Jednokrotne wywołania

CODERS SCHOOL

https://coders.school

Łukasz Ziobroń
lukasz@coders.school

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
- Practial 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

Jednokrotne wywołania - Agenda

- call_once
- once_flag
- Zasada działania call_once
- Zadanie 1: gonitwa
- Zadanie 2: wykluczające się wywołania
- Przykład: thread-safe Singleton
- Zadanie 3: wyjątki w call_once
- Wyjątki w call_once – bug w implementacji biblioteki standardowej

call_once

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

std::once_flag flag;

void do_once() {
    std::call_once(flag, [] {
        std::cout << "Called once!\n";
    });
}

int main() {
    std::thread t1(do_once);
    std::thread t2(do_once);
    std::thread t3(do_once);
    t1.join();
    t2.join();
    t3.join();
    return 0;
}
```

- `#include <mutex>`
- `std::call_once`
- Opakowuje funkcję, która zostanie wykonana tylko raz
- Gwarantuje jednokrotne wywołanie nawet w przypadku jej współbieżnego wywołania przez kilka wątków
- Wywołuje przekazaną funkcję w swoim wątku (nie tworzy nowego)
- Potrzebuje flagi `std::once_flag`

```
$> g++ 01_call_once.cpp -lpthread -fsanitize=thread
$> ./a.out
Called once!
```

once_flag

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

std::once_flag flag;

void do_once() {
    std::call_once(flag, [] {
        std::cout << "Called once!\n";
    });
}

int main() {
    std::thread t1(do_once);
    std::thread t2(do_once);
    std::thread t3(do_once);
    t1.join();
    t2.join();
    t3.join();
    return 0;
}
```

- `#include <mutex>`
- `std::once_flag`
- Pomocnicza struktura do użytku z `std::call_once`
- Brak kopiowania i przenoszenia
- Zawiera informację, czy funkcja z jej użyciem została już wywołana
- Konstruktor ustawia stan na niewywołany

```
$> g++ 01_call_once.cpp -lpthread -fsanitize=thread
$> ./a.out
Called once!
```

Zasada działania call_once

- Jeśli once_flag jest w stanie "wywołana", `call_once` natychmiast zwraca - return (passive call)
- Jeśli `once_flag` jest w stanie "nie wywołana", `call_once` wykonuje przekazaną funkcję, przekazując do niej dalsze argumenty (active call)
  - Jeśli funkcja rzuci wyjątkiem to jest on propagowany dalej, a `once_flag` nie zostaje ustawiona w stanie "wywołana" (exceptional call), więc inny `call_once` może zostać wywołany (przynajmniej w teorii ☺)
  - Jeśli funkcja zakończy się normalnie, `once_flag` zostaje ustawiona w stan "wywołana" (returning call). Gwarantowane jest, że wszystkie inne wywołania będą pasywne.
- Kilka aktywnych zawołań na tej samej fladze `once_flag` jest kolejkowanych.
- Jeśli tej samej flagi używamy do współbieżnych wywołań różnych funkcji, to nie jest wyspecyfikowane, która funkcja zostanie zawołana.

Zadanie 1: gonitwa

```
void setWinner() {
    auto id = this_thread::get_id();
    auto sleepDuration = dist(rng);
    stringstream msg;
    msg << "Called " << __FUNCTION__
        << "(" << id << "). Chasing time: "
        << sleepDuration << "ms\n";
    cout << msg.str();

    // TODO: set me as a winner
    // but don't let others overwrite this!
}
```

- 10 zawodników (wątków) ściga się o w zawodach o milion $
- Tylko pierwszy zawodnik zdobywa nagrodę, reszta nie dostanie nic
- Zaimplementuj funkcję `setWinner()` tak, aby zwycięski wątek ustawił siebie jako zwycięzcę i nie pozwolił na nadpisanie innym tej wartości

```
$> g++ 01_race.cpp -lpthread -fsanitize=thread
$> ./a.out
Called setWinner(139887531521792). Sleeping for 15ms
Called setWinner(139887523129088). Sleeping for 35ms
Called setWinner(139887497950976). Sleeping for 31ms
Call once for 139887531521792
Called setWinner(139887489558272). Sleeping for 16ms
Called setWinner(139887481165568). Sleeping for 14ms
Called setWinner(139887453927168). Sleeping for 35ms
And the winner is... 139887531521792
```

