



Uniwersytet  
Wrocławski

# Programowanie współbieżne w C++11

Zbigniew Koza

Wydział Fizyki i Astronomii

`std::async`


# std::async

```
#include <iostream>
#include <string>
#include <future>
#include <chrono>

double my_fun(int n)
{
    double s = 0.0;
    for (int i = 1; i <= n; i += 2)
        s += 1.0/i - 1.0/(i+1);
    return s;
}

int main()
{
    constexpr int n = 1000000000;
    auto fut = std::async(my_fun, n);
    std::cout << "Czekam na wynik...\n";
    auto value = fut.get();
    std::cout << value << "\n";
}
```

Zwykła funkcja C++



- **std::async**  
to wrapper dla  
**std::thread**  
i **std::promise**
- Zwraca  
**std::future**
- **future::get**  
zwraca wartość  
zwróconą w funkcji  
wywołanej  
asynchronicznie

# Cechy `std::async`

- Zalety:
  - Prostota
- Wady:
  - Liczne niespodzianki w dokumentacji i dostępnych implementacjach
  - Łatwość napisania kodu, który zachowuje się wbrew zdroworozsądkowym oczekiwaniom...

# std::launch::async/deferred

```
void print_ten (char c, int ms) {
    for (int i=0; i<10; ++i) {
        std::this_thread::sleep_for (std::chrono::milliseconds(ms));
        std::cout << c;
    }
}

int main ()
{
    std::cout << "with launch::async:\n";
    std::future<void> foo = std::async (std::launch::async, print_ten, '*', 100);
    std::future<void> bar = std::async (std::launch::async, print_ten, '@', 200);
    // async "get" (wait for foo and bar to be ready):
    foo.get();
    bar.get();
    std::cout << "\n\n";

    std::cout << "with launch::deferred:\n";
    foo = std::async (std::launch::deferred, print_ten, '*', 100);
    bar = std::async (std::launch::deferred, print_ten, '@', 200);
    // deferred "get" (perform the actual calls):
    foo.get();
    bar.get();
    std::cout << '\n';

    return 0;
}
```

# std::launch:async/deferred

```
void print_ten (char c, int ms) {
    for (int i=0; i<10; ++i) {
        std::this_thread::sleep_for (std::chrono::milliseconds(ms));
        std::cout << c;
    }
}

int main ()
{
    std::cout << "with launch::async:\n";
    std::future<void> foo = std::async (std::launch::async, print_ten, '*', 100);
    std::future<void> bar = std::async (std::launch::async, print_ten, '@', 200);
    // async "get" (wait for foo and bar to be ready):
    foo.get();
    bar.get();
    std::cout << "\n\n";

    std::cout << "with launch::deferred:\n";
    foo = std::async (std::launch::deferred, print_ten, '*', 100);
    bar = std::async (std::launch::deferred, print_ten, '@', 200);
    // deferred "get" (perform the actual calls):
    foo.get();
    bar.get();
    std::cout << '\n';

    return 0;
}
```

Bariery

Tu faktyczne uruchomienie

synchronicznie (szeregowo)

Possible output:

```
with launch::async:
**@**@**@*@**@*@@@@@

with launch::deferred:
*****@@@@@@@@@@@@@
```

# `std::launch:async/deferred`

- `std::launch::async` – natychmiastowe uruchomienie funkcji w osobnym wątku
- `std::launch::deferred` – uruchomienie funkcji przy pierwszym wywołaniu `wait` lub `get` na obiekcie `std::future`
- Brak tego parametru – sposób wywołania funkcji zależy od implementacji :-)
- Mój komputer (Ubuntu 16.04 64 bit):  
*deferred is preferred*

# `std::async` „bez wartości”

```
std::cout << "with launch::async and temporaries:\n";  
std::async (std::launch::async, print_ten, '*', 100);  
std::async (std::launch::async, print_ten, '@', 200);  
std::cout << "\n";
```

- Powyższe dwa wywołania `std::async` spowodują:
  - Asynchroniczne uruchomienie funkcji `print_ten`
  - I natychmiastową blokadę na destruktorze *tymczasowego* `std::future` do zakończenia `print_ten`
- Efektywnie będą więc synchroniczne



# **std::shared\_future**

- Jest to „future” wielokrotnego dostępu
- Ale po co, skoro promise może stan shared memory ustalić tylko raz?
- Przydatne, jeśli do tych samych danych (zapisanych przez jakiś promise) należy dać dostęp kilku wątkom...

