


```
class realtime_mutex
{
    std::atomic_flag flag = ATOMIC_FLAG_INIT;

public:
    // Non-blocking try_lock for real-time contexts
    bool try_lock() noexcept
    {
        // First test: cheap read-only check
        if (flag.test (std::memory_order_relaxed))
            return false; // Already locked

        // Then test-and-set if the first test passed
        return ! flag.test_and_set (std::memory_order_acquire);
    }

    // Blocking lock with test, test-and-set optimization
    void lock() noexcept
    {
        while (true)
        {
            // Optimistically assume the lock is free on the first try
            if (! flag.test_and_set (std::memory_order_acquire))
                break; // Successfully acquired the lock

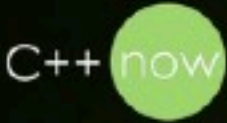
            // Wait for lock to be released without generating cache misses
            while (flag.test (std::memory_order_relaxed))
                CPU_PAUSE(); // CPU-specific pause instruction
        }
    }

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


<https://rightorp.se/spinlock/>

Lightning Talk: A Spinlock Implementation

- Fedor Pikus
- CppNow 2022





2022
MAY 1-6
Aspen, Colorado, USA



Fedor Pikus

A Spinlock Implementation



```
#ifndef SPINLOCK
#define SPINLOCK

#include <atomic>
#include <time.h>

class spinlock {
public:
    spinlock() : flag_(0) {}
    void lock() {
        static const timespec ns = { 0, 1 };
        for (int i = 0; flag_.load(std::memory_order_relaxed) || flag_.exchange(1, std::memory_order_acquire); ++i) {
            if (i == 8) {
                i = 0;
                nanosleep(&ns, NULL);
            }
        }
    }
    void unlock() { flag_.store(0, std::memory_order_release); }
private:
    std::atomic<unsigned int> flag_;
};

#endif // SPINLOCK
```

spinlock.h [cpp] 5,16 All






```

class realtime_mutex
{
    std::atomic_flag flag = ATOMIC_FLAG_INIT;

public:
    // Non-blocking try_lock for real-time contexts
    bool try_lock() noexcept
    {
        // First test: cheap read-only check
        if (flag.test (std::memory_order_relaxed))
            return false; // Already locked

        // Then test-and-set if the first test passed
        return ! flag.test_and_set (std::memory_order_acquire);
    }

    // Blocking lock with test, test-and-set optimization
    void lock() noexcept
    {
        while (true)
        {
            // Optimistically assume the lock is free on the first try
            if (! flag.test_and_set (std::memory_order_acquire))
                break; // Successfully acquired the lock

            // Wait for lock to be released without generating cache misses
            while (flag.test (std::memory_order_relaxed))
                CPU_PAUSE(); // CPU-specific pause instruction
        }
    }

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

```

<https://rigtorp.se/spinlock/>

Lightning Talk: A Spinlock Implementation

- Fedor Pikus
- CppNow 2022



182,987,000

x4.5

41,400,000

20,178,300

x0.5

2.00E+08

1.50E+08

1.00E+08

5.00E+07

0.00E+00

queue

mutex_queue

realtime_mutex_queue

