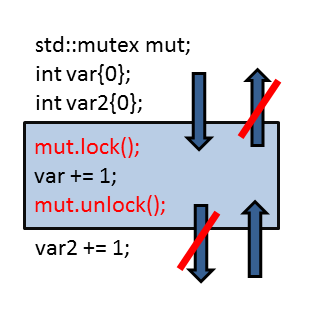
With the acquire-release semantic the memory model gets very thrilling. Because now, we have not to reason about the synchronisation of threads, now we have to reason about the synchronisation of the same atomic in different threads.

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](http://modernescpp.com/index.php/threads-sharing-data). That hold for both directions. On the other side, a variable can be moved inside of a criticial section. Because the variable moves from the not protected to the protected area. Now with a little delay the picture.



Did II 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](http://modernescpp.com/index.php/thread-creation) 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](http://modernescpp.com/index.php/condition-variables). wait is 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](http://modernescpp.com/index.php/the-atomic-flag) more efficient, because the synchronization takes place on the atomic variable flag.

|  |  |
| --- | --- |
| 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 | // spinlockAcquireRelease.cpp  #include <atomic>  #include <thread>  class Spinlock{  std::atomic\_flag flag;  public:  Spinlock(): flag(ATOMIC\_FLAG\_INIT) {}  void lock(){  while(flag.test\_and\_set(std::memory\_order\_acquire) );  }  void unlock(){  flag.clear(std::memory\_order\_release);  }  };  Spinlock spin;  void workOnResource(){  spin.lock();  // shared resource  spin.unlock();  }  int main(){  std::thread t(workOnResource);  std::thread t2(workOnResource);  t.join();  t2.join();  } |

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 sequencial 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.