# Wątek roboczy z **shared\_future**

```
void th_fun(std::shared_future<void> sfut, int id)
{
    std::ofstream F("parallel.txt", std::ios::app);
    while(sfut.wait_for(std::chrono::seconds(0)) != std::future_status::ready)
    {
        for (volatile long int i = 0; i < id*1'000'000'000L+ 390'005'000; i++)
            continue;
        F << std::to_string(id) + "\n" << std::flush;
    };
    std::cout << "bye from " + std::to_string(id) + "\n";
}
```

- W powyższym kodzie wątek roboczy w pętli „coś robi”, od czasu do czasu monitorując stan swojego **shared\_future**

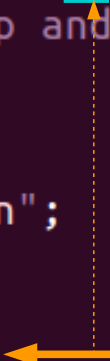
# Dygresja (C++, C++14)

```
void th_fun(std::shared_future<void> sfut, int id)
{
    std::ofstream F("parallel.txt", std::ios::app);
    while(sfut.wait_for(std::chrono::seconds(0)) != std::future_status::ready)
    {
        for (volatile long int i = 0; i < id*1'000'000'000L+ 390'005'000; i++)
            continue;
        F << std::to_string(id) + "\n" << std::flush;
    };
    std::cout << "bye from " + std::to_string(id) + "\n";
}
```

- **volatile** – zakaz optymalizacji operacji na obiektach z tym modyfikatorem
- 1'000'000 – lukier składniowy C++14
- 10000000000L – typ literału (tu: **long** int)

# Wątek główny z `shared_future`

```
int main()
{
    constexpr int N = 4;
    std::promise<void> prms;
    std::shared_future<void> sf = prms.get_future().share();
    std::vector<std::thread> v;
    for (int i = 0; i < 4; i++)
        v.emplace_back(th_fun, sf, i+1);
    std::cout << "all is up and running\nNow waiting for q...\n";
    char c;
    while (std::cin >> c)
    {
        std::cout << c << "\n";
        if (c == 'q')
        {
            prms.set_value();
            break;
        }
    }
    for (auto & th: v)
        th.join();
}
```



Sygnal dla wątków roboczych

The diagram consists of a yellow box containing the text "Sygnal dla wątków roboczych" (Signal for worker threads). A yellow arrow points from this box to the `prms.set_value();` line in the code. A dashed yellow arrow points from the `sf` variable in the `v.emplace_back` call to the `prms.set_value();` line, indicating the shared future's state change.

# Promise–future

- W poprzednim przykładzie:
  - Kierunek przepływu informacji był **odwrotny**:  
od wątku głównego do wątków roboczych
  - Tę informację wątek główny przekazał  
**z opóźnieniem** (czego nie da się zrobić np. poprzez argumenty funkcji)

# Dygresja (C++11)

```
v.emplace_back(th_fun, sf, i+1);
```

- Składowe kontenerów z nazwą **emplace** np. **emplace\_back**, **emplace**, etc. służą do jednoczesnej konstrukcji i dodania obiektu do kontenera (bez osobnej konstrukcji i kopiowania!)
- Charakterystyczne dla kontenerów STL w C++11

Sekcje krytyczne

# Przypomnienie: wyścig

```
#include <iostream>
#include <thread>

void my_fun(char c, int n)
{
    for (int i = 0; i < n; i++)
        std::cout << c;
    std::cout << "\n";
}

int main()
{
    std::thread th(&my_fun, 'w', 200);
    my_fun('m', 200);
    th.join();
}
```

- W tym programie dwa wątki jednocześnie modyfikują ten sam obiekt (std::cout)



# Przypomnienie: wyścig

- W tym programie dwa wątki jednocześnie modyfikują ten sam obiekt (std::cout)

```
#include <iostream>
#include <thread>

void my_fun(char c, int n)
{
    for (int i = 0; i < n; i++)
        std::cout << c;
    std::cout << "\n";
}

int main()
{
    std::thread th(&my_fun, 'w', 200);
    my_fun('m', 200);
    th.join();
}
```



```
00000000X0000000X0000X00
XXXXXXXXXXXXXXXXXXXX
```

# Rozwiązanie: `std::mutex`

```
#include <iostream>
#include <thread>
#include <mutex>

std::mutex mtx;

void my_fun(char c, int n)
{
    mtx.lock();
    for (int i = 0; i < n; i++)
        std::cout << c << std::flush;
    std::cout << "\n";
    mtx.unlock();
}

int main()
{
    std::thread th(&my_fun, 'x', 20);
    my_fun('o', 20);
    th.join();
}
```

- `std::mutex`  
zarządza „własną”  
**sekcją krytyczną**
- `lock()`
- `unlock()`

# Rozwiązanie: `std::mutex`

```
#include <iostream>
#include <thread>
#include <mutex>

std::mutex mtx;

void my_fun(char c, int n)
{
    mtx.lock();
    for (int i = 0; i < n; i++)
        std::cout << c << std::flush;
    std::cout << "\n";
    mtx.unlock();
}

int main()
{
    std::thread th(&my_fun, 'x', 20);
    my_fun('o', 20);
    th.join();
}
```

- `std::mutex`  
zarządza „własną”  
**sekcją krytyczną**
- `lock()`
- `unlock()`



000000000000000000000000

XXXXXXXXXXXXXXXXXXXXXX

# std::mutex

- „Mutex” = *mutual exclusion*
- Tylko jeden wątek może w danej chwili znajdować się wewnątrz sekcji krytycznej kodu zarządzanego mureksem
- Podstawowe, bardzo popularne narzędzie synchronizacji wątków

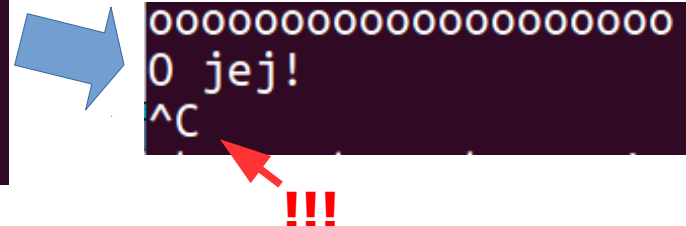
# std::mutex a wyjątki

```
std::mutex mtx;

void my_fun(char c, int n)
{
    try{
        mtx.lock();
        for (int i = 0; i < n; i++)
            std::cout << c << std::flush;
        std::cout << "\n";
        throw std::runtime_error("0 jej!");
        mtx.unlock();
    }
    catch(std::exception const& e)
    {
        std::cout << e.what() << "\n";
    }
}

int main()
{
    std::thread th(&my_fun, 'x', 20);
    my_fun('o', 20);
    th.join();
}
```

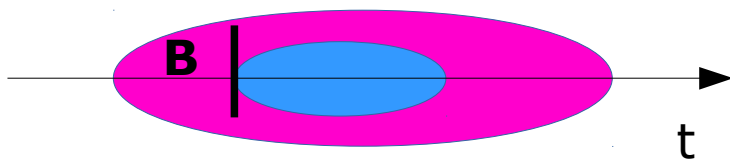
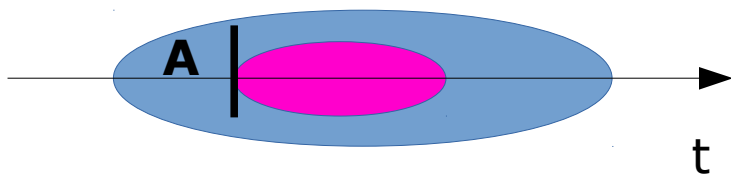
- Mutex nigdy nie zostanie odblokowany
- Drugi wątek nigdy nie wejdzie do sekcji krytycznej
- Deadlock!



```
00000000000000000000000000000000
0 jej!
^C
!!!
```

# Zakleszczenie (*deadlock*)

- **Zakleszczenie:** gdy jeden z wątków czeka na barierze zarządzanej przez inny wątek
- Przykład: dwie sekcje krytyczne organizowane w dwóch różnych miejscach w różnej kolejności
- Wtedy A czeka na B.unlock(), a B czeka na A.unlock().