Zadanie 1 - rozwiązanie

```
void setWinner() {
    auto id = this_thread::get_id();
    auto sleepDuration = dist(rng);
    stringstream msg;
    msg << "Called " << __FUNCTION__
        << "(" << id << "). Chasing time: "
        << sleepDuration << "ms\n";
    cout << msg.str();
    this_thread::sleep_for(
        chrono::milliseconds(sleepDuration)
    );

    call_once(once, [&]{
        cout << "Call once for " << id << '\n';
        stringstream troublesomeConversion;
        troublesomeConversion << id;
        winnerId = troublesomeConversion.str();
    });
}
```

- 10 zawodników (wątków) ściga się o w zawodach o milion $
- Tylko pierwszy zawodnik zdobywa nagrodę, reszta nie dostanie nic
- Zaimplementuj funkcję `setWinner()` tak, aby zwycięski wątek ustawił siebie jako zwycięzcę i nie pozwolił na nadpisanie innym tej wartości

```
$> g++ 01_race.cpp -lpthread -fsanitize=thread
$> ./a.out
Called setWinner(139887531521792). Sleeping for 15ms
Called setWinner(139887523129088). Sleeping for 35ms
Called setWinner(139887497950976). Sleeping for 31ms
Call once for 139887531521792
Called setWinner(139887489558272). Sleeping for 16ms
Called setWinner(139887481165568). Sleeping for 14ms
Called setWinner(139887453927168). Sleeping for 35ms
And the winner is... 139887531521792
```

Zadanie 2: wykluczające się wywołania

```
class X {
    vector<double> values;

    void initializeOne()   { values = {1.0}; }
    void initializeTwo()   { values = {1.0, 2.0}; }
    void initializeThree() { values = {1.0, 2.0, 3.0}; }

public:
    explicit X(int i) noexcept {
        switch (i) {
        case 2: // top priority
            initializeTwo();
            [[fallthrough]];
        case 3:
            initializeThree();
            [[fallthrough]];
        default: // least priority
            initializeOne();
        }
    }
    // ...
};
```

- Dopisz odpowiednie jednokrotne wywołania oraz komunikaty, aby na wyjściu pojawiło się to co poniżej
- Nie modyfikuj konstruktora ;)

```
$> g++ 02_exclusive_calls.cpp -lpthread -fsanitize=thread
$> ./a.out
initializeTwo
Call once initializeTwo
initializeThree
initializeOne
1 2

initializeThree
Call once initializeThree
initializeOne
1 2 3

initializeOne
Call once initializeOne
1
```

Zadanie 2 - rozwiązanie

```
class X {
    once_flag once;
    vector<double> values;

    void initializeOne() {
        cout << __FUNCTION__ << '\n';
        call_once(once, [&]{
            cout << "Call once initializeOne\n";
            values = {1.0};
        });
    }

    void initializeTwo() {
        cout << __FUNCTION__ << '\n';
        call_once(once, [&]{
            cout << "Call once initializeTwo\n";
            values = {1.0, 2.0};
        });
    }

    void initializeThree() {
        cout << __FUNCTION__ << '\n';
        call_once(once, [&]{
            cout << "Call once initializeThree\n";
            values = {1.0, 2.0, 3.0};
        });
    }
    // ...
};
```

- Dopisz odpowiednie jednokrotne wywołania oraz komunikaty, aby na wyjściu pojawiło się to co poniżej
- Nie modyfikuj konstruktora ;)

```
$> g++ 02_exclusive_calls.cpp -lpthread -fsanitize=thread
$> ./a.out
initializeTwo
Call once initializeTwo
initializeThree
initializeOne
1 2

initializeThree
Call once initializeThree
initializeOne
1 2 3

initializeOne
Call once initializeOne
1
```

