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**.

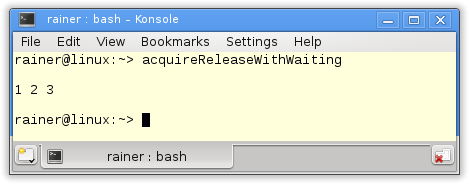
What's my motivation for writing a post about the typical misunderstanding of the acquire-release semantic? Sure, I and many of my listeners and trainees have already fallen into the trap. But at first the straightforward case.

Waiting included

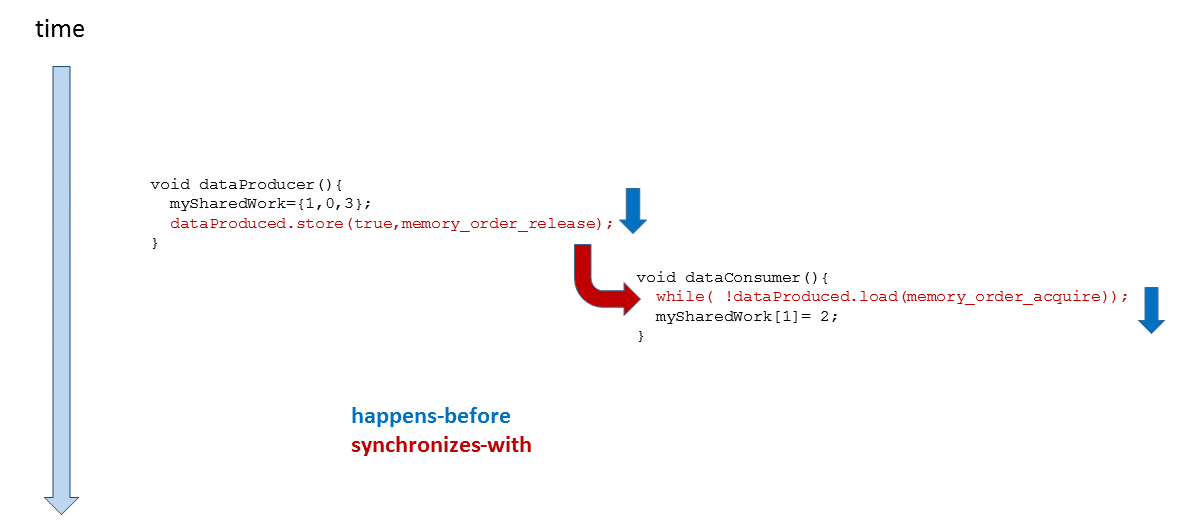
I use this simple program as a starting point.

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

The consumer thread t1 in line 17 is waiting until the consumer thread t2 in line 13 has set dataProduced to true.dataPruduced 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, than 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 *synchroniz-with* relation between the threads. *synchronize-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.



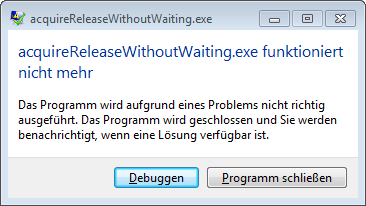
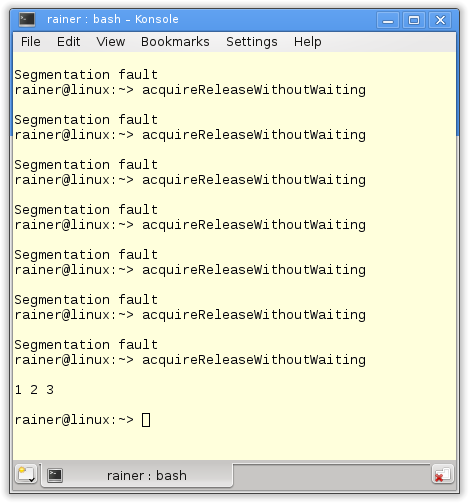
But what aspect is often missing in this reasoning. The **if.**

If, ...

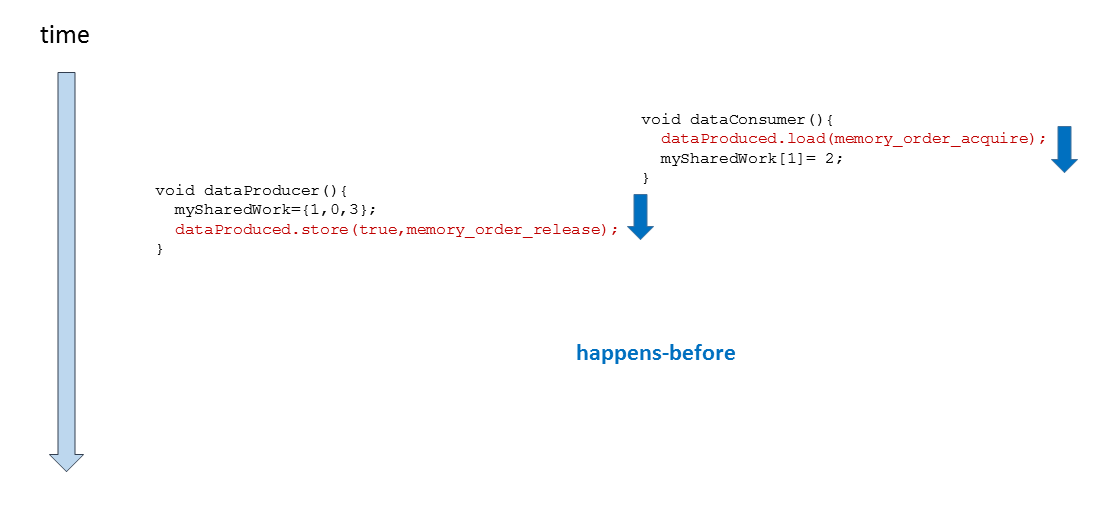
What is happing, **if**the consumer thread t2 in line 17 is not waiting for the producer thread?

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

The program has undefined behaviour because there is a [data race](http://modernescpp.com/index.php/threads-sharing-data)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) *synchron*izes*-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).