

Zmienne atomowe

CODERS SCHOOL

<https://coders.school>



Łukasz Ziobroń
lukasz@coders.school

Łukasz Ziobroń & Bartosz Szurgot - autorzy



Łukasz Ziobroń

Not only programming experience:

- C++ and Python developer @ Nokia & Credit Suisse
- Scrum Master @ Nokia & Credit Suisse
- Code Reviewer @ Nokia
- Webmaster (HTML, PHP, CSS) @ StarCraft Area

Training experience:

- C++ trainings @ Coders School
- Practical Aspects Of Software Engineering @ PWr, UWr
- Nokia Academy @ Nokia
- Internal corporate trainings

Public speaking experience:

- Academic Championships in Team Programming
- code::dive conference
- code::dive community



Zmienne atomowe - Agenda

- Model pamięci C++
- Model pamięci C++ - kiedy synchronizować?
- Model pamięci C++ - jak synchronizować?
- Przykład: jak zatrzymać zapętłony wątek?
- Zwykłe zmienne vs zmienne atomowe
- Zmienne atomowe
- Zadanie: synchronizacja danych
- `std::memory_order`
- Sequential consistency i optymalizacje kompilatora
- Zmienna atomowa – podsumowanie

Model pamięci C++

- Najmniejsza jednostka – 1 bajt
- Każdy bajt ma unikalny adres w pamięci
- Synchronizacja nie jest potrzebna jeśli zapisujemy coś wielowątkowo do różnych obszarów pamięci

```
vector<int> v{10};  
thread t1([&]{ v[0] = 5; });  
thread t2([&]{ v[1] = 15; });
```
- Synchronizacja jest potrzebna jeśli zapisujemy coś wielowątkowo do tych samych obszarów pamięci
- Synchronizacja jest potrzebna jeśli co najmniej jeden wątek zapisuje a inne odczytują ten sam obszar pamięci
- Brak synchronizacji gdy jest wymagana == wyścig == niezdefiniowane zachowanie
- `const` implikuje bezpieczeństwo wielowątkowe, bo gwarantuje tylko odczyt
- [C++ Memory model on cppreference.com](http://cplusplusmemorymodel.com)

Model pamięci C++ - kiedy synchronizować?

- Czy tutaj potrzebna jest synchronizacja?

```
struct S {  
    char a;  
    int b;  
} obj;  
thread t1([&]{ obj.a = 'a'; });  
thread t2([&]{ obj.b = 4; });
```

- Od C++11 nie, pomimo tej samej struktury obszary pamięci w których zapisujemy dane są rozłączne
- W starych wątkach POSIX możliwy jest wyścig

Model pamięci C++ - kiedy synchronizować?

- Czy tutaj potrzebna jest synchronizacja?

```
vector<int> v(10, 0);  
for (int i = 0; i < 10; i++)  
    thread t([&]{ v[i] = i; });
```

- Nie, pomimo tej samej struktury obszary pamięci w których zapisujemy dane są rozłączne

Model pamięci C++ - kiedy synchronizować?

- Czy tutaj potrzebna jest synchronizacja?

```
vector<int> v;  
for (int i = 0; i < 10; i++)  
    thread t([&]{ v.emplace_back(i); });
```

- TAK
- Podczas wrzucania nowego obiektu trzeba zinkrementować iterator end() – możliwy wyścig
- Podczas wrzucania nowego obiektu może dojść do realokacji wektora. Niektóre wątki mogą mieć iteratory na nieaktualną pozycję wektora.

Model pamięci C++ - jak synchronizować?

- Jak zsynchronizować zapisy / zapis + odczyt?

- `std::mutex` - to już znacie

```
int a = 0;
mutex m;
thread t1([&]{
    lock_guard<mutex> lg(m);
    a = 1;
});
thread t2([&]{
    lock_guard<mutex> lg(m);
    a = 2;
});
```

- `std::atomic<T>` - to teraz poznamy

```
atomic<int> a = 0;
thread t1([&]{ a = 1; });
thread t2([&]{ a = 2; });
```

Przykład: jak zatrzymać zapętłony wątek?

```
#include <thread>
using namespace std;

int main() {
    bool stop = false;
    auto f = [&] {
        while (not stop) {
            /* do sth... */
        }
    };
    thread t(f);
    stop = true;
    t.join();
    return 0;
}
```

```
$> g++ 01_stop.cpp -lpthread -fsanitize=thread
$> ./a.out
WARNING: ThreadSanitizer: data race (pid=10179)
...
$> g++ 01_stop.cpp -lpthread -fsanitize=thread -O3
$> ./a.out
deadlock
```

- Zapis wykonywany w kilku krokach
- Optymalizacje kompilatora
- Optymalizacje w procesorze (*cache*)
- Są wyścigi
- Możliwe zakleszczenie
- Jak zrobić to lepiej?

