



# <atomic> <future>

Wielowątkowość we współczesnym C++

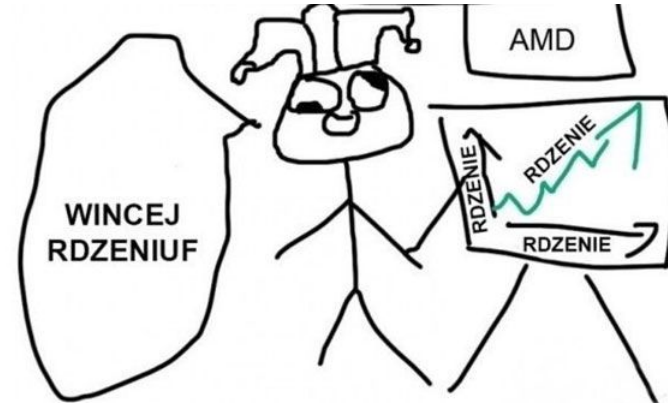
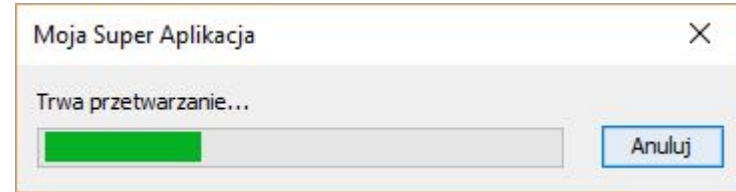
Mateusz Kwiatkowski

Marcin Radomski

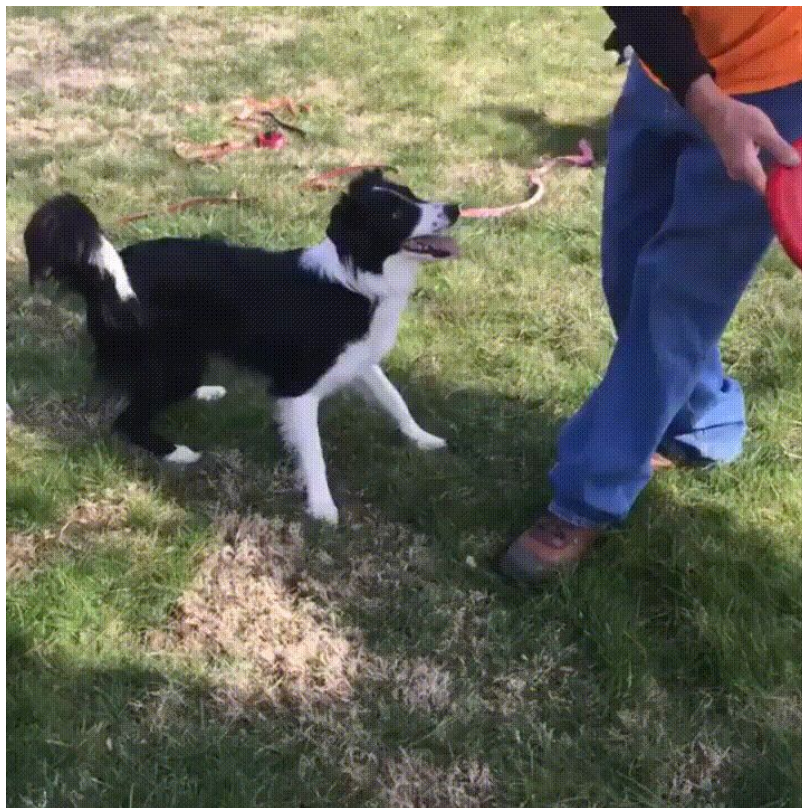
<https://goo.gl/ei5pJr>

# Czy na pewno chcesz programować równoległe?

- Asynchroniczne operacje
  - Event loop
  - Komunikacja międzyprocesowa
- Wydajność
  - Tak!
- Alternatywne optymalizacje
  - Cache-friendliness
  - Najpierw zmierz!

