Atomics guarantee two characteristics. At one hand, they are atomic, at the other hand, they provide synchronisation and order constraints on the program execution.

I introduced in the last post the sequential consistency as the default behaviour of atomic operations. But, what does that mean? You can specify for each atomic operation the memory order. If not specified, std::memory\_order\_seq\_cst is used.

So this piece of code

x.store(1);

res= x.load();

is equivalent to the following piece of code.

x.store(1,std::memory\_order\_seq\_cst);

res= x.load(std::memory\_order\_seq\_cst);

For simplicity reasons, I will use the first spelling in this post.

std::atomic\_flag

std::atomic\_flag has a simple interface. It's method clear enables you the set its value to false, with test\_and\_set back to true. In case you use test\_and\_set you get the old value back. To use std::atomic\_flag it must be initialized to false with the constant ATOMIC\_FLAG\_INIT. That is not so thrilling. But std::atomic\_flag has two very interesting properties.

**std::atomic\_flag** is

* the only lock-free atomic.
* the building block for higher thread abstractions.

The only lock-free atomic? The remaining more powerful atomics can provide their functionality by using a [mutex.](http://modernescpp.com/index.php/the-risk-of-mutexes) That is according to the C++ standard.  So these atomics have a method is\_lock\_free to check if the atomic uses internally a mutex. On the popular platforms, I always get the answer false. But you should be aware of that.

The interface of a std::atomic\_flag is sufficient to build a spinlock. With a spinlock, you can protect a [critical section](http://modernescpp.com/index.php/component/content/article?id=157:threads-sharing-data&catid=35:c&Itemid=239#CriticalSection) similar to a mutex. But the spinlock will not passively wait in opposite to a mutex until it gets it mutex. It will eagerly ask for the critical section. So it saves the expensive context change in the wait state but it fully utilises the CPU:

The example shows the implementation of a spinlock with the help of std::atomic\_flag.

// spinLock.cpp

#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();

}

int main(){

std::thread t(workOnResource);

std::thread t2(workOnResource);

t.join();

t2.join();

}

The both threads t and t2 (line 31 and 32) are fighting for the critical section. For simplicity reasons the critical section in line 24 consists only of a comment. How does it work? The class Spinlock has - similar to a mutex - the two methods lock and unlock. In addition to that the constructor of Spinlock initializes in line 9 the std::atomic\_flag to false. In case thread t wants to execute the function workOnRessource two scenarios can happen.

**First,** the thread t gets the lock. So 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. That value true is the value, the while loop returns to the other 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 eagerly wait until thread t1 executes the unlock method and sets the flag to false (line 15 - 17).

**Second,** the thread t don't get the lock. So we are in scenario 1 with changed roles.

It's very interesting to compare the active waiting of a spinlock with the passive waiting of a mutex.

## Spinlock versus Mutex

What is happening to the CPU load, if the function workOnRessoucre locks the spinlock for 2 seconds (line 23 - 25)?

// spinLockSleep.cpp

#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();

std::this\_thread::sleep\_for(std::chrono::milliseconds(2000));

spin.unlock();

}

int main(){

std::thread t(workOnResource);

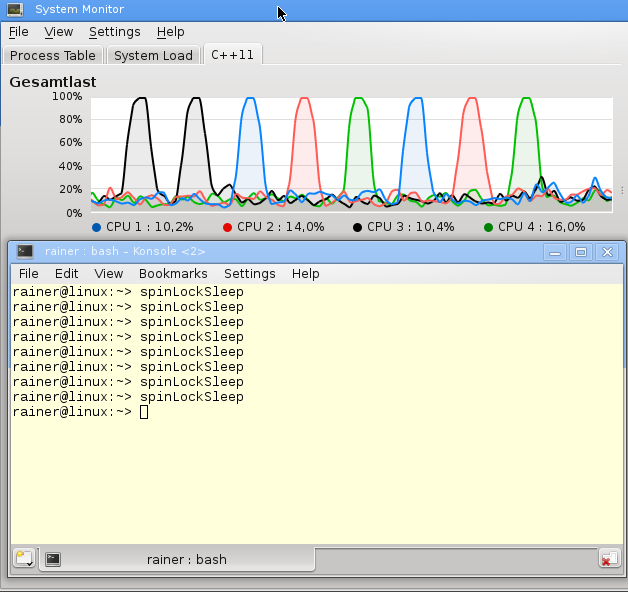
std::thread t2(workOnResource);

t.join();

t2.join();

}

In case the theory is right, one of the four cores of my PC must be fully utilized. Exactly that you can see in the screenshot.



The screenshot shows in a nice way, that the load of one core gets 100%. But my PC is fair. Each time a different core has to perform the busy waiting.

I use in the next concise program a mutex instead of a spinlock.

#include <mutex>

#include <thread>

std::mutex mut;

void workOnResource(){

mut.lock();

std::this\_thread::sleep\_for(std::chrono::milliseconds(5000));

mut.unlock();

}

int main(){

std::thread t(workOnResource);

std::thread t2(workOnResource);

t.join();

t2.join();

}

Although I execute the program several times, I can not observe a higher load of the cores.