Przykład

```
#include <thread>
using namespace std;

int main() {
    bool stop = false;
    auto f = [&stop]() {
        while (!stop) {
            // ...
        }
    };
    thread t(f);
    stop = true;
    t.join();
    return 0;
}
```

```
$> g++ 01_stop.cpp
$> ./a.out
WARNING: Thread 1 is stuck in a loop
...
$> g++ 01_stop.cpp
$> ./a.out
deadlock
```



EPIC FAIL.

You're doing it wrong!

krokach

(*cache*)

Przykład: jak zatrzymać zapętłony wątek – volatile?

```
#include <thread>
using namespace std;

int main() {
    volatile bool stop = false;
    auto f = [&] {
        while (not stop) {
            /* do sth... */
        }
    };
    thread t(f);
    stop = true;
    t.join();
    return 0;
}
```

```
$> g++ 01b_volatile.cpp -lpthread -fsanitize=thread
$> ./a.out
WARNING: ThreadSanitizer: data race (pid=10179)
...
$> g++ 01b_volatile.cpp -lpthread -fsanitize=thread -O3
$> ./a.out
WARNING: ThreadSanitizer: data race (pid=10179)
...
```

- Zapis wykonywany w kilku krokach
- ~~Optymalizacje kompilatora~~
- Optymalizacje w procesorze (*cache*)
- Są wyścigi
- Jest niezdefiniowane zachowanie
- Iluzja poprawności
- volatile == may be modified by external agents
- volatile != may be modified concurrently by the program
- Jak zrobić to lepiej?

Przykład: j

```
#include <thread>
using namespace std;

int main() {
    volatile bool stop = false;
    auto f = [&]() {
        while (true) {
            /* do something */
        }
    };
    thread t(f);
    stop = true;
    t.join();
    return 0;
}
```

```
$> g++ 01b_volatile.cpp
$> ./a.out
WARNING: ThreadSanitizer: data race
...
$> g++ 01b_volatile.cpp
$> ./a.out
WARNING: ThreadSanitizer: data race
...
```



FAIL

ku krokach

fe

rze (*cache*)

chowanie

fied by

ed
ram

Przykład: jak zatrzymać zapętłony wątek – zmienna z mutexem?

```
#include <thread>
#include <mutex>
using namespace std;

int main() {
    bool flag = false;
    mutex m;
    auto stop = [&] {
        lock_guard<mutex> lg(m);
        return flag;
    };
    auto f = [&] {
        while (not stop()) {
            /* do sth... */
        }
    };
    thread t(f);
    {
        lock_guard<mutex> lg(m);
        flag = true;
    }
    t.join();
    return 0;
}
```

- Poprawne!
- Długi kod...
- Wolne...

```
$> g++ 01c_mutex.cpp -lpthread -fsanitize=thread
$> ./a.out
$> g++ 01c_mutex.cpp -lpthread -fsanitize=thread -O3
$> ./a.out
```


Przykład

```
#include <thread>
#include <mutex>
using namespace
```

```
int main() {
    bool flag =
    mutex m;
    auto stop =
    lock_gu
    return

};
auto f = [8
    while
    /*
    }
};
thread t(f)
{
    lock_gu
    flag =

}
t.join();
return 0;
}
```



SPIDER-PIG

Does Whatever A Spider-Pig Does

em?

ze=thread

ze=thread -03

Przykład: jak zatrzymać zapętłony wątek – zmienna atomowa

```
#include <thread>
#include <atomic>
using namespace std;

int main() {
    atomic<bool> stop{false};

    auto f = [&] {
        while (not stop) {
            /* do sth... */
        }
    };
    thread t(f);
    stop = true;
    t.join();
    return 0;
}
```

- Poprawne!
- Lekkie
- Mało kodu

```
$> g++ 01d_atomic.cpp -lpthread -fsanitize=thread -O3
$> ./a.out
```


Zwykłe zmienne vs zmienne atomowe

Zwykłe zmienne

- jednoczesny zapis i odczyt == niezdefiniowane zachowanie
- potrzeba blokowania muteksami w przypadku modyfikacji

Zmienne atomowe

- jednoczesny zapis i odczyt == zdefiniowane zachowanie
- brak dodatkowych mechanizmów blokowania

Zmienne atomowe

- `#include <atomic>`
- `std::atomic`
- Lekka synchronizacja
- Pozwala na prostą arytmetykę i operacje bitowe: `++`, `--`, `+=`, `-=`, `&=`, `|=`, `^=`
- Typowo: liczby, wskaźniki
- Najważniejsze operacje:
 - `store()` - zapisuje wartość w zmiennej atomowej, dodatkowo można podać `std::memory_order`
 - `operator=()` - zapisuje wartość w zmiennej atomowej
 - `load()` - odczytuje wartość ze zmiennej atomowej, dodatkowo można podać `std::memory_order`
 - `operator T()` - odczytuje wartość ze zmiennej atomowej