Przykład: thread-safe Singleton

```
class Singleton {
    static std::unique_ptr<Singleton> instance_;
    Singleton() = default;
public:
    static Singleton& getInstance() {
        std::mutex mutex_;
        std::unique_lock<std::mutex> lock(mutex_);
        if (!instance_) {
            instance_.reset(new Singleton{});
        }
        lock.unlock();
        return *instance_;
    }
};
```

- Wolne (mutexy)
- Bezpieczne

```
class Singleton {
    static std::unique_ptr<Singleton> instance_;
    static std::once_flag flag_;
    Singleton() = default;
public:
    static Singleton& getInstance() {
        std::call_once(flag_, [&] {
            instance_.reset(new Singleton{});
        });
        return *instance_;
    }
};
```

- Wolne (once_flag)
- Bezpieczne
- Trochę mniej kodu
- Czy call_once jest potrzebne?

Przykład: thread-safe Singleton

```
class Singleton {
    static std::unique_ptr<Singleton> instance_;
    Singleton() = default;

public:
    static Singleton& getInstance() {
        if (!instance_) {
            instance_.reset(new Singleton{});
        }
        return *instance_;
    }
};
```

```
class Singleton {
    static std::unique_ptr<Singleton> instance_;
    static std::once_flag flag_;
    Singleton() = default;
public:
    static Singleton& getInstance() {
        std::call_once(flag_, [&] {
            instance_.reset(new Singleton{});
        });
        return *instance_;
    }
};
```

- Szybkie (porównanie)
- Bezpieczne – statyczna inicjalizacja jest thread-safe od C++11

- Wolne (once_flag)
- Bezpieczne
- ~~Trochę mniej kodu~~
- call_once jest nadmiarowe

Przykład: thread-safe Singleton

```
class Singleton {
    static std::unique_ptr<Singleton> instance_;
    Singleton() = default;

public:
    static Singleton& getInstance() {
        if (!instance_) {
            instance_.reset(new Singleton{});
        }
        return *instance_;
    }
};
```

```
class Singleton {
    Singleton() = default;

public:
    static Singleton& getInstance() {
        static Singleton instance_;
        return instance_;
    }
};

// Meyers Singleton
```

- Szybkie (porównanie)
- Bezpieczne – statyczna inicjalizacja jest thread-safe od C++11

- Najszybsze
- Bezpieczne
- Krótkie
- Śliczne

Zadanie 3: wyjątki w call_once

```
class X {
    once_flag once;
    vector<double> values;

    void initializeOne() {
        cout << __FUNCTION__ << '\n';
        call_once(once, [&]{
            cout << "Call once initializeOne\n";
            values = {1.0};
        });
    }

    // ...

    void initializePierdyliard() {
        cout << __FUNCTION__ << '\n';
        call_once(once, [&]{
            cout << "Call once initializePierdyliard\n";
            throw std::bad_alloc{};
            // TODO: Can you fix me?
        });
    }
};
```

- Spróbuj naprawić problem z rzucaniem wyjątków w `call_once`
- Wg cppreference.com:

  If that invocation throws an exception, it is propagated to the caller of call_once, and the flag is not flipped so that another call will be attempted (such call to call_once is known as exceptional).

```
$> g++ 03_exceptional_exclusive_calls.cpp -lpthread -
fsanitize=thread
$> ./a.out
...

initializePierdyliard
Call once initializePierdyliard
terminate called after throwing an instance of
'std::bad_alloc'
  what():  std::bad_alloc
Aborted (core dumped)
```

Zadanie 3 - rozwiązanie

```
class X {
    once_flag once;
    vector<double> values;

    void initializeOne() {
        cout << __FUNCTION__ << '\n';
        call_once(once, [&]{
            cout << "Call once initializeOne\n";
            values = {1.0};
        });
    }

    // ...

    void initializePierdyliard() try {
        cout << __FUNCTION__ << '\n';
        call_once(once, [&]{
            cout << "Call once initializePierdyliard\n";
            throw std::bad_alloc{};
        });
    } catch (...) { /* