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.

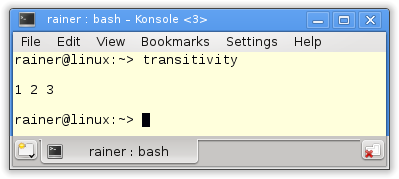
The answer to the riddle is easy. Because of the transitivity of the acquire-release semantic, threads can be synchronised, which act independently of each other.

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  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  38  39  40  41  42  43  44  45 | // 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";    } |

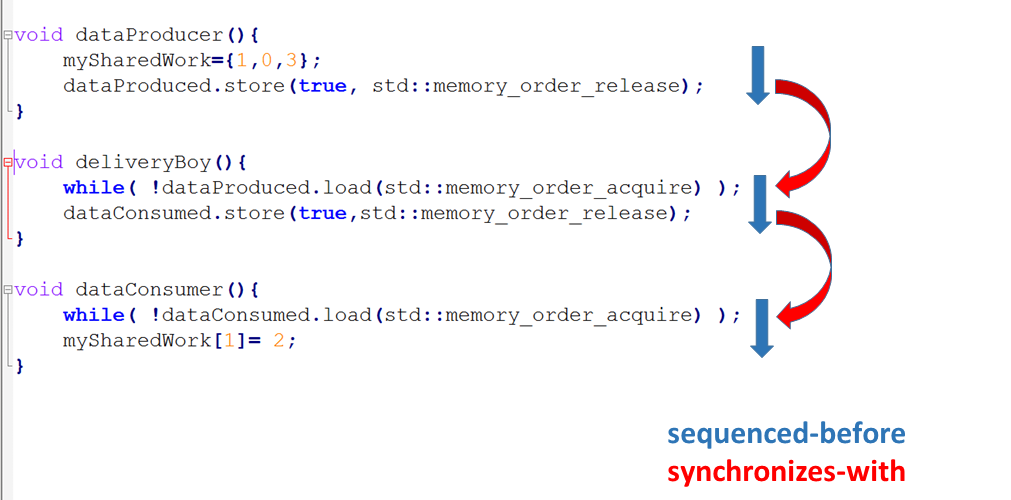
 The output of the program is totally deterministic. mySharedWork will have the values 1,2, and 3.



 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 import parts of the picture are the arrows.

* The blow 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 *synchronize-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 the direction of the arrows from top to bottom. So, we have the guarantee, that mySharedWork[1] == 2 will be executed last.