# std::mutex a RAI

```
std::mutex mtx;

void my_fun(char c, int n)
{
    try{
        std::lock_guard<std::mutex> lg(mtx);
        for (int i = 0; i < n; i++)
            std::cout << c << std::flush;
        std::cout << "\n";
        throw std::runtime_error("0 jej!");
    }
    catch(std::exception const& e)
    {
        std::cout << e.what() << "\n";
    }
}

int main()
{
    std::thread th(&my_fun, 'x', 20);
    my_fun('o', 20);
    th.join();
}
```

- **lock\_guard** zamyka mutex w parze konstruktor/destruktor
- Zgodnie z RAI
- Wszystko działa!



```
oooooooooooooooooooooooooooo
0 jej!
xxxxxxxxxxxxxxxxxxxxxxxxxxxx
0 jej!
```



# std::unique\_lock

```
std::mutex mtx;

void my_fun(char c, int n)
{
    try{
        std::unique_lock<std::mutex> lg(mtx);
        for (int i = 0; i < n; i++)
            std::cout << c << std::flush;
        std::cout << "\n";
        throw std::runtime_error("O jej!");
    }
    catch(std::exception const& e)
    {
        std::cout << e.what() << "\n";
    }
}

int main()
{
    std::thread th(&my_fun, 'x', 20);
    my_fun('o', 20);
    th.join();
}
```

- **unique\_lock** działa jak **lock\_guard**
- Posiada dodatkowe funkcjonalności (lock, unlock...)
- Preferuj prostszy **lock\_guard**



Operacje atomowe

# Jak zbudować licznik?

```
class Counter
{
    int _counter = 0;
public:
    void operator++() { _counter++; }
    void operator--() { _counter--; }
    int get() const { return _counter; }
};
```

- Wiemy, że powyższa klasa w programie wielowątkowym doprowadzi do wyścigu

# Może mutex?

```
std::mutex mu;

class Counter
{
    int _counter = 0;
public:
    void operator++()
    {
        std::lock_guard<std::mutex> lg(mu);
        _counter++;
    }
    void operator--()
    {
        std::lock_guard<std::mutex> lg(mu);
        _counter--;
    }
    int get() const
    {
        std::lock_guard<std::mutex> lg(mu);
        return _counter;
    }
};
```

- Działa
- Ale czy nie za wolno?

# Program testujący...

```
void test(Counter & c, std::vector<int> const& v, int i0, int n)
{
    for (int i = i0; i < i0 + n; i++)
        if (v[i] % 2 == 0)
            ++c;
        else
            --c;
}

int main()
{
    constexpr int N = 50000000;
    constexpr int M = 4;
    Counter c;
    std::vector<int> dane(N);
    std::vector<std::thread> v;
    std::generate(dane.begin(), dane.end(), std::rand);
    for (int i = 0; i < M; i++)
        v.emplace_back(&test, std::ref(c), std::cref(dane), i*N/M, N/M);
    for (auto & th : v)
        th.join();
    std::cout << "przewaga parzystych: " << c.get() << std::endl;
}
```

# Wyniki

- Wersja z wyścigiem:

```
real    0m0.516s
user    0m0.448s
sys     0m0.096s
```

- Wersja z muteksem:

```
real    0m6.915s
user    0m8.412s
sys     0m15.696s
```

- Cały program działa ok. 13 razy wolniej
- Ale fragment wykonywany równolegle nie powinien zająć więcej niż 20% całości, czyli maksymalnie 0.1 s
- `std::mutex` spowolnił więc część równoległą o co najmniej 50-70 razy...

# Wersja „atomowa”

```
#include <atomic>

class Counter
{
    std::atomic<int> _counter;
public:
    Counter()
        : _counter(0)
    {}
    void operator++() { _counter++; }
    void operator--() { _counter--; }
    int get() const { return _counter; }
};
```



real	0m1.873s
user	0m5.724s
sys	0m0.112s

Znacznie szybciej!

# Porównanie

- Wersja 1 (błędna) →

```
real    0m0.516s
user    0m0.448s
sys     0m0.096s
```

- Wersja 2 (mutex) →

```
real    0m6.915s
user    0m8.412s
sys     0m15.696s
```

- Wersja 3 (atomic) →

```
real    0m1.873s
user    0m5.724s
sys     0m0.112s
```

- `std::atomic` jest szybsze niż `std::mutex`
- ale mniej elastyczne
- I... każda synchronizacja kosztuje!



# Operacje atomowe

- Bardzo proste operacje, które raz rozpoczęte, nie mogą zostać przerwane
- `std::atomic<T>`  
T= wszystkie standardowe typy całkowite,  
w tym: `long long int`
- Operacje:  
    `++`, `--`,  
    `=`, `+=`, `-=`, `&=`, `|=`, `^=`,  
    `load`, `store`, `exchange`



Zmienne warunkowe

# Zmienne warunkowe

- Zmienna warunkowa to obiekt, którego można użyć do zablokowania wątku lub wątków, dopóki inny wątek tej zmiennej nie odblokuje
- Blokadę zakłada się funkcją:  
**`wait`, `wait_for` lub `wait_until`**
- Argumentem funkcji blokujących jest zawsze **`std::unique_lock`** (na **`std::mutex`**)
- Blokadę znosi wywołanie na obiekcie metody
  - **`notify_one`** – znosi blokadę jednego wątku
  - **`notify_all`** – znosi blokadę wszystkich wątku

**#include <condition\_variable>**

- **wait**
- **wait\_for**
- **wait\_until**
- **notify\_one**
- **notify\_all**

# Przykład


- Wątek roboczy ma odblokować wątek główny po 5 sekundach pracy

**Wersja atomowa**

```
std::atomic<bool> is_ready(false);

void test()
{
    std::this_thread::sleep_for(std::chrono::seconds(5));
    is_ready.store(true);
}

int main()
{
    std::thread th(test);
    while (!is_ready.load())
        std::this_thread::yield();
    th.join();
}
```



Oddaje sterowanie do systemu

# Wersja ze zmienną warunkową

```
#include <condition_variable>

bool is_ready(false);
std::mutex m;
std::condition_variable cv; } Zmienne globalne

void test()
{
    std::this_thread::sleep_for(std::chrono::seconds(5));
    std::unique_lock<std::mutex> lk(m);
    is_ready = true;
    cv.notify_one();
}

int main()
{
    std::thread t(test);
    std::unique_lock<std::mutex> lk(m);
    while (!is_ready)
    {
        cv.wait(lk);
        if (!is_ready)
            std::cout << "To niemożliwe!\n";
    }
    t.join();
}
```

**Blokada na muteksie m**

## Wersja ze zmienną warunkową

```
#include <condition_variable>

bool is_ready(false);
std::mutex m;
std::condition_variable cv;

void test()
{
    std::this_thread::sleep_for(std::chrono::seconds(5));
    std::unique_lock<std::mutex> lk(m);
    is_ready = true;
    cv.notify_one();
}

int main()
{
    std::thread t(test);
    std::unique_lock<std::mutex> lk(m);
    while (!is_ready)
    {
        cv.wait(lk);
        if (!is_ready)
            std::cout << "To niemożliwe!\n";
    }
    t.join();
}
```



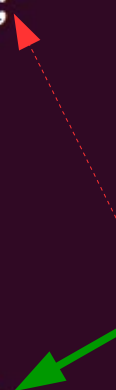
# Wersja ze zmienną warunkową

```
#include <condition_variable>

bool is_ready(false);
std::mutex m;
std::condition_variable cv;

void test()
{
    std::this_thread::sleep_for(std::chrono::seconds(5));
    std::unique_lock<std::mutex> lk(m);
    is_ready = true;
    cv.notify_one();
}

int main()
{
    std::thread t(test);
    std::unique_lock<std::mutex> lk(m);
    while (!is_ready)
    {
        cv.wait(lk);
        if (!is_ready)
            std::cout << "To niemożliwe!\n";
    }
    t.join();
}
```