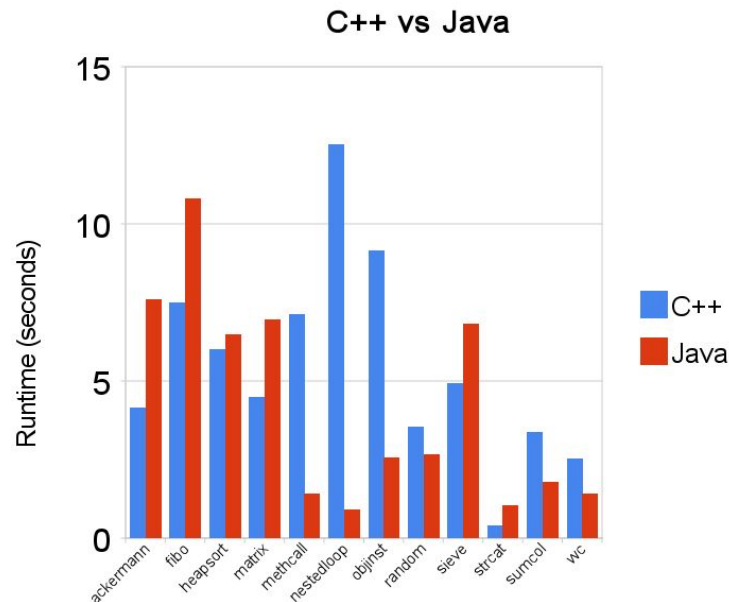


Czy na pewno chcesz programować równolegle?



# Dlaczego chcesz programować w C++?

- C++ to niewdzięczny język
  - potężne abstrakcje...
  - ...ale kompletny brak zabezpieczeń
- Może jednak Java/.NET?
  - „Kod natywny jest wydajniejszy”
- Współcześniejsze alternatywy
  - Go
  - Rust
  - ...
- A kiedy brakuje zasobów?
  - Może jednak C?



# Programowanie równoległe przed C++11





# Programowanie równoległe przed C++11

- API dostępne też w C:
  - pthreads
  - OpenMP
  - API systemowe (`clone(CLONE_THREAD)`, `CreateThread()`)
- Zewnętrzne biblioteki
  - `boost::thread`
  - Intel TBB (Threading Building Blocks)
  - Microsoft Parallel Patterns Library
  - [Sporo innych](#)
- Masywna równoległość (nie do końca C/C++)
  - CUDA
  - OpenCL

# Programowanie równoległe w C++11

- `std::thread`
- `std::mutex`
- `std::condition_variable`
- `std::atomic`
- `std::promise`, `std::future`

# Jak (nie) używać std::thread

```
int main() {
    std::thread fred([](){
        while (std::cin) {
            int n;
            std::cin >> n;
            std::cout << n * n << std::endl;
        }
    });
}
```



```
marcin@marian:/tmp/test$ g++ -Wall -Wextra -std=c++11 -o test test.cpp -lpthread
marcin@marian:/tmp/test$ ./test
terminate called without an active exception
Aborted (core dumped)
```



# Jak używać std::thread

```
int main() {  
    std::thread fred([](){  
        while (std::cin) {  
            int n;  
            std::cin >> n;  
            std::cout << n * n << std::endl;  
        }  
    });  
  
    fred.join(); // lub fred.detach();  
}
```

- Dokumentacja std::thread:
  - If \*this has an associated thread (joinable() == true), std::terminate() is called.
- Uwaga na thread::detach!
  - Odpięte wątki giną po zakończeniu procesu

# Współdzielenie stanu między wątkami

```
int main() {  
    volatile int n = 0;  
  
    std::thread fred([&n]() {  
        for (int i = 0; i < 10000; ++i) {  
            ++n;  
        }  
    });  
  
    for (int i = 0; i < 10000; ++i) {  
        ++n;  
    }  
  
    fred.join();  
    std::cout << n << std::endl;  
}
```

```
marcin@marian:/tmp/test$ ./test  
16403  
marcin@marian:/tmp/test$ ./test  
18380  
marcin@marian:/tmp/test$ ./test  
19280  
marcin@marian:/tmp/test$ ./test  
17075  
marcin@marian:/tmp/test$ ./test  
20000  
marcin@marian:/tmp/test$ ./test  
18581  
marcin@marian:/tmp/test$ ./test  
18700
```

# Jak (nie) używać std::mutex

```
std::mutex mtx;  
  
void foo();  
  
void thread_safe_foo() {  
    mtx.lock();  
    foo();  
    mtx.unlock();  
}
```

# Jak (nie) używać std::mutex

```
std::mutex mtx;
```

```
void foo() {  
    mtx.lock(); // oops  
    // ...  
    mtx.unlock();  
}
```

```
void thread_safe_foo() {  
    mtx.lock();  
    foo();  
    mtx.unlock();  
}
```

- lock() na już zablokowanym std::mutex = UB
  - Może zakleszczyć wątek
  - Może rzucić wyjątek
  - Może zadziałać (!)
  - Może zrobić **cokolwiek**

- <http://en.cppreference.com/w/cpp/thread/mutex/lock>

Więcej o Undefined Behavior w C/C++:

<http://blog.llvm.org/2011/05/what-every-c-programmer-should-know.html>

# Jak (nie) używać std::recursive\_mutex

```
std::recursive_mutex mtx;
```

```
void foo() {  
    mtx.lock(); // OK?  
    // ...  
    mtx.unlock();  
}
```

```
void thread_safe_foo() {  
    mtx.lock();  
    foo();  
    mtx.unlock();  
}
```

- Więcej niż 1 lock() z tego samego wątku?
- std::recursive\_mutex
- 1 lock() - 1 unlock()

# Muteksy + wyjątki

```
std::recursive_mutex mtx;  
  
void foo() {  
    throw std::runtime_error("oops");  
}  
  
void thread_safe_foo() {  
    mtx.lock();  
    foo();  
    mtx.unlock();  
}
```



# Muteksy + wyjątki

```
std::recursive_mutex mtx;
```

```
void foo() {  
    throw std::runtime_error("oops");  
}
```

```
void thread_safe_foo() {  
    mtx.lock();  
    foo();  
    mtx.unlock(); // nope  
}
```

