`. To make the picture complete, the atomic version of `std::unique_ptr` is called `std::atomic_unique_ptr`.

Synchronization and Ordering Constraints

In this post, our tour through the c++ memory model goes one step deeper. Until now, the posts were only about the atomicity of the atomic data types but now we deal with the synchronisation and ordering constraints of the operations.

You can not configure the atomicity of an atomic data type, but you can adjust very accurately the synchronisation and ordering constraints of atomic operations. A leverage, which is unique to C++. That's not possible in the C#'s or Java's memory model.

The six variants of the C++ memory model

C++ has six variants of the memory model. The default for atomic operations is `std::memory_order_seq_cst`. But you can explicitly specify one of the other five. But what has C++11 to offer?

```
enum memory_order{
    memory_order_relaxed,
    memory_order_consume,
    memory_order_acquire,
    memory_order_release,
    memory_order_acq_rel,
    memory_order_seq_cst
}
```

It helps a lot to answer two questions, to get a system into the six memory models.

1. For which type of atomic operations should you use the memory model?
2. Which synchronisation and ordering constraints are defined by the memory model?

The rest of this post is about answering these questions. So what are the types of atomic operations?

Types of atomic operations

The memory model deals with reading and/or writing atomic operations.

- **read operation:** `memory_order_acquire` and `memory_order_consume`
- **write operation:** `memory_order_release`
- **read-modify-write operation:** `memory_order_acq_rel` and `memory_order_seq_cst`

`memory_order_relaxed` defines no synchronisation and ordering constraints. So it will not fit in this taxonomy.

The table orders the atomic operations based on their reading and/or writing characteristics.

Operation	read operation	write operation	read-modify-write operation
test_and_set			✓
clear		✓	
is_lock_free	✓		
load	✓		
store		✓	
exchange			✓
compare_exchange_weak compare_exchange_strong			✓
fetch_add, += fetch_sub, -=			✓
fetch_or, = fetch_and, &= fetch_xor, ^=			✓
++ --			✓

In case you use an atomic operation `atomVar.load(5)` with a memory model, that is designed for a write or read-modify-write operation, the write part has no effect. So an `atomVar.load(5, std::memory_order_acq_rel)` is equivalent to an `atomVar.load(5, std::memory_order_acquire)`, an `atomVar.load(5, std::memory_order_release)` is equivalent to an `atomVar.load(5, std::memory_order_relaxed)`.

The different synchronisation and ordering constraints

There are three different types synchronization and ordering constraints in C++11:

- **Sequential consistency:** `memory_order_seq_cst`
- **Acquire-release:** `memory_order_consume`, `memory_order_acquire`, `memory_order_release` and `memory_order_acq_rel`
- **Relaxed:** `memory_order_relaxed`

While the sequential consistency establishes a global order between threads, the acquire-release semantic establishes an ordering between read and write operations on the same atomic variable on different threads. The relaxed semantic only guarantees that operations on the same atomic data type in the same thread can not be reordered. That guarantee is called [modification order consistency](#). But other threads can see this operation in a different order.

Sequential Consistency Applied

I have introduced In the post [Sequential Consistency](#) the default memory model. This model, in which all operations in all threads takes place in a global time clock, has a big advantage but also a big disadvantage.

Heavyweight synchronization

The big advantage of the sequential consistency is, that it matches our intuition of many threads running in parallel. The big disadvantages is, that the system has a lot of work to do to synchronise all the threads.

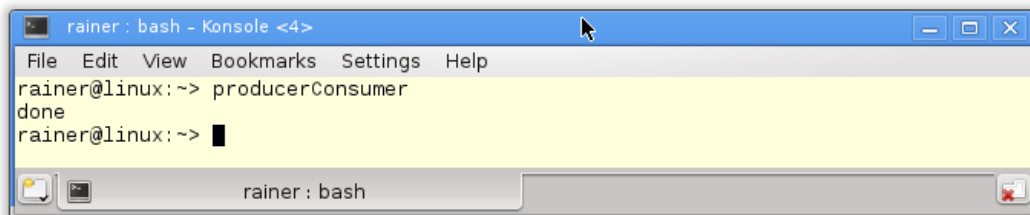
The following program synchronises the producer and the consumer thread with the help of the sequential consistency.

```

1 // producerConsumer.cpp
2
3 #include <atomic>
4 #include <iostream>
5 #include <string>
6 #include <thread>
7
8 std::string work;
9 std::atomic<bool> ready(false);
10
11 void consumer(){
12     while(!ready.load()){ }
13     std::cout<< work << std::endl;
14 }
15
16 void producer(){
17     work= "done";
18     ready=true;
19 }
20
21 int main(){
22     std::thread prod(producer);
23     std::thread con(consumer);
24     prod.join();
25     con.join();
26 }

```

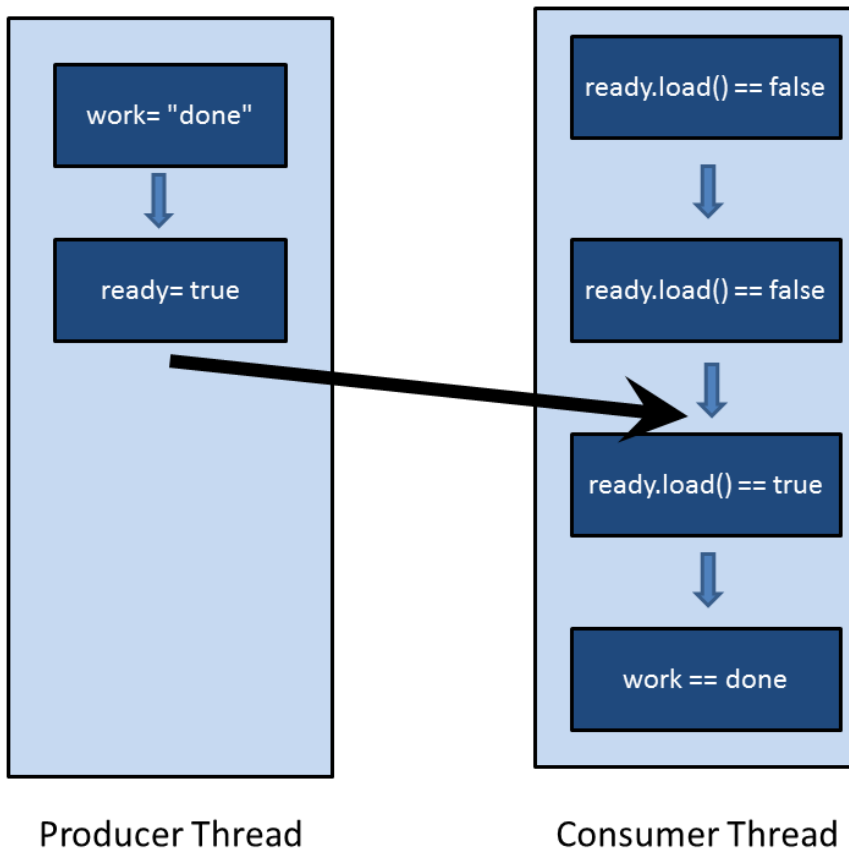
The output of the program is concise and interesting.



