# Is the Acquire-Release Semantic Transitive?

This lesson introduces the concept of the acquire-release semantic being transitive.

WE'LL COVER THE FOLLOWING



The acquire-release semantic is transitive. That means if you have an acquire-release semantic between threads (a,b) and an acquire-release semantic between threads (b,c), you get an acquire-release semantic between (a,c).

## Transitivity

A release operation synchronizes with an acquire operation on the same atomic variable and additionally establishes ordering constraint. These are the components to synchronize threads in a performant way if they act on the same atomic. How can that work if two threads share no atomic variable? We do not want any sequential consistency because that is too expensive, but we want the light-weight acquire-release semantic.

The answer to this question is straightforward. Applying the transitivity of the acquire-release semantic, we are able to synchronize threads that are independent.

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

```
// transitivity.cpp

#include <atomic>
#include <iostream>
#include <thread>
#include <vector>
```

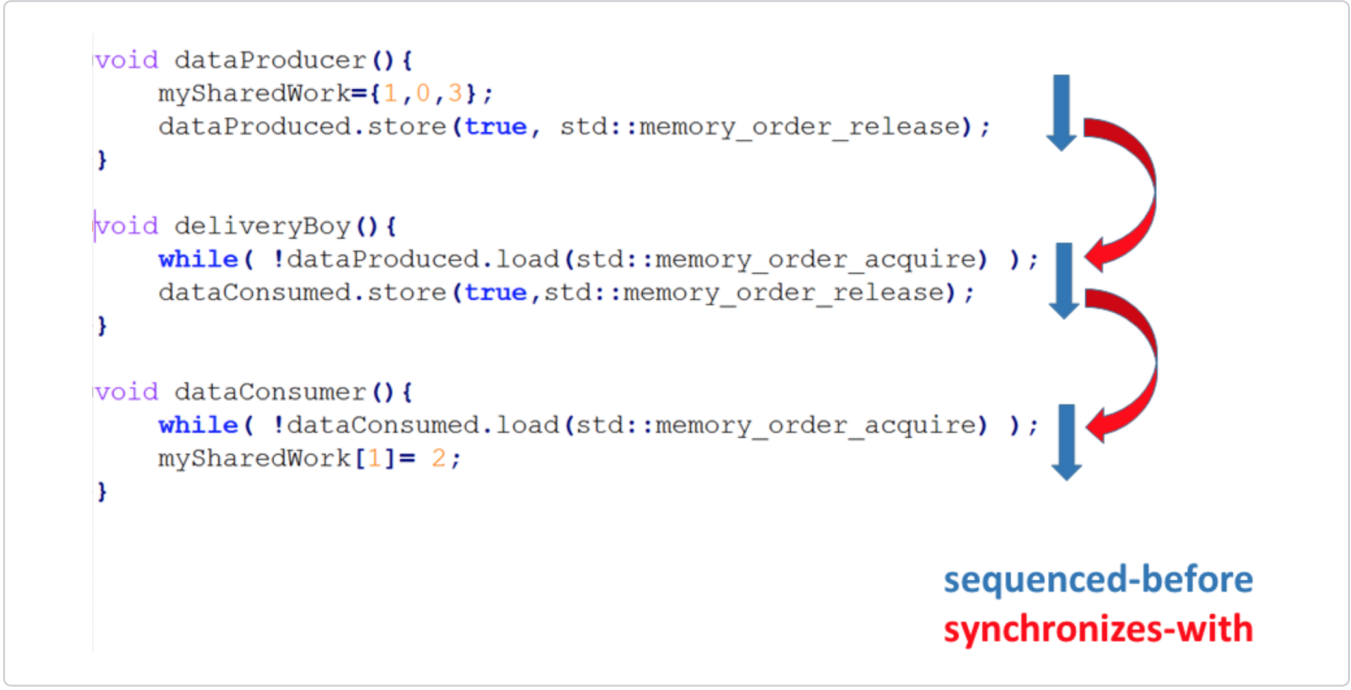
```
std::vector<int> mySharedWork;
std::atomic<bool> dataProduced(false);
std::atomic<bool> dataConsumed(false);


void dataProducer(){
    mySharedWork = {1,0,3};
    dataProduced.store(true, std::memory_order_release);
}

void deliveryBoy(){
    while(!dataProduced.load(std::memory_order_acquire));
    dataConsumed.store(true, std::memory_order_release);
}

void dataConsumer(){
    while(!dataConsumed.load(std::memory_order_acquire));
    mySharedWork[1] = 2;
}

int main(){

  std::cout << std::endl;

  std::thread t1(dataConsumer);
  std::thread t2(deliveryBoy);
  std::thread t3(dataProducer);

  t1.join();
  t2.join();
  t3.join();

  for (auto v: mySharedWork){
      std::cout << v << " ";
  }

  std::cout << "\n\n";

}
```

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

There are two important observations:

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

The rest is better explained with a graphic.




The important parts of the picture are the arrows.

- The **blue** arrows are the `sequenced-before` relations. This means that all operations in one thread will be executed in source code order.
- The **red** arrows are the *synchronizes-with* relations; the reason for this is the acquire-release semantic of the atomic operations on the same atomic. Subsequently, the synchronization between the atomics, and therefore between the threads at specific points, takes place.
- Both `sequenced-before` and `synchronizes-with` establishes a *happens-before* relation.

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

A release operation `synchronizes-with` an acquire operation on the same atomic variable, so we can easily synchronize threads, **if**... The typical misunderstanding is about the **if**.