**1. Blokada na muteksie m**



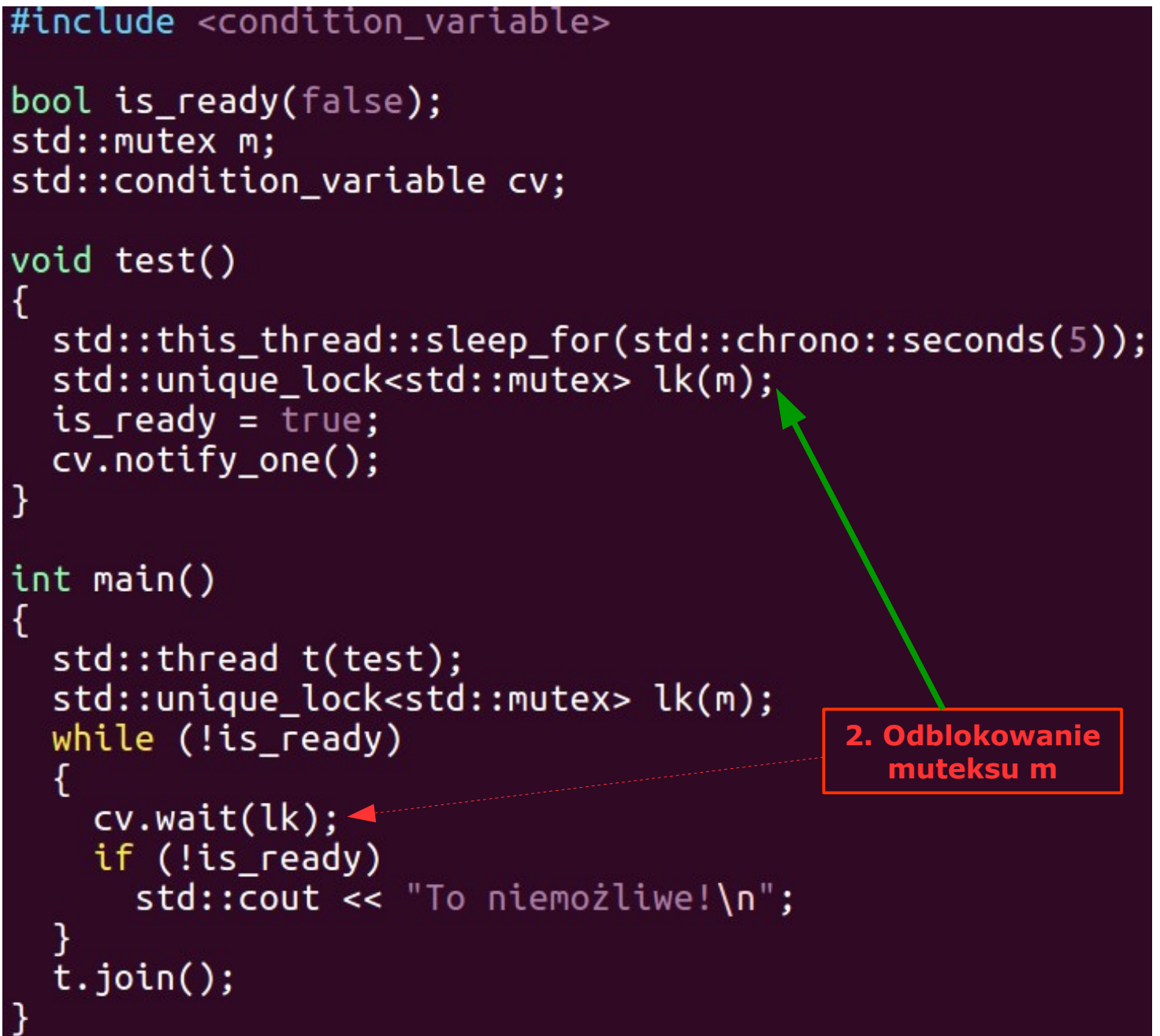
## Wersja ze zmienną warunkową

```
#include <condition_variable>

bool is_ready(false);
std::mutex m;
std::condition_variable cv;

void test()
{
    std::this_thread::sleep_for(std::chrono::seconds(5));
    std::unique_lock<std::mutex> lk(m);
    is_ready = true;
    cv.notify_one();
}

int main()
{
    std::thread t(test);
    std::unique_lock<std::mutex> lk(m);
    while (!is_ready)
    {
        cv.wait(lk);
        if (!is_ready)
            std::cout << "To niemożliwe!\n";
    }
    t.join();
}
```



2. Odblokowanie muteksu m



## Wersja ze zmienną warunkową

```
#include <condition_variable>

bool is_ready(false);
std::mutex m;
std::condition_variable cv;

void test()
{
    std::this_thread::sleep_for(std::chrono::seconds(5));
    std::unique_lock<std::mutex> lk(m);
    is_ready = true;
    cv.notify_one();
}

int main()
{
    std::thread t(test);
    std::unique_lock<std::mutex> lk(m);
    while (!is_ready)
    {
        cv.wait(lk);
        if (!is_ready)
            std::cout << "To niemożliwe!\n";
    }
    t.join();
}
```