The screenshot shows a terminal window titled "rainer : bash - Konsole <4>". The window has a menu bar with "File", "Edit", "View", "Bookmarks", "Settings", and "Help". The terminal content shows the command "producerConsumer" being executed, which results in the output "done". The prompt "rainer@linux:~>" is visible before and after the command. The terminal window has a yellow background and a grey border.

Because of the sequential consistency, the program execution is totally deterministic. It always displays "done".

The graphic hit the spot. The consumer thread waits in the while-loop until the atomic variable `ready` is set to `true`. In case that happens, the consumer threads continue with its work.



It easy to reason, that the program will always return "done". I have only to use the two characteristics of the sequential consistency. At one hand, both threads execute their instructions in the source code order, at the other hand, each thread sees the operations of the other thread in the same order. So both threads are following the same global time clock. This time clock will also hold - with the help of the `while(!ready.load()) {}` -loop - for the synchronisation of the producer and the consumer thread.

But I can do the reasoning a lot more formal by using the terminology of the memory model. So the formal version:

=> Means it follows in the next lines:

1. `work= "done"` is **sequenced-before** `ready=true` **=>** `work= "done"` **happens-before** `ready=true`
2. `while(!ready.load()) {}` is **sequenced-before** `std::cout<< work << std::endl` **=>**
`while(!ready.load()) {}` **happens-before** `std::cout<< work << std::endl`
3. `ready= true` **synchronizes-with** `while(!ready.load()) {}` **=>** `ready= true` **inter-thread happens-before**
`while (!ready.load()) {}` **=>** `ready= true` **happens-before** `while (!ready.load()) {}`

=> Because the happens-before relation is transitive, it follows `work= "done"` **happens-before** `ready= true` **happens-before** `while(!ready.load()) {}` **happens-before** `std::cout<< work << std::endl`

From the sequential consistency to the acquire-release semantic

A thread sees the operations of another thread and therefore of all other threads in the same order. The key characteristic of the sequential consistency will not hold, if we use the acquire-release semantic for atomic operations. This is an area, in which C# or Java will not follow. But that's also an area, in which our intuition begins to wane.

There is no global synchronization between threads in the acquire-release semantic, there is only a synchronisation between atomic operations on the same atomic variable. So a write operation on one thread synchronises with a read operation on another thread on the same atomic variable. This synchronization relation on the same atomic variable helps to establish a happens-before relation between atomic variables and therefore between threads.

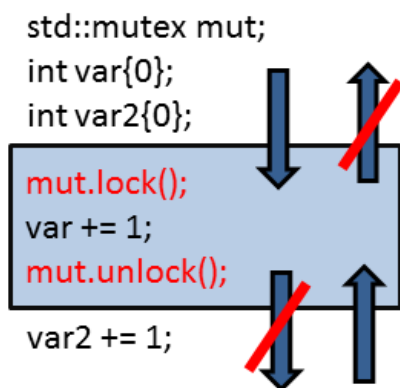
Acquire Release Semantic

With the acquire-release semantic the memory model gets very thrilling. Because now, we have not to reason about the synchronisation of threads, now we have to reason about the synchronisation of the same atomic in different threads.

Synchronisation and ordering constraints

The acquire-release semantic is based on one key idea. A release operation synchronises with an acquire operation on the same atomic and establishes, in addition, an ordering constraint. So, all read and write operations can not be moved before an acquire operation, all read and write operations can not be move behind a release operation. But what is an acquire or release operation? The reading of an atomic variable with `load` or `test_and_set` is an acquire operation. But in addition the acquiring of a lock. Of course, the opposite is also true. The releasing of a lock is a release operation. Accordingly, a `store` or `clear` operation on an atomic variable.

It's worth to reason once more on the few last sentences from a different perspective. The lock of a mutex is an acquire operation, the unlock of a mutex a release operation. Figuratively speaking that implies, that an operation on a variable can not moved outside of a [critical section](#). That hold for both directions. On the other side, a variable can be moved inside of a critical section. Because the variable moves from the not protected to the protected area. Now with a little delay the picture.



Did I not promised it? The acquire-release semantic helps a lot to better understand the lock and unlock operation of a mutex. The same reasoning will hold for the [starting of a thread](#) or the `join`-call on a thread. Both are release operations. But that story goes on with the `wait` and `notify_one`-call on a [condition variable](#). `wait` in this case is the acquire, `notify_one` the release operation. But what is about `notify_all`? Of course you can already guess it. That is a release operation.

Because of the theory, I can write the spinlock from the post [The atomic flag](#) more efficient, because the synchronization takes place on the atomic variable `flag`.

```

1 // spinlockAcquireRelease.cpp
2
3 #include <atomic>
4 #include <thread>
5
6 class Spinlock{
7     std::atomic_flag flag;
8 public:
9     Spinlock(): flag(ATOMIC_FLAG_INIT) {}
10
11     void lock(){
12         while(flag.test_and_set(std::memory_order_acquire) );
13     }
14
15     void unlock(){
16         flag.clear(std::memory_order_release);
17     }
18 };
19
20 Spinlock spin;
21
22 void workOnResource(){
23     spin.lock();
24     // shared resource
25     spin.unlock();
26 }
27
28
29 int main(){
30
31     std::thread t(workOnResource);
32     std::thread t2(workOnResource);
33
34     t.join();
35     t2.join();
36
37 }

```

The `flag.clear`-call in line 16 is a release, the `flag.test_and_set`-call in line 12 an acquire-operation. And - but that is boring - the acquire synchronizes with the release operation. So the heavyweight synchronization with sequential consistency (`std::memory_order_seq_cst`) of two threads is replaced with the lightweight and more performant acquire-release semantic (`std::memory_order_acquire` and `std::memory_order_release`). The behaviour is unchanged.

In case more than two threads uses the spinlock, the acquire semantic of the lock method is not sufficient. Now the lock method is an acquire-release operation. So the memory model in line 12 has to be changed to `std::memory_order_acq_rel`. But that is still cheaper than the default: `std::memory_order_seq_cst`.

Transitivity of the Acquire-Release Semantic

A release operation synchronises with an acquire operation on the same atomic variable and establishes, in addition, an ordering constraints. These are the components to synchronise threads in a performant way, in case they act on the same atomic. But how can that work, if two threads share no atomic variable? We want no sequential consistency because that is too heavy. We want the light acquire-release semantic.

The answer to the riddle is easy. Because of the transitivity of the acquire-release semantic, threads can be synchronised, which act independently of each other.

Transitivity

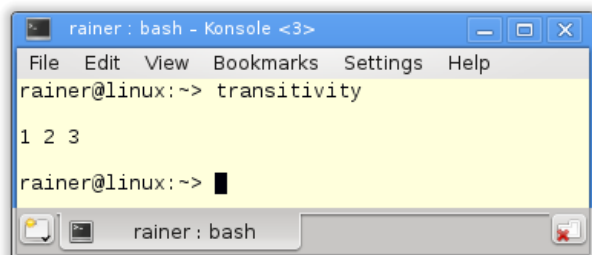
In the following example, the thread `t2` with its work package `deliveryBoy` is the glue between the two independent threads `t1` and `t3`.

```

1 // transitivity.cpp
2
3 #include <atomic>
4 #include <iostream>
5 #include <thread>
6 #include <vector>
7
8 std::vector<int> mySharedWork;
9 std::atomic<bool> dataProduced(false);
10 std::atomic<bool> dataConsumed(false);
11
12 void dataProducer() {
13     mySharedWork={1,0,3};
14     dataProduced.store(true, std::memory_order_release);
15 }
16
17 void deliveryBoy() {
18     while( !dataProduced.load(std::memory_order_acquire) );
19     dataConsumed.store(true, std::memory_order_release);
20 }
21
22 void dataConsumer() {
23     while( !dataConsumed.load(std::memory_order_acquire) );
24     mySharedWork[1]= 2;
25 }
26
27 int main() {
28
29     std::cout << std::endl;
30
31     std::thread t1(dataConsumer);
32     std::thread t2(deliveryBoy);
33     std::thread t3(dataProducer);
34
35     t1.join();
36     t2.join();
37     t3.join();
38
39     for (auto v: mySharedWork) {
40         std::cout << v << " ";
41     }
42
43     std::cout << "\n\n";
44
45 }

```

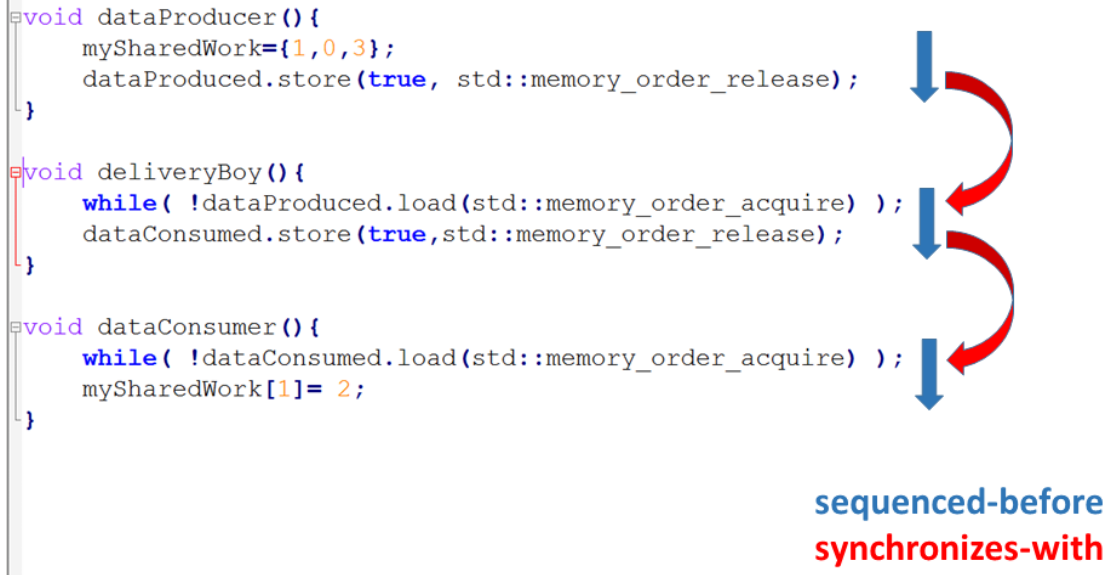
The output of the program is totally deterministic. `mySharedWork` will have the values 1,2, and 3.



Why is the program totally deterministic? There are two important observations:

1. Thread `t2` waits in line 18, until thread `t3` has set `dataProduced` on `true` (line 14).
2. Thread `t1` waits in line 23, until thread `t2` has set `dataConsumed` on `true` (line 19).

The rest is the easier explained with a picture.



The important parts of the picture are the arrows.

- The blue arrows are the *sequenced-before* relations. That means, that all operations in one thread will be executed in source code order.
- The red arrows are the *synchronizes-with* relations. The reason is the acquire-release semantic of the atomic operations on the same atomic. So the synchronisation between the threads takes place.
- As well *sequenced-before* as *synchronizes-with* establishes a *happens-before* relation.

The rest is pretty simple. The chronological order of the instructions (*happens-before*) corresponds to the direction of the arrows from top to bottom. So, we have the guarantee, that `mySharedWork[1] == 2` will be executed last.

Acquire-Release Semantic: The Typical Error

A release operation synchronizes-with an acquire operation on the same atomic variable. So we can easily synchronise threads **if** Today's post is about the **if**.

What's my motivation for writing a post about the typical misunderstanding of the acquire-release semantic? Sure, I and many of my listeners and trainees have already fallen into the trap. But at first the straightforward case.

Waiting included

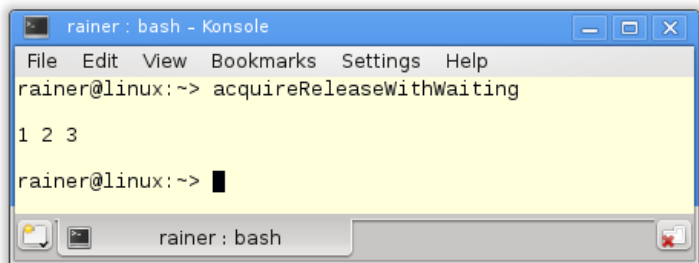
I use this simple program as a starting point.

```

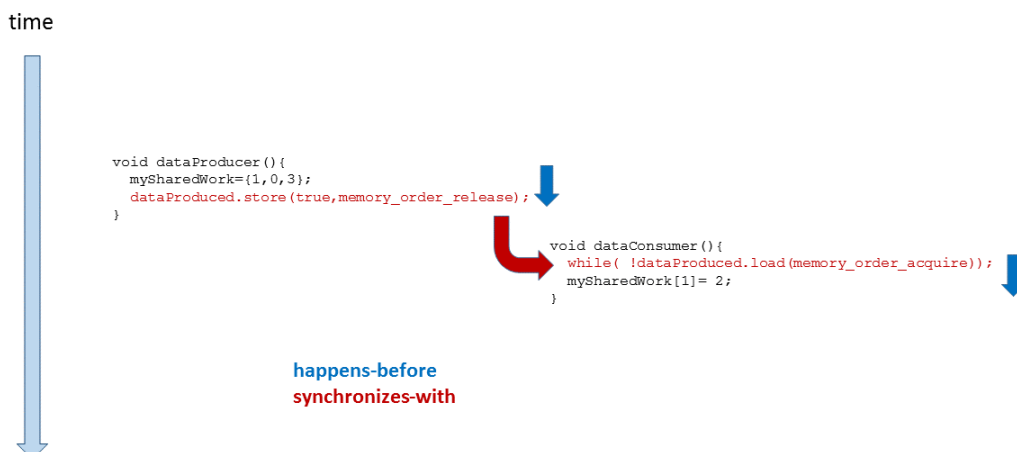
1 // acquireReleaseWithWaiting.cpp
2
3 #include <atomic>
4 #include <iostream>
5 #include <thread>
6 #include <vector>
7
8 std::vector<int> mySharedWork;
9 std::atomic<bool> dataProduced(false);
10
11 void dataProducer() {
12     mySharedWork={1,0,3};
13     dataProduced.store(true, std::memory_order_release);
14 }
15
16 void dataConsumer() {
17     while( !dataProduced.load(std::memory_order_acquire) );
18     mySharedWork[1]= 2;
19 }
20
21 int main() {
22
23     std::cout << std::endl;
24
25     std::thread t1(dataConsumer);
26     std::thread t2(dataProducer);
27
28     t1.join();
29     t2.join();
30
31     for (auto v: mySharedWork) {
32         std::cout << v << " ";
33     }
34
35     std::cout << "\n\n";
36
37 }

```

The consumer thread t1 in line 17 is waiting until the consumer thread t2 in line 13 has set `dataProduced` to true. `dataProduced` is the guard, because it guarantees, that the access to the non atomic variable `mySharedWork` is synchronized. That means, at first the producer thread t2 initializes `mySharedWork`, then the consumer thread t2 finishes the work by setting `mySharedWork[1]` to 2. So the program is well defined.



The graphic shows the *happens-before* relation within the threads and the *synchronizes-with* relation between the threads. *synchronizes-with* establishes a *happens-before* relation. The rest of the reasoning is the transitivity of the *happens-before* relation. `mySharedWork={1,0,3}` *happens-before* `mySharedWork[1]=2`.



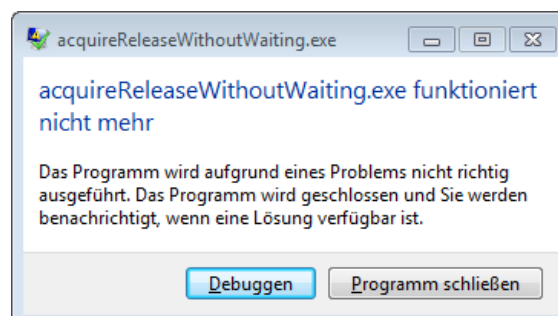
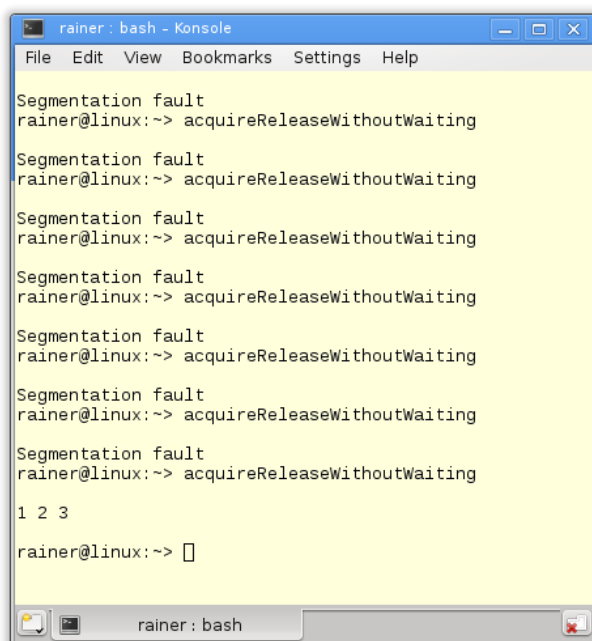
But what aspect is often missing in this reasoning. The **if**.

If, ...

What is happening, **if** the consumer thread t2 in line 17 is not waiting for the producer thread?

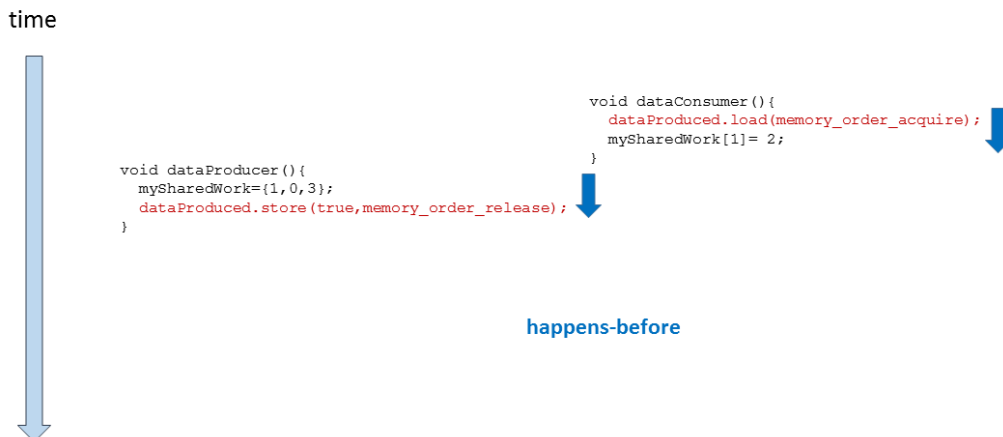
```
1 // acquireReleaseWithoutWaiting.cpp
2
3 #include <atomic>
4 #include <iostream>
5 #include <thread>
6 #include <vector>
7
8 std::vector<int> mySharedWork;
9 std::atomic<bool> dataProduced(false);
10
11 void dataProducer() {
12     mySharedWork={1,0,3};
13     dataProduced.store(true, std::memory_order_release);
14 }
15
16 void dataConsumer() {
17     dataProduced.load(std::memory_order_acquire);
18     mySharedWork[1]= 2;
19 }
20
21 int main() {
22
23     std::cout << std::endl;
24
25     std::thread t1(dataConsumer);
26     std::thread t2(dataProducer);
27
28     t1.join();
29     t2.join();
30
31     for (auto v: mySharedWork) {
32         std::cout << v << " ";
33     }
34
35     std::cout << "\n\n";
36
37 }
```

The program has undefined behaviour because there is a [data race](#) on the variable `mySharedWork`. In case I let the program run, the undefined behaviour gets immediately visible. That holds for Linux and Windows.



What's the issue? It holds: `store(true, std::memory_order_release)` *synchronizes-with*

`dataProduced.load(std::memory_order_acquire)`. Yes of course, but that doesn't mean the acquire operation is waiting for the release operation. Exactly that is displayed in the graphic. In the graphic the `dataProduced.load(std::memory_order_acquire)` instruction is performed before the instruction `dataProduced.store(true, std::memory_order_release)`. So we have no *synchronize-with* relation.