- Ten sam problem dotyczy też
  - bibliotek w C
  - OpenMP



# Muteksy + wyjątki + std::lock\_guard

```
std::recursive_mutex mtx;
```

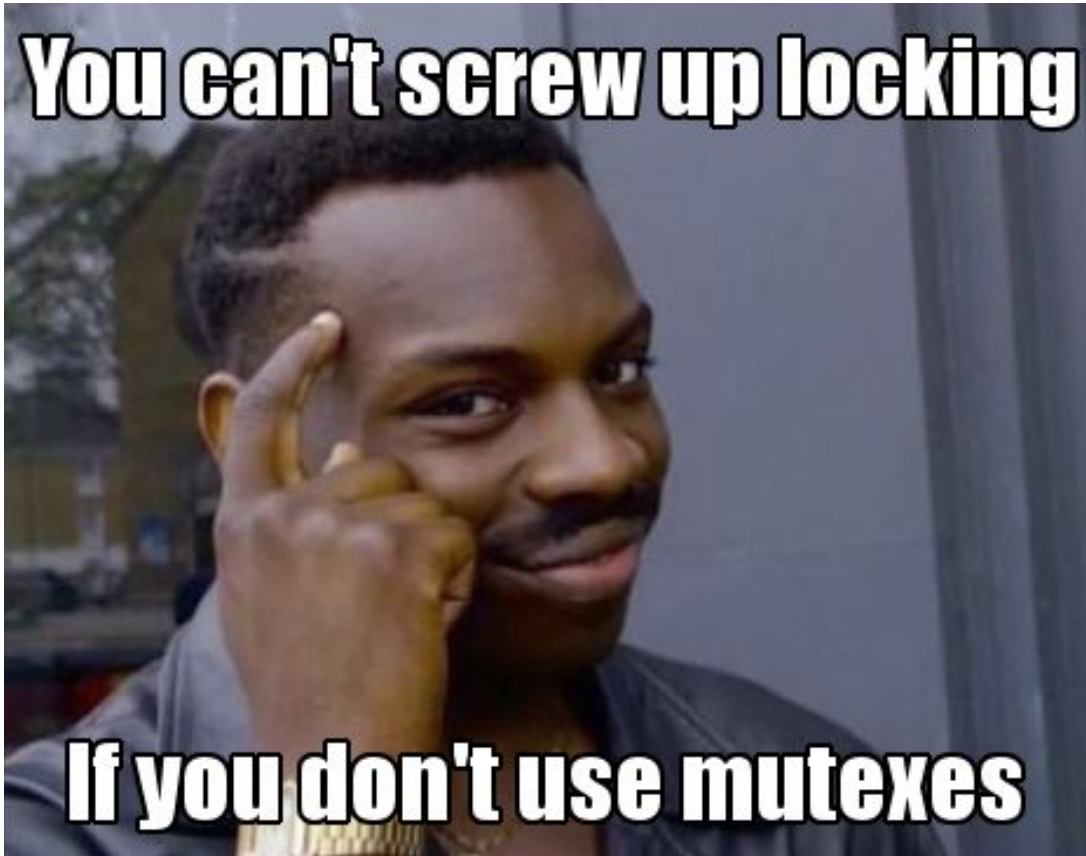
```
void foo() {  
    throw std::runtime_error("oops");  
}
```

```
void thread_safe_foo() {  
    std::lock_guard<std::recursive_mutex> lock(mtx);  
    foo();  
}
```

- Nie używaj lock()/unlock() bezpośrednio
- RAII twoim przyjacielem







# std::atomic

- Implementacja:
  - Blokująca
  - Implementacja niewstrzymywana (lock-free)
    - `std::atomic_is_lock_free()`



# std::condition\_variable

```
std::mutex mtx;  
std::condition_variable cv;  
  
std::thread fred([&]() {  
    std::chrono::seconds delay(5);  
    std::this_thread::sleep_for(delay);  
  
    cv.notify_one();  
});  
  
std::unique_lock<std::mutex> lock(mtx);  
cv.wait(lock);  
  
fred.join();
```

# std::condition\_variable

```
std::mutex mtx;  
std::condition_variable cv;  
  
std::thread fred([&]() {  
    std::chrono::seconds delay(5);  
    std::this_thread::sleep_for(delay);  
  
    cv.notify_one();  
});  
  
std::chrono::seconds delay(6);  
std::this_thread::sleep_for(delay);  
  
std::unique_lock<std::mutex> lock(mtx);  
cv.wait(lock); // oops  
  
fred.join();
```

# std::condition\_variable

```
int main() {
    std::mutex mtx;
    std::condition_variable cv;
    bool done = false;

    std::thread fred([&]() {
        {
            std::lock_guard<std::mutex> lock(mtx);
            done = true;
        }
        cv.notify_one();
    });

    std::chrono::seconds delay(1);
    std::this_thread::sleep_for(delay);

    {
        std::unique_lock<std::mutex> lock(mtx);
        while (!done) {
            cv.wait(lock);
        }
    }

    fred.join();
}
```

# std::condition\_variable

```
bool wait_until( std::unique_lock<std::mutex>& lock,  
                const std::chrono::time_point<...>& timeout_time,  
                Predicate pred );
```

- Można podać warunek końca jako argument
- Wymaga ustalenia skończonego timeoutu
  - Jako bezwzględny punkt w czasie!