3. Zdjęcie blokady z cv

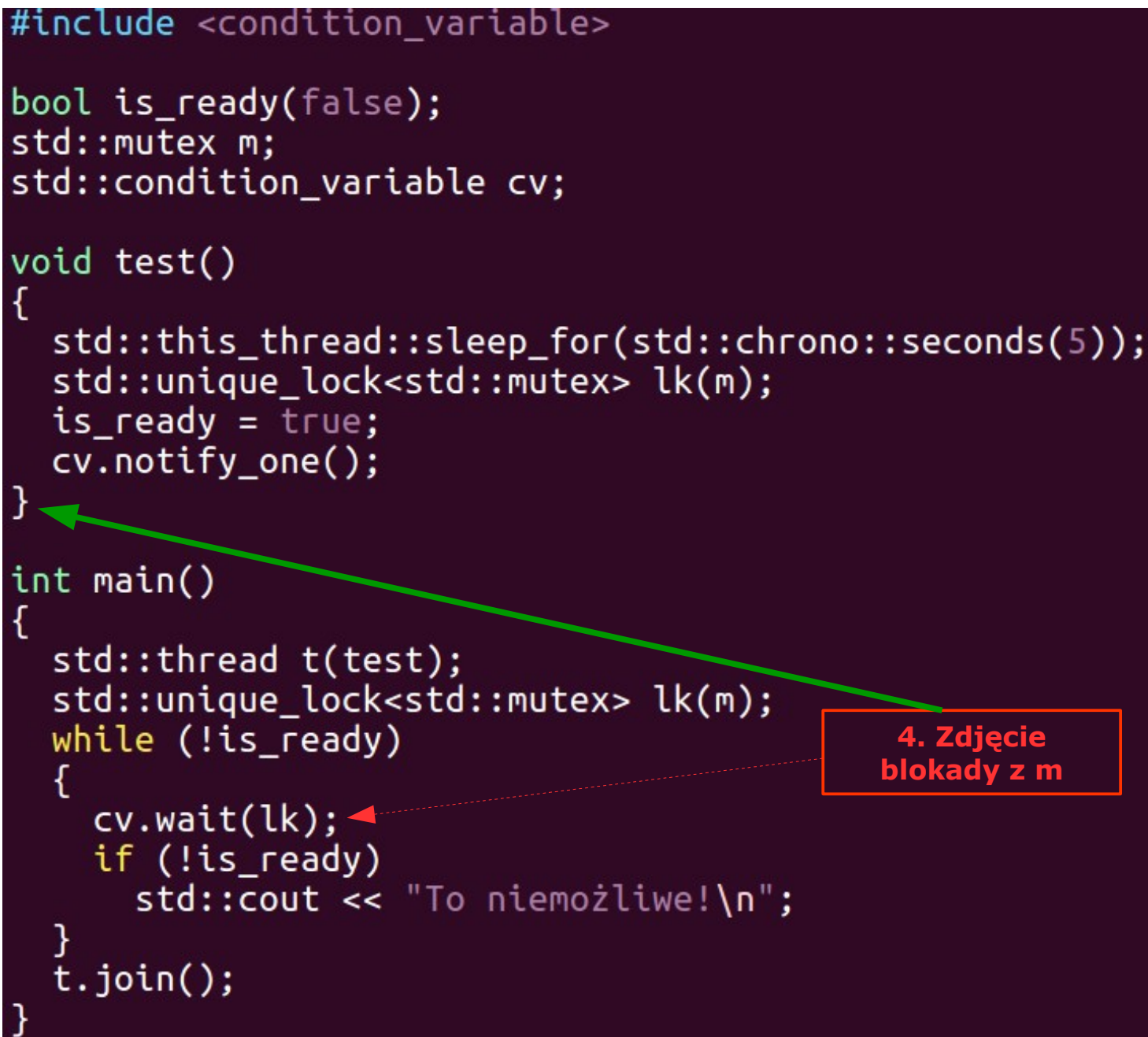
## Wersja ze zmienną warunkową

```
#include <condition_variable>

bool is_ready(false);
std::mutex m;
std::condition_variable cv;

void test()
{
    std::this_thread::sleep_for(std::chrono::seconds(5));
    std::unique_lock<std::mutex> lk(m);
    is_ready = true;
    cv.notify_one();
}

int main()
{
    std::thread t(test);
    std::unique_lock<std::mutex> lk(m);
    while (!is_ready)
    {
        cv.wait(lk);
        if (!is_ready)
            std::cout << "To niemożliwe!\n";
    }
    t.join();
}
```



4. Zdjęcie blokady z m

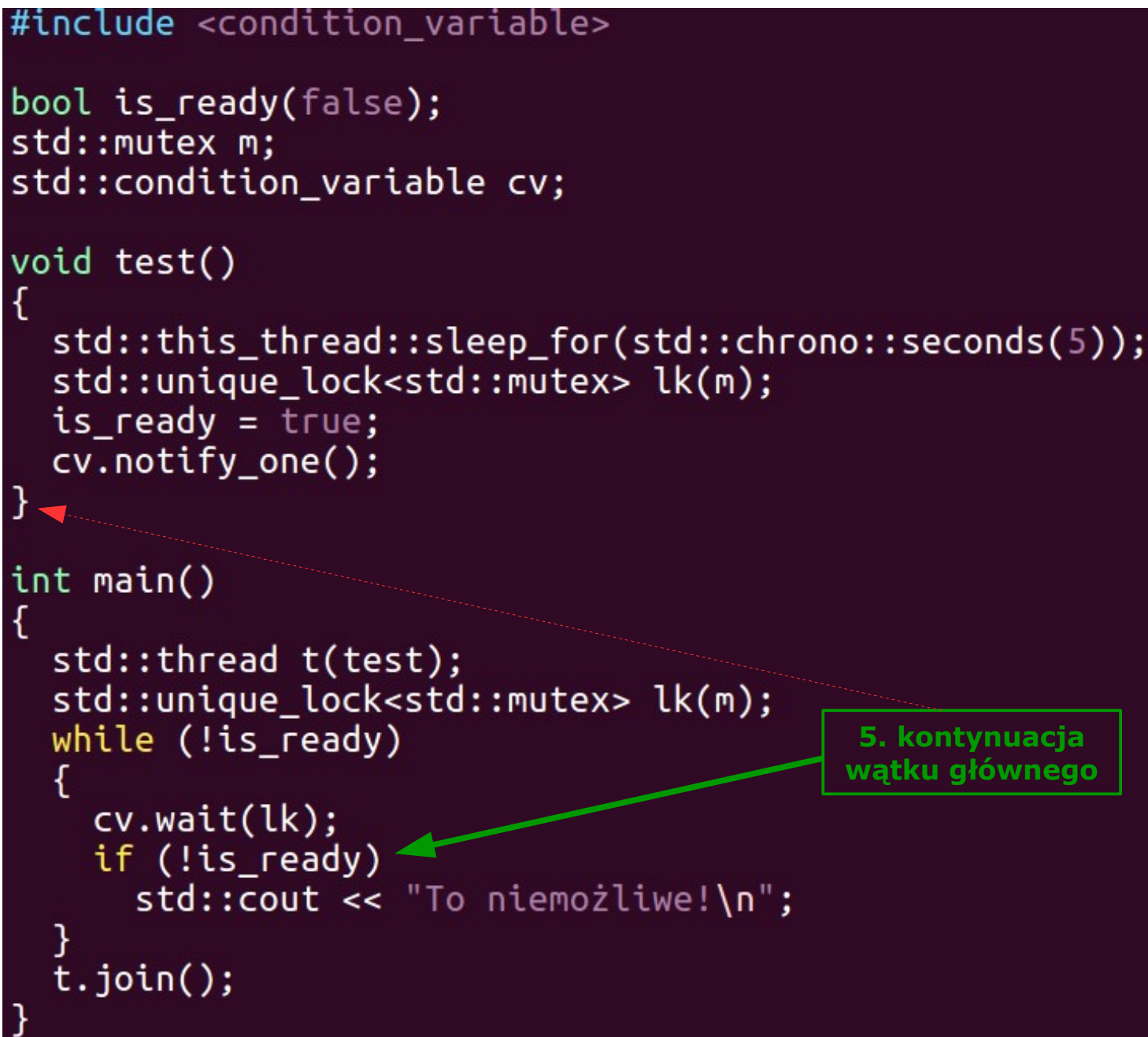
## Wersja ze zmienną warunkową

```
#include <condition_variable>

bool is_ready(false);
std::mutex m;
std::condition_variable cv;

void test()
{
    std::this_thread::sleep_for(std::chrono::seconds(5));
    std::unique_lock<std::mutex> lk(m);
    is_ready = true;
    cv.notify_one();
}

int main()
{
    std::thread t(test);
    std::unique_lock<std::mutex> lk(m);
    while (!is_ready)
    {
        cv.wait(lk);
        if (!is_ready)
            std::cout << "To niemożliwe!\n";
    }
    t.join();
}
```



5. kontynuacja wątku głównego

# Wyniki

- Wersja atomowa

```
real    0m5.006s
user    0m0.604s
sys     0m4.404s
```

- Wersja ze zmienną warunkową

```
real    0m5.007s
user    0m0.004s
sys     0m0.000s
```

- (trochę oszukałem: wystarczy zastąpić yield na sleep\_for(0))...
- Niemniej, zmienne warunkowe są naprawdę szybkie