The solution

synchronize-with means in this specific case: **If** `dataProduced.store(true, std::memory_order_release)` happens before `dataProduced.load(std::memory_order_acquire)`, **then** all visible effect of operations before `dataProduced.store(true, std::memory_order_release)` are visible after `dataProduced.load(std::memory_order_acquire)`. The key is the word **if**. Exactly that **if** will be guaranteed in the first program with `(while(!dataProduced.load(std::memory_order_acquire))`.

Once again, but formal.

- All operations before `dataProduced.store(true, std::memory_order_release)` *happens-before* all operations after `dataProduced.load(std::memory_order_acquire)`, if holds: `dataProduced.store(true, std::memory_order_release)` *happens-before* `dataProduced.load(std::memory_order_acquire)`.

Memory Order Consume

`std::memory_order_consume` is the most legendary of the [six memory models](#). That's for two reasons. At one hand, `std::memory_order_consume` is extremely hard to get. At the other hand - that may change in the future - no compiler supports it.

How can it happen, that a compiler supports the C++11 standard, but doesn't support the memory model `std::memory_order_consume`? The answer is, that compiler maps `std::memory_order_consume` to `std::memory_order_acquire`. That is fine because both are load or acquire operations. `std::memory_order_consume` requires weaker [synchronisation and ordering constraints](#). So the release-acquire ordering is potentially slower than the release-consume ordering but - that is the key point - well defined.

To get an understanding of the release-consume ordering, it's a good idea to compare it with the release-acquire ordering. I speak in the post explicitly from the release-acquire ordering and not from the [acquire-release semantic](#) to emphasise the strong relationship of `std::memory_order_consume` and `std::memory_order_acquire`.

Release-acquire ordering

As starting point I use a program with two threads `t1` and `t2`. `t1` plays the role of the producer, `t2` the role of the consumer. The atomic variable `ptr` helps to synchronize the producer and consumer.


```

1 // acquireRelease.cpp
2
3 #include <atomic>
4 #include <thread>
5 #include <iostream>
6 #include <string>
7
8 std::atomic<std::string*> ptr;
9 int data;
10 std::atomic<int> atoData;
11
12 void producer() {
13     std::string* p = new std::string("C++11");
14     data = 2011;
15     atoData.store(2014, std::memory_order_relaxed);
16     ptr.store(p, std::memory_order_release);
17 }
18
19 void consumer() {
20     std::string* p2;
21     while (! (p2 = ptr.load(std::memory_order_acquire)));
22     std::cout << "*p2: " << *p2 << std::endl;
23     std::cout << "data: " << data << std::endl;
24     std::cout << "atoData: " << atoData.load(std::memory_order_relaxed) << std::endl;
25 }
26
27 int main() {
28
29     std::cout << std::endl;
30
31     std::thread t1(producer);
32     std::thread t2(consumer);
33
34     t1.join();
35     t2.join();
36
37     std::cout << std::endl;
38
39 }

```

Before I analyse the program, I want to introduce a small variation. I replace in line 21 the memory model `std::memory_order_acquire` by `std::memory_order_consume`.

Release-consume ordering

```

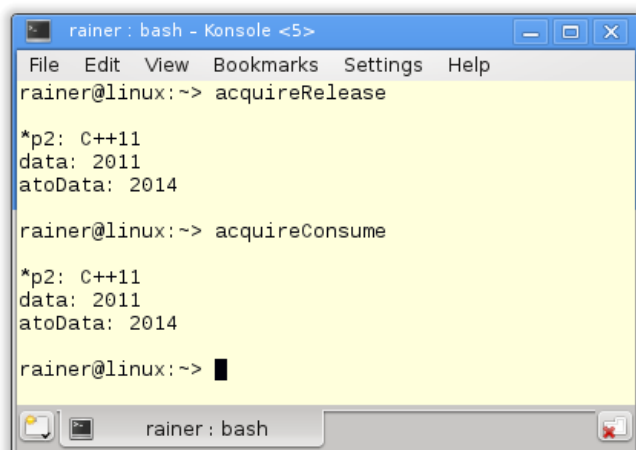
1 // acquireConsume.cpp
2
3 #include <atomic>
4 #include <thread>
5 #include <iostream>
6 #include <string>
7
8 std::atomic<std::string*> ptr;
9 int data;
10 std::atomic<int> atoData;
11
12 void producer() {
13     std::string* p = new std::string("C++11");
14     data = 2011;
15     atoData.store(2014, std::memory_order_relaxed);
16     ptr.store(p, std::memory_order_release);
17 }
18
19 void consumer() {
20     std::string* p2;
21     while (!(p2 = ptr.load(std::memory_order_consume)));
22     std::cout << "*p2: " << *p2 << std::endl;
23     std::cout << "data: " << data << std::endl;
24     std::cout << "atoData: " << atoData.load(std::memory_order_relaxed) << std::endl;
25 }
26
27 int main() {
28     std::cout << std::endl;
29
30     std::thread t1(producer);
31     std::thread t2(consumer);
32
33     t1.join();
34     t2.join();
35
36     std::cout << std::endl;
37 }
38
39 }

```

That was easy. But now the program has undefined behaviour. That statement is very hypothetical, because my compiler implements `std::memory_order_consume` by `std::memory_order_acquire`. So under the hood both program actually do the same.

Release-acquire versus Release-consume ordering

The output of the programs is identical.



```

rainer : bash - Konsole <5>
File Edit View Bookmarks Settings Help
rainer@linux:~> acquireRelease

*p2: C++11
data: 2011
atoData: 2014

rainer@linux:~> acquireConsume

*p2: C++11
data: 2011
atoData: 2014

rainer@linux:~> █

```

Although I repeat myself, I want to sketch in a few words, why the first program `acquireRelease.cpp` is well defined.

The store operation in line 16 synchronizes-with the load operation in line 21. The reason is, that the store operation uses `std::memory_order_release`, that the load operation uses `std::memory_order_acquire`. That was the synchronization. What's about the ordering constraints of the release-acquire ordering? The release-acquire ordering guarantees, that all operations before the store operation (line 16) are available after the load operation (line 21). So the release-acquire operation orders in addition the access on the non-atomic variable `data` (line 14) and the atomic variable `atoData` (line 15). That holds, although `atoData` uses the `std::memory_order_relaxed` memory model.

The key question is. What happens, if I replace in the program `std::memory_order_acquire` by `std::memory_order_consume`?

Data dependencies with `std::memory_order_consume`

The `std::memory_order_consume` is about data dependencies on atomics. Data dependencies exist in two ways. At first *carries-a-dependency-to* in a thread and *dependency-ordered-before* between two threads. Both dependencies introduce a *happens-before* relation. That is this kind of relation a well defined program needs. But what means *carries-a-dependency-to* and *dependency-ordered-before*?

