Acquire Release Semantic

This lesson introduces the concept of the acquire-release semantic used in C++.

WE'LL COVER THE FOLLOWING

- Aquire-Release Operations

There is no global synchronization between threads in the acquire-release semantic; there is only a synchronization between atomic operations **on the same atomic variable**. A write operation on one thread synchronizes with a read operation on another thread on the same atomic variable.

The acquire-release semantic is based on one key idea: a release operation synchronizes with an acquire operation on the same atomic and establishes an ordering constraint. This means all subsequent **read** and **write** operations cannot be moved before an acquire operation, and all read and write operations cannot be moved after a release operation.

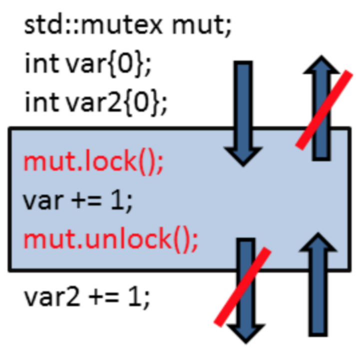
Aquire-Release Operations

What is an acquire or release operation? The reading of an atomic variable with `load` or `test_and_set` is an acquire operation. That being said, there is more: the acquiring of a lock, the creation of a thread, or waiting on a condition variable. Of course, the opposite is also true: releasing a lock, the join call on a thread or the notification of a condition variable are release operations. Accordingly, a `store` or `clear` operation on an atomic variable is a release operation. Acquire and release operations usually come in pairs.

It is worthwhile to think about the last few sentences from a different perspective. The lock of a mutex is an acquire operation, and the unlock of a mutex is a release operation. Figuratively speaking, this implies that an operation [illegible] cannot be moved outside of a critical section. On the other operation `var += 1` cannot be moved outside of a critical section. On the other

hand, a variable can be moved inside of a critical section because the variable moves from the non-protected to the protected area.

It helps a lot to keep that picture in mind.



This is the main reason you should keep the memory model in mind. In particular, the acquire-release semantic helps you to get a better understanding of the high-level synchronization primitives such as a mutex. The same reasoning holds for the starting of a thread and the join-call on a thread: both are acquire-release operations. The story goes on with the `wait` and `notify_one` call on a condition variable; `wait` is the acquire and `notify_one` the release operation. What about `notify_all`? That is a release operation as well.

Now, let us look once more at the spinlock in the subsection `std::atomic_flag**`. We can write it more efficiently because the synchronization is done with the atomic_flag `flag`, therefore the acquire-release semantic applies.

```
// spinlockAcquireRelease.cpp
#include<iostream>
#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(){
```

```
  void unlock(){
    flag.clear(std::memory_order_release);
  }
};

Spinlock spin;

void workOnResource(){
  spin.lock();
  // shared resource
  spin.unlock();
  std::cout << "Work done" << std::endl;
}


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 is an acquire operation, and the acquire synchronizes with the release operation. The heavyweight synchronization of two threads with sequential consistency (`std::memory_order_seq_cst`) is replaced by the lightweight and more performant acquire-release semantic (`std::memory_order_acquire` and `std::memory_order_release`). The behavior is not affected.

Although the `flag.test_and_set(std::memory_order_acquire)` call is a ***read-modify-write*** operation, the acquire semantic is sufficient. In summary, `flag` is an atomic.