The Atomic Flag

This lesson gives an overview of the atomic flag, which is used from the perspective of concurrency in C++.

The atomic flag, i.e. `std::atomic_flag`, has a very simple interface. Its `clear` method enables you to set its value to `false`; with the `test_and_set` method you can set the value back to `true`. There is no method to exclusively ask for the current value. To use `std::atomic_flag` it must be initialized to `false` with the constant `ATOMIC_FLAG_INIT`. `std::atomic_flag` has two outstanding properties.

`std::atomic_flag` is:

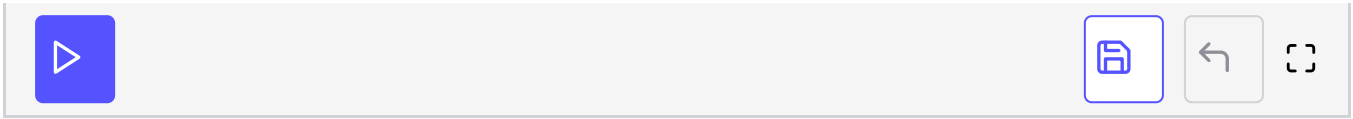
- the only lock-free atomic. A non-blocking algorithm is lock-free if there is guaranteed system-wide progress.
- the building block for higher level thread abstractions.

The only lock-free atomic? The remaining more powerful atomics can provide their functionality by using a mutex internally according to the C++ standard. These remaining atomics have a method called `is_lock_free` to check if the atomic uses a mutex internally. On the popular microprocessor architectures, I always get the answer `true`. That being said, my implementation internally uses no mutex; you should be aware of this and check it on your target system if you want to program lock-free.

The interface of `std::atomic_flag` is powerful enough to build a spinlock. With a spinlock, you can protect a critical section as you would with a mutex. The spinlock will not passively wait, in contrast to a mutex, until it gets it to lock. It will eagerly ask for the lock to get access to the critical section. It saves the expensive context switch in the wait state from the user space to the kernel space, but it fully utilizes the CPU and doesn't waste CPU cycles.

The example shows the implementation of a spinlock with the help of `std::atomic_flag`.

```
// spinLock.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() );
  }

  void unlock(){
    flag.clear();
  }
};

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();

}
```



Both threads `t` and `t2` (lines 31 and 32) are competing for the critical section. For simplicity, the critical section in line 24 consists only of a comment. How does it work? The class `Spinlock` has the methods `lock` and `unlock` - similar to a mutex. In addition to this, the constructor of `Spinlock` initializes the `std::atomic_flag` to `false` (line 9).

If thread `t` is going to execute the function `workOnResource`, the following scenarios can happen:

1. Thread `t` gets the lock because the `lock` invocation was successful. The `lock` invocation is successful if the initial value of the flag in line 12 is `false`. In this case, thread `t` sets it in an atomic operation to `true`. The

`false`. In this case, thread `t` sets it in an atomic operation to `true`. The value `true` is the value the while loop returns to thread `t2` if it tries to get the lock. So thread `t2` is caught in the rat race. Thread `t2` has no possibility to set the value of the flag to `false`, so `t2` must wait until thread `t1` executes the `unlock` method and sets the flag to `false` (lines 15 - 17).

2. Thread `t` doesn't get the lock, so we are in scenario 1 with swapped roles.

I want you to focus your attention on the method `test_and_set` of `std::atomic_flag`. The method `test_and_set` consists of two operations: reading and writing. It's key that both operations are performed in one atomic operation. If not, we would have a read and a write on the shared resource (line 24). That is-- by definition-- a data race, and the program has undefined behavior.