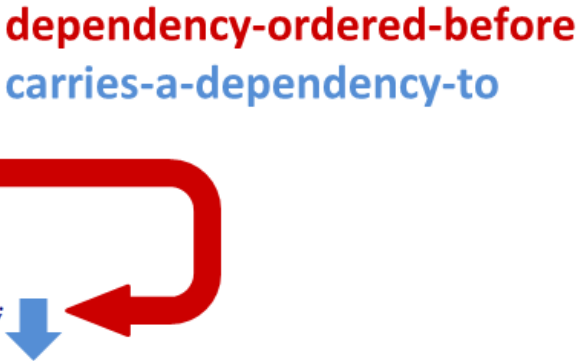
- ***carries-a-dependency-to***: If the result of an operation A is used as an operand of an operation B, then: A *carries-a-dependency-to* B.
- ***dependency-ordered-before***: A store operation (with `std::memory_order_release`, `std::memory_order_acq_rel` or `std::memory_order_seq_cst`), is *dependency-ordered-before* a load operation B (with `std::memory_order_consume`), if the result of the load operation B is used in a further operation C in the same thread. The operations B and C have to be in the same thread.

Of course I know from personal experience, that both definitions are not easy to digest. So I will use a graphic to visually explain them.

```
std::atomic<std::string*> ptr;
int data;
std::atomic<int> atoData;

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

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



The expression `ptr.store(p, std::memory_order_release)` is *dependency-ordered-before* while `!(p2 = ptr.load(std::memory_order_consume))`, because in the following line `std::cout << "*p2: " << *p2 << std::endl` the result of the load operation will be read. Further, holds `while (!(p2 = ptr.load(std::memory_order_consume))` *carries-a-dependency-to* `std::cout << "*p2: " << *p2 << std::endl`, because the output of `*p2` uses the result of the `ptr.load` operation.

But we have no guarantee for the following outputs of `data` and `atoData`. That's because both have no *carries-a-dependency* relation to the `ptr.load` operation. But it gets even worse. Because `data` is a nonatomic variable, there is a [race condition](#) on `data`. The reason is, that both threads can access `data` at the same time and thread t1 wants to modify `data`. Therefore, the program is undefined.

Relaxed Semantic

The relaxed semantic is the end of the Scala. The relaxed semantic is the weakest of all memory models and guarantees only, that the operations on atomic variables are atomic.

No synchronisation and ordering constraints

That's quite easy. If there are no rules, we can not break them. But that's too easy. The program should have well-defined behaviour. That means in this case: No [race condition](#). To guarantee this, you typically use synchronisation and ordering constraints of stronger memory models to control operations with relaxed semantic. How does this work? A thread can see the effects of another thread in arbitrary order. So, you must only be sure, that there are points in your program, in which all operations on all threads gets synchronised.

A typical example for an atomic operation, in which the sequence of operations doesn't matter, is a counter. The key of a counter

is not, in which order the different threads increment the counter. The key of the counter is, that all increments are atomic and that all threads are done at the end. Have a look at the example.

```
1 // relaxed.cpp
2
3 #include <vector>
4 #include <iostream>
5 #include <thread>
6 #include <atomic>
7
8 std::atomic<int> cnt = {0};
9
10 void f()
11 {
12     for (int n = 0; n < 1000; ++n) {
13         cnt.fetch_add(1, std::memory_order_relaxed);
14     }
15 }
16
17 int main()
18 {
19     std::vector<std::thread> v;
20     for (int n = 0; n < 10; ++n) {
21         v.emplace_back(f);
22     }
23     for (auto& t : v) {
24         t.join();
25     }
26     std::cout << "Final counter value is " << cnt << '\n';
27 }
```

The three most interesting lines are 13, 24, and 26.

In line 13 the atomic number `cnt` is incremented with relaxed semantic. So, we have the guarantee, that the operation is atomic. The `fetch_add` operation established an ordering on `cnt`. The function `f` (line 10 - 15) is the work package of the threads. Each thread gets its work package in line 21.

Thread creation is one synchronisation point. The other synchronisation point is the `t.join()` call in line 24.

The creator thread synchronises with all its child's in line 24. It waits with the `t.join()` call until all its children are done. `t.join()` is the reason, that the results of the atomic operations are published. To say it more formally `t.join()` is a release operation.

At the end, there is a *happen-before* relation between the increment operation in line 13 and the reading of the counter `cnt` in line 26.

The result is, that the program returns always 10000. Boring? No, calming!

A typical example for an atomic counter, which uses the relaxed semantic, is the reference counter of `std::shared_ptr`. That will only hold for the increment operation. Key for incrementing the reference counter is, that the operation is atomic. The order of the increment operations does not matter. That will not hold for the decrementation of the reference counter. These operations need an acquire-release semantic with the destructor.

I want to explicitly say thanks to Anthony Williams, author of the well-known book C++ Concurrency in Action. He gave me very valuable tips for this post. Anthony writes his own blog to concurrency in modern C++:

<https://www.justsoftwaresolutions.co.uk/blog/>.

Business before pleasure

Business before pleasure. That's my simple motto for the next posts. So, I will use the theory about [atomics](#) and the [memory model](#) in practice.

```
int x= 0;
int y= 0;

void writing() {
    x= 2000;
    y= 11;
}

void reading() {
    std::cout << "y: " << y << " ";
    std::cout << "x: " << x << std::endl;
}

int main() {
    std::thread thread1(writing);
    std::thread thread2(reading);
    thread1.join();
    thread2.join();
};
```

Fences are Memory Barriers

The key idea of a `std::atomic_thread_fence` is, to establish synchronisation and ordering constraints between threads without an atomic operation.

`std::atomic_thread_fence` are simply called fences or memory barriers. So you get immediately the idea, what a `std::atomic_thread_fence` is all about.

A `std::atomic_thread_fence` prevents, that specific operations can overcome a memory barrier.

Memory barriers

But what does that mean? Specific operations, which can not overcome a memory barrier. What kind of operations? From a bird's perspective, we have two kinds of operations: Read and write or load and store. So the expression `if(resultRead) return result` is a load, followed by a store operation.

There are four different ways to combine load and store operations:

- **LoadLoad:** A load followed by a load.
- **LoadStore:** A load followed by a store.
- **StoreLoad:** A store followed by a load.
- **StoreStore:** A store followed by a store.

Of course, there are more complex operations, consisting of a load and store part(`count++`) . But these operations didn't contradict my general classification.

But what's about memory barriers?. In case you place memory barriers between two operations like LoadLoad, LoadStore, StoreLoad or StoreStore, you have the guarantee, that specific LoadLoad, LoadStore, StoreLoad or StoreStore operations can not be reordered. The risk of reordering is always given if non-atomics or atomics with relaxed semantic are used.

Typically, three kinds of memory barriers are used. They are called **full fence**, **acquire fence** and **release fence**. Only in order to remind you. Acquire is a load, release is a store operation. So, what's happening if I place one of the three memory barriers between the four combinations of load and store operations?

- **Full fence:** A full fence `std::atomic_thread_fence()` between two arbitrary operations prevents the reordering of these operations. But that guarantee will not hold for StoreLoad operations. They can be reordered.
- **Acquire fence:** An acquire fence `std::atomic_thread_fence(std::memory_order_acquire)` prevents, that a read operation before an acquire fence can be reordered with a read or write operation after the acquire fence.
- **Release fence:** A *release fence* `std::atomic_thread_fence(std::memory_order_release)` prevents, that a read or write operation before a release fence can be reordered with a write operation after a release fence.

I admit, that I invested a lot of energy to get the definitions of an acquire and release fence and there consequences for lock-free programming. Especially the subtle difference to the acquire-release semantic of atomic operations are not so easy to get. But, before I come to that point, I will illustrate the definitions with graphics.

Memory barriers illustrated

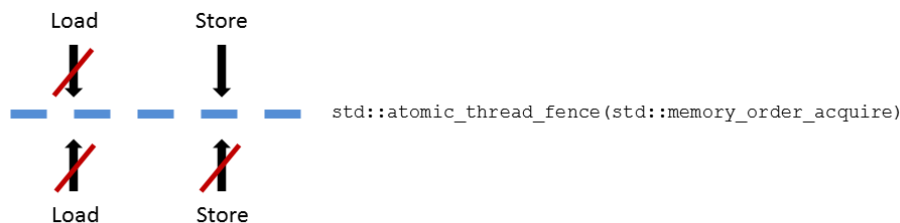
Which kind of operations can overcome a memory barrier? Have a look at the three following graphics. If the arrow is crossed with a red bar, the fence prevents this kind of operation.

Full fence

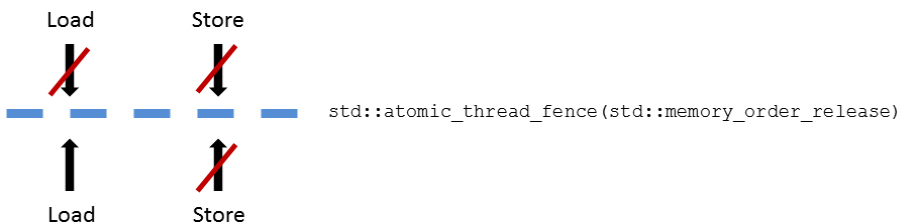


Of course, you can explicitly write instead of `std::atomic_thread_fence()` `std::atomic_thread_fence(std::memory_order_seq_cst)`. Per default, [sequential consistency](#) is used for fences. Is sequential consistency used for a full fence, the `std::atomic_thread_fence` follows a global order.

Acquire fence

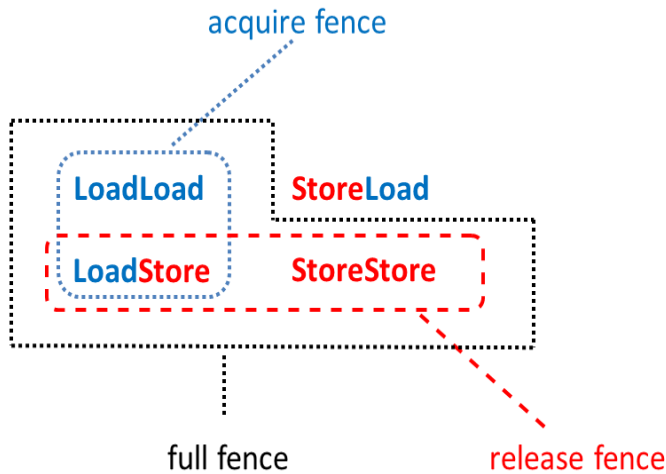


Release fence



But I can depict the three memory barriers even more concise.

Memory barriers at a glance



Acquire-Release Fence

Acquire and release fences guarantees similar [synchronisation and ordering constraints](#) as atomics with [acquire-release semantic](#). Similar, because the differences are in the details.

The most obvious difference between acquire and release memory barriers (fences) and atomics with acquire-release semantic is that memory barriers need no operations on atomics. But there is a more subtle difference. The acquire and release memory barriers are more heavyweight.

Atomic operations versus memory barriers

To make my job of writing simpler, I will now simply speak of acquire operations, if I use memory barriers or atomic operations with acquire semantic. The same will hold for release operations.

The key idea of an acquire and a release operation is, that it establishes synchronisations and ordering constraints between thread. This will also hold for atomic operations with relaxed semantic or non-atomic operations. So you see, the acquire and release operations come in pairs. In addition, for the operations on atomic variables with acquire-release semantic must hold that these act on the same atomic variable. Said that I will in the first step look at these operations in isolation.

I start with the acquire operation.

Acquire operation

A read operation on an atomic variable attached with `std::memory_order_acquire` is an acquire operation.

```
int ready= var.load(std::memory_order_acquire);
// load and store operations
```



In opposite to that there is the `std::atomic_thread_fence` with acquire semantic.

```
// load operations
int ready= var.load(std::memory_order_relaxed);
std::atomic_thread_fence(std::memory_order_acquire);
// load and store operations
```



This comparison emphasise two points.

1. A memory barrier with acquire semantic establishes stronger ordering constraints. Although the acquire operation on an

atomic and on a memory barrier requires, that no read or write operation can be moved before the acquire operation, there is an additional guarantee with the acquire memory barrier. No read operation can be moved after the acquire memory barrier.

2. The relaxed semantic is sufficient for the reading of the atomic variable `var`. The

`std::atomic_thread_fence(std::memory_order_acquire)` ensures that this operation can not be moved after the acquire fence.

The similar statement holds for the release memory barrier.

Release operation

The write operation on an atomic variable attached with the memory model `std::memory_order_release` is a release operation.

```
// load and store operations  
var.store(1, std::memory_order_release);
```



And further the release memory barrier.

```
// load and store operations  
std::atomic_thread_fence(std::memory_order_release);  
var.store(1, std::memory_order_relaxed);  
// store operations
```



In addition to the release operation on an atomic variable `var`, the release barrier guarantees two points:

1. Store operations can't be moved before the memory barrier.
2. It's sufficient for the variable `var` to have relaxed semantic.

In case you want a simple overview of memory barriers, please read the last post in this blog. But now, I want to go one step further and build a program out of the presented components.

Synchronisation with atomic operations versus memory barriers

I implement as starting point for my comparison a typical consumer-producer workflow with acquire-release semantic. I will do this job with atomics and memory barriers.

Let's start with atomics because the most of us are comfortable with them. That will not hold for memory barriers. They are almost complete ignored in the literature to the C++ memory model.

Atomic operations


```

1 // acquireRelease.cpp
2
3 #include <atomic>
4 #include <thread>
5 #include <iostream>
6 #include <string>
7
8 std::atomic<std::string*> ptr;
9 int data;
10 std::atomic<int> atoData;
11
12 void producer(){
13     std::string* p = new std::string("C++11");
14     data = 2011;
15     atoData.store(2014, std::memory_order_relaxed);
16     ptr.store(p, std::memory_order_release);
17 }
18
19 void consumer(){
20     std::string* p2;
21     while (!(p2 = ptr.load(std::memory_order_acquire)));
22     std::cout << "*p2: " << *p2 << std::endl;
23     std::cout << "data: " << data << std::endl;
24     std::cout << "atoData: " << atoData.load(std::memory_order_relaxed) << std::endl;
25 }
26
27 int main(){
28     std::cout << std::endl;
29
30     std::thread t1(producer);
31     std::thread t2(consumer);
32
33     t1.join();
34     t2.join();
35
36     delete ptr;
37
38     std::cout << std::endl;
39
40 }
41 }

```

I hope, this program looks familiar to you. That my classic that I used in the post to [memory_order_consume](#). The graphic goes directly to the point, why the consumer thread t2 sees all values from the producer thread t1.

```

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

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

```

**happens-before
synchronizes-with**

The program is well defined, because the *happens-before* relation is transitive. I have only to combine the three *happens-before* relations:

1. Line 13 - 15 *happens-before* line 16 (`ptr.store(p, std::memory_order_release)`).
2. Line 21 `while (!(p2 = ptr.load(std::memory_order_acquire)))` *happens-before* the lines 22 - 24.
3. Line 16 *synchronizes-with* line 21. => Line 16 *happens-before* line 21.

But now the story gets more thrilling. How can I adjust the workflow to memory barriers?

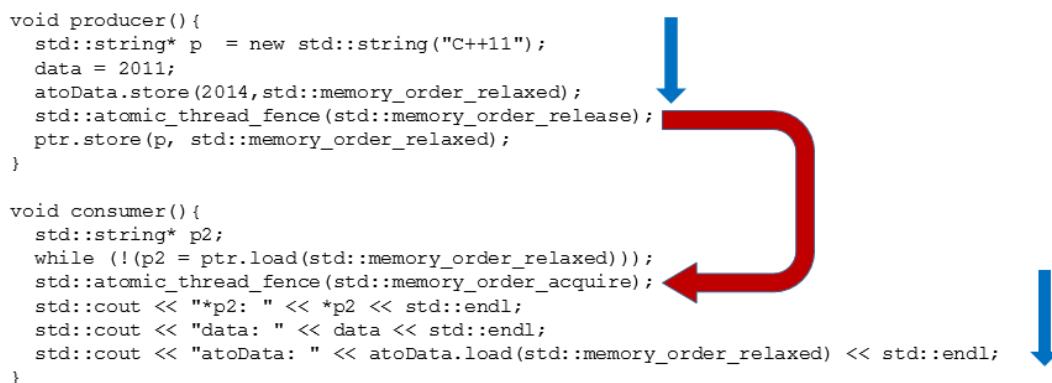
Memory barriers

It's straightforward to port the program to memory barriers.

```
1 // acquireReleaseFences.cpp
2
3 #include <atomic>
4 #include <thread>
5 #include <iostream>
6 #include <string>
7
8 std::atomic<std::string*> ptr;
9 int data;
10 std::atomic<int> atoData;
11
12 void producer() {
13     std::string* p = new std::string("C++11");
14     data = 2011;
15     atoData.store(2014, std::memory_order_relaxed);
16     std::atomic_thread_fence(std::memory_order_release);
17     ptr.store(p, std::memory_order_relaxed);
18 }
19
20 void consumer() {
21     std::string* p2;
22     while (!(p2 = ptr.load(std::memory_order_relaxed)));
23     std::atomic_thread_fence(std::memory_order_acquire);
24     std::cout << "*p2: " << *p2 << std::endl;
25     std::cout << "data: " << data << std::endl;
26     std::cout << "atoData: " << atoData.load(std::memory_order_relaxed) << std::endl;
27 }
28
29 int main() {
30
31     std::cout << std::endl;
32
33     std::thread t1(producer);
34     std::thread t2(consumer);
35
36     t1.join();
37     t2.join();
38
39     delete ptr;
40
41     std::cout << std::endl;
42
43 }
```

The first step is to insert just in place of the operations with acquire and release semantic the corresponding memory barriers with acquire and release semantic (line 16 and 23). In the next step, I change the atomic operations with acquire or release semantic to relaxed semantic (line 17 and 22). That was already mechanically. Of course, I can only replace one acquire or release operation with the corresponding memory barrier. The key point is, that the release operation establishes with the acquire operation a *synchronizes-with* relation and therefore a *happens-before* relation.

For the more visual reader, the whole description in a picture.



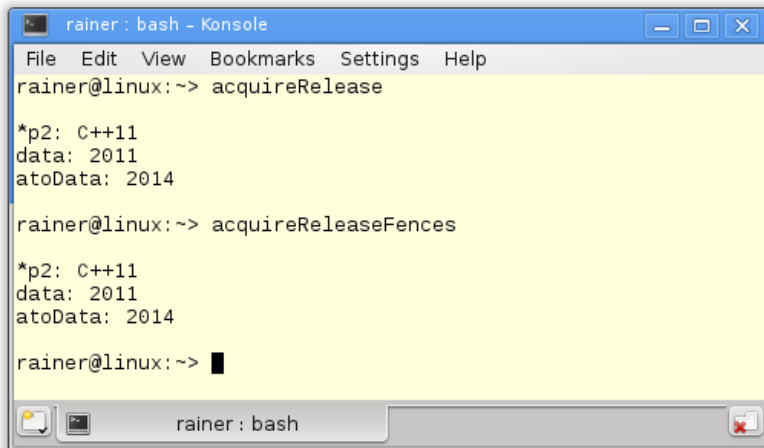
happens-before
synchronizes-with

The key question is. Why do the operations after the acquire memory barrier sees the effects of the operations before the release memory barrier? Because `data` is a non-atomic variable and `atoData` is used with relaxed semantic, both can be reordered. But that's not possible. The `std::atomic_thread_fence(std::memory_order_release)` as a release operation in combination with the `std::atomic_thread_fence(std::memory_order_acquire)` forbid the partial reordering. To follow my reasoning in detail, read the analysis of the memory barriers at the beginning of the post.

For clarity, the whole reasoning to the point.

1. The acquire and release memory barriers prevents the reordering of the atomic and non-atomic operations across the memory barriers.
2. The consumer thread `t2` is waiting in the `while (! (p2= ptr.load(std::memory_order_relaxed)))` loop, until the pointer `ptr.stor(p, std::memory_order_relaxed)` is set in the producer thread `t1`.
3. The release memory barrier *synchronizes-with* the acquire memory barrier.

Finally, the output of the programs.



```
rainer : bash - Konsole
File Edit View Bookmarks Settings Help
rainer@linux:~> acquireRelease

*p2: C++11
data: 2011
atoData: 2014

rainer@linux:~> acquireReleaseFences

*p2: C++11
data: 2011
atoData: 2014

rainer@linux:~> █
```