Zadanie: synchronizacja danych

```
vector<int> generateContainer() {
    vector<int> input =
        {2, 4, 6, 8, 10, 1, 3, 5, 7, 9};
    vector<int> output;
    vector<thread> threads;
    for (auto i = 0u; i < input.size(); i++) {
        threads.emplace_back([&]{
            output.push_back(input[i]);
        });
    }
    for (auto && t : threads) {
        t.join();
    }
    return output;
}

vector<int> generateOtherContainer() {
    int start = 5;
    bool add = true;
    vector<int> output;
    vector<thread> threads;
    for (int i = 0; i < 10; i++) {
        threads.emplace_back([&]{
            if (add)
                output.push_back(start+=i);
            else
                output.push_back(start-=i);
            add = !add;
        });
    }
    for (auto && t : threads) {
        t.join();
    }
    return output;
}

void powerContainer(vector<int>& input) {
    vector<thread> threads;
    for (auto i = 0u; i < input.size(); i++) {
        threads.emplace_back([&]{
            input[i]*=input[i];
        });
    }
    for (auto && t : threads) {
        t.join();
    }
}
```

- Użyj właściwych mechanizmów synchronizacji

```
$> g++ 01_synchronization.cpp -Wall -Wextra -Werror -pedantic -lpthread -fsanitize=thread -O3
$> ./a.out
```

Rozwiązanie - synchronizacja danych

- operacje wstawiania na vector powinny być synchronizowane za pomocą mutexu
- wstawianie wątków do vectora threads jest wykonywane sekwencyjnie w pętli – nie trzeba tego synchronizować
- gdy każdy wątek pisze do innego elementu wektora synchronizacja nie jest potrzebna
- proste typy powinny być opakowane w typ `atomic`
 - `bool add`
 - `int start`
- funkcje lambda powinny być wyciągnięte przed pętlę

```
$> g++ 01_synchronization.cpp -Wall -Wextra -Werror -pedantic -lpthread -fsanitize=thread -O3
$> ./a.out
2 4 6 8 10 1 3 5 7 9
4 16 36 64 100 1 9 25 49 81
1 0 2 -1 3 -2 4 -3 5 -4
1 0 4 1 9 4 16 9 25 16
```

std::memory_order

- W ramach optymalizacji kompilator ma prawo zmienić kolejność wykonywanych operacji
- Kompilator musi wiedzieć, które operacje może przestawiać, a które muszą następować w określonej kolejności
- SC - Sequential consistency (`memory_order_seq_cst`) gwarantuje zachowaną kolejność operacji ustaloną przez programistę, czasami kosztem wydajności. Jest to domyślne zachowanie zmiennych `std::atomic`
- Dzięki SC możemy poprawnie wnioskować jakie wartości mogą mieć zmienne niezależnie od optymalizacji procesora
- Optymalizacje kompilatora nie mogą tworzyć wyścigów
- Nudne szczegóły: memory_order_on_cppreference.com

Sequential consistency i optymalizacje kompilatora

```
// INPUT:  
int foo(int a)  
{  
    if(a<1)  
        b=2;  
    if(a==2)  
        b=2;  
    if(a>2)  
        b=2;  
    return b;  
}
```

```
// OPT1:  
int foo(int a)  
{  
    if(a>2)  
        b=2;  
    else  
        if(a<1)  
            b=2;  
        else  
            if(a==2)  
                b=2;  
    return b;  
}
```

```
// OPT2:  
int foo(int a)  
{  
    const int tmp=b;  
    b=2;  
    if(a==1)  
        b=tmp;  
    return b;  
}
```

- Poprawne?
- Tylko OPT1
- W OPT2 stan b został zmieniony niezależnie od wartości a
- Inny wątek mógł odczytać wartość b w tym czasie

Zmienna atomowa - podsumowanie

- `std::atomic` to lekka synchronizacja
- Pozwala na prostą arytmetykę i operacje bitowe: `++`, `--`, `+=`, `-=`, `&=`, `|=`, `^=`
- Typowo: liczby, wskaźniki
- Używa specjalnych instrukcji procesora do niepodzielnej modyfikacji danych
- `std::atomic` na typach złożonych nie ma sensu
 - nie ma specjalnych instrukcji procesora, które zapewniają niepodzielność takich operacji
 - nie ma modelu pamięci transakcyjnej w C++ (jeszcze)
 - jeśli się uda, to może nie działać zgodnie z założeniami (patrz. [Stack Overflow](#))
 - należy użyć mutexów

Przydatne linki

- [C++ Atomic Types and Operations \(C++ Standard\)](#)
- [C++ Memory model on cppreference.com](#)
- [std::memory_order on cppreference.com](#)

CODERS SCHOOL

<https://coders.school>

ASK A NINJA



Łukasz Ziobroń
lukasz@coders.school