# std::condition\_variable

```
int main() {
    std::mutex mtx;
    std::condition_variable cv;
    bool done = false;

    std::thread fred([&]() {
        {
            std::lock_guard<std::mutex> lock(mtx);
            done = true;
        }
        cv.notify_one();
    });

    std::chrono::seconds delay(1);
    std::this_thread::sleep_for(delay);

    auto WAIT_END = std::chrono::steady_clock::now() + std::chrono::minutes(1);
    {
        std::unique_lock<std::mutex> lock(mtx);
        cv.wait_until(lock, WAIT_END, [&]() { return done; });
    }

    fred.join();
}
```

# std::condition\_variable: haczyki

- std::condition\_variable jest ściśle związany z muteksem
  - Zawołanie wait() przy zwolnionym muteksie: UB
    - notify() bez muteksa: OK
  - Zawołanie wait() przy zablokowanym innym muteksie niż inne wątki korzystające z tego samego std::condition\_variable: UB
- notify() nie robi nic jeżeli żaden wątek akurat nie robi wait()
  - Kolejne wywołania wait() nadal blokują!



# Synchronizacja stanu w pigułce

- `volatile` nie służy do synchronizacji!
- Muteksy są użyteczne, ale trzeba na nie uważać
  - `mtx.lock(); mtx.lock();` == UB (chyba, że `std::recursive_mutex`)
  - Brak `unlock()` po `lock()` == deadlock albo UB
  - Brak `lock()` przed `unlock()` == UB
- Jak nie strzelać sobie w stopę?
  - Synchronizowany dostęp do zmiennej typu prymitywnego? `std::atomic<T>`
  - Używaj `std::recursive_mutex` zamiast `std::mutex`
    - chyba że **naprawdę** dobrze wiesz co robisz
    - ...albo używasz `std::condition_variable`
  - Używaj RAII wrapperów typu `std::lock_guard`

# std::promise, std::future

```
int main() {  
    int a = 10, b = 5;  
  
    std::promise<int> sum_promise;  
    std::future<int> sum = sum_promise.get_future();  
    std::thread sum_thread([&sum_promise, a, b]() {  
        sum_promise.set_value_at_thread_exit(a + b);  
    });  
  
    int diff = a - b;  
  
    std::cout << "(a + b) * (a - b) = " << sum.get() * diff << std::endl;  
  
    sum_thread.join();  
}
```

# std::async

```
int main() {  
    int a = 10, b = 5;  
  
    std::future<int> sum = std::async([=]() { return a + b; });  
    std::future<int> diff = std::async(std::launch::async, [=]() { return a - b; });  
  
    std::cout << "(a + b) * (a - b) = " << sum.get() * diff.get() << std::endl;  
}
```

- std::launch::async - nowy wątek
- std::launch::deferred - leniwa ewaluacja
- default – oba jednocześnie – implementacja może wybrać **którekolwiek**

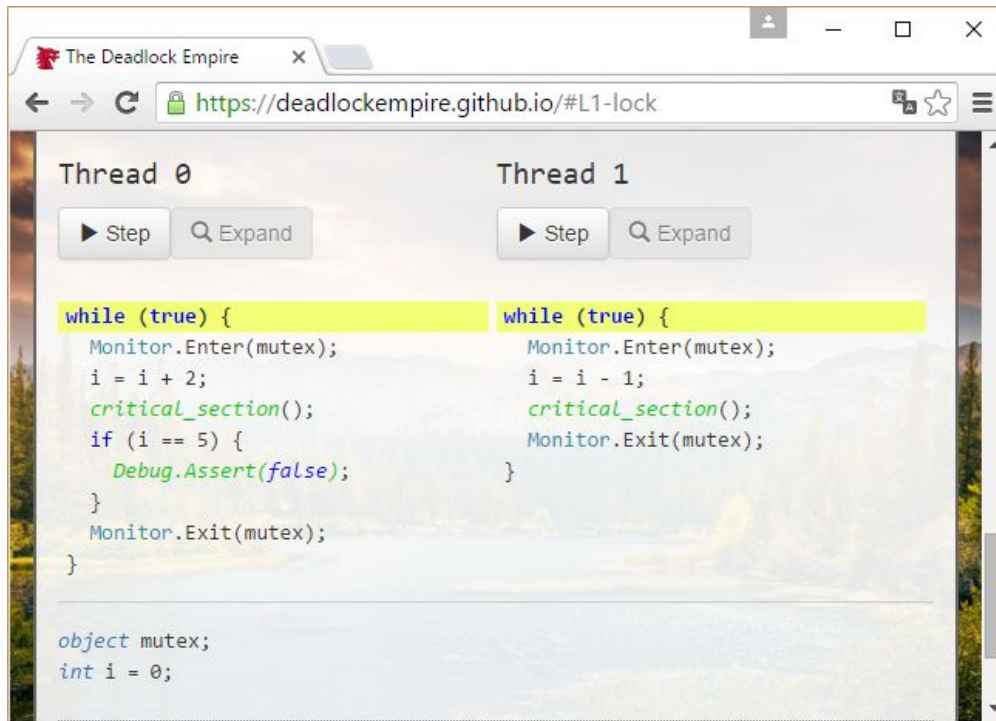
# Programowanie równoległe w C++17

- `std::execution_policy` i równoległe algorytmy STL

```
std::vector<int> ints{1, 4, 2, 6, 2, 3, 4};  
std::sort(std::execution::par, ints.begin(), ints.end());
```

- Haczyk: jeszcze nie ma dostępnych implementacji
  - <https://parallelstl.codeplex.com/>

# Współbieżność jest trudna



The screenshot shows a web browser window titled "The Deadlock Empire" with the URL <https://deadlockempire.github.io/#L1-lock>. The page displays two threads, Thread 0 and Thread 1, with their respective code snippets. Thread 0's code increments a counter 'i' by 2 and enters a critical section when 'i' equals 5. Thread 1's code decrements 'i' by 1 and exits the critical section. The background of the code editor shows a scenic lake and mountains.

```

Thread 0
while (true) {
    Monitor.Enter(mutex);
    i = i + 2;
    critical_section();
    if (i == 5) {
        Debug.Assert(false);
    }
    Monitor.Exit(mutex);
}

object mutex;
int i = 0;

Thread 1
while (true) {
    Monitor.Enter(mutex);
    i = i - 1;
    critical_section();
    Monitor.Exit(mutex);
}

```

<https://deadlockempire.github.io>

# Współbieżność jest trudna

```
std::mutex m1;
std::mutex m2;

void deadlock1() {
    {
        std::lock_guard<std::mutex> lock1(m1);
        {
            std::lock_guard<std::mutex> lock2(m2);
        }
    }
}

void deadlock2() {
    {
        std::lock_guard<std::mutex> lock1(m2);
        {
            std::lock_guard<std::mutex> lock2(m1);
        }
    }
}

int main() {
    auto job1 = std::async(std::launch::async, deadlock1);
    auto job2 = std::async(std::launch::async, deadlock2);
    job1.get(); job2.get();
}
```



# Współbieżność jest trudna - Helgrind

```
$ g++ -g deadlock.cpp -o deadlock -pthread
$ valgrind --tool=helgrind ./deadlock
==2129== Helgrind, a thread error detector
[...]
```

==2129== Thread #3: lock order "0x3170E0 before 0x317120" violated

==2129==

==2129== Observed (incorrect) order is: acquisition of lock at 0x317120

==2129== at 0x4C3109C: ??? (in /usr/lib/valgrind/vgpreload\_helgrind-amd64-linux.so)

==2129== by 0x10A28E: \_\_gthread\_mutex\_lock(pthread\_mutex\_t\*) (gthr-default.h:748)

==2129== by 0x10A9B3: std::mutex::lock() (std\_mutex.h:103)

==2129== by 0x10B5BD: std::lock\_guard<std::mutex>::lock\_guard(std::mutex&) (std\_mutex.h:162)

==2129== by 0x10A481: deadlock2() (deadlock.cpp:21)

[...]

==2129==

==2129== followed by a later acquisition of lock at 0x3170E0

==2129== at 0x4C3109C: ??? (in /usr/lib/valgrind/vgpreload\_helgrind-amd64-linux.so)

==2129== by 0x10A28E: \_\_gthread\_mutex\_lock(pthread\_mutex\_t\*) (gthr-default.h:748)

==2129== by 0x10A9B3: std::mutex::lock() (std\_mutex.h:103)

==2129== by 0x10B5BD: std::lock\_guard<std::mutex>::lock\_guard(std::mutex&) (std\_mutex.h:162)

==2129== by 0x10A494: deadlock2() (deadlock.cpp:23)

[...]

==2129==

==2129== Required order was established by acquisition of lock at 0x3170E0

==2129== at 0x4C3109C: ??? (in /usr/lib/valgrind/vgpreload\_helgrind-amd64-linux.so)

==2129== by 0x10A28E: \_\_gthread\_mutex\_lock(pthread\_mutex\_t\*) (gthr-default.h:748)

# Współbieżność jest trudna - ThreadSanitizer

```
$ clang++ -std=c++11 -g deadlock.cpp -o deadlock -fsanitize=thread
```

```
$ ./deadlock
```

```
=====
```

```
WARNING: ThreadSanitizer: lock-order-inversion (potential deadlock) (pid=4154)
```

```
  Cycle in lock order graph: M8 (0x000001542548) => M9 (0x000001542570) => M8
```

```
  Mutex M9 acquired here while holding mutex M8 in thread T1:
```

```
    #0 pthread_mutex_lock <null> (deadlock+0x000000439220)
```

```
    #1 __gthread_mutex_lock(pthread_mutex_t*) gthr-default.h:748 (deadlock+0x0000004a4f66)
```

```
    #2 std::mutex::lock() std_mutex.h:103 (deadlock+0x0000004a65b8)
```

```
    #3 lock_guard std_mutex.h:162 (deadlock+0x0000004a51e5)
```

```
    #4 deadlock1() deadlock.cpp:14 (deadlock+0x0000004a4c60)
```

```
    [...]
```

```
  Hint: use TSAN_OPTIONS=second_deadlock_stack=1 to get more informative warning message
```

```
  Mutex M8 acquired here while holding mutex M9 in thread T2:
```

```
    #0 pthread_mutex_lock <null> (deadlock+0x000000439220)
```

```
    #1 __gthread_mutex_lock(pthread_mutex_t*) gthr-default.h:748 (deadlock+0x0000004a4f66)
```

```
    #2 std::mutex::lock() std_mutex.h:103 (deadlock+0x0000004a65b8)
```

```
    #3 lock_guard std_mutex.h:162 (deadlock+0x0000004a51e5)
```

```
    #4 deadlock2() /tmp/deadlock.cpp:23 (deadlock+0x0000004a4cd0)
```

```
    [...]
```

# Helgrind, ThreadSanitizer

- Wykrywają potencjalne problemy, nawet jeśli nie nastąpiły
- Zdarzają się fałszywe alarmy
- Nie są w stanie wykryć wszystkich problemów
  - Testowanie na obu programach może mieć sens
- Analizują tylko rzeczywiście wykonane fragmenty kodu
  - Testy, testy, testy!



# Wzorzec: then()

```
#define BOOST_THREAD_VERSION 4
#include <boost/thread/future.hpp>
#include <cstdlib>

std::string recv_message() {
    // let's pretend it receives from the network
    boost::this_thread::sleep_for(boost::chrono::seconds(5));
    return "Hello, world!";
}

int main() {
    boost::async([]() { return recv_message(); })
        .then([](boost::future<std::string> f) { std::cout << f.get() << std::endl; })
        .then([](boost::future<void> f) { std::exit(0); });

    while (true) {
        boost::this_thread::sleep_for(boost::chrono::seconds(10));
    }
}
```

## Wzorzec: then() – pułapki

- Kiedyś może wejdzie do std...
  - Póki co można użyć Boost.Thread lub Microsoft PPL
- `boost::async` vs. `std::async`
  - future zwrócony z `std::async` **blokuje** aż do zakończenia wątku!
  - future zwrócony z `boost::async` **nie**
  - wyjście z `main()` **zabija wszystkie wątki Boosta!**

# Wzorzec: współbieżna kolejka

```
boost::concurrent::sync_queue<int> queue;

void producer_thread() {
    while (true) {
        boost::this_thread::sleep_for(boost::chrono::milliseconds(rand() % 5000));
        queue.push(rand());
    }
}

void consumer_thread() {
    while (true) {
        std::cout << queue.pull() << std::endl;
    }
}

int main() {
    for (int i = 0; i < 5; ++i) {
        boost::thread(producer_thread).detach();
    }
    for (int i = 0; i < 5; ++i) {
        boost::thread(consumer_thread).detach();
    }

    while (true) {
        boost::this_thread::sleep_for(boost::chrono::seconds(10));
    }
}
```

# Wzorzec: współbieżna kolejka

```
class SomeObject {
    boost::concurrent::sync_queue<std::function<void(SomeObject &)>> queue_;
    boost::thread thread_;

    void thread() {
        while (true) {
            queue_.pull()(*this);
        }
    }
public:
    SomeObject() : thread_([this]() { this->thread(); }) {}

    void queue(std::function<void(SomeObject &)> task) {
        queue_.push(task);
    }
};
```

# C++ Actor Framework

- Sprawdzony paradygmat aktorowy
  - Przez Javowców i Scalowców ([www.akka.io](http://www.akka.io))
  - Przez .NETowców ([www.getakka.net](http://www.getakka.net))
- **Aktor** – enkapsuluje stan i zachowanie
  - Framework zajmuje się przekazywaniem komunikatów
  - Przekazywanie komunikatów **zamiast** wywoływania metod



# C++ Actor Framework

```
#include "caf/all.hpp"

using read_atom = caf::atom_constant<caf::atom("read")>;

class SimpleActor : public caf::event_based_actor {
    std::string state_;
public:
    SimpleActor(caf::actor_config &cfg) : caf::event_based_actor(cfg) {}

    caf::behavior make_behavior() override {
        return {
            [this](std::string new_value) {
                state_ = new_value;
            },
            [this](read_atom) {
                return state_;
            }
        };
    }
};
```

# C++ Actor Framework

```
int main() {  
    caf::actor_system_config cfg;  
    caf::actor_system system(cfg);  
    auto actor = system.spawn<SimpleActor>();  
    system.spawn([=](caf::event_based_actor *self) {  
        self->send(actor, std::string("Hello, world"));  
        self->request(actor, std::chrono::seconds(10), read_atom())  
            .then([](const std::string &value) {  
                std::cout << value << std::endl;  
            });  
    });  
}
```

# Więcej o wielowątkowości w C++

- Anthony A. Williams, “C++ Concurrency in Action: Practical Multithreading”
- <https://herbsutter.com/2013/01/15/videos-panel-and-c-concurrency/>
- <https://herbsutter.com/2013/02/11/atomic-weapons-the-c-memory-model-and-modern-hardware/>

# Pytania?

<https://goo.gl/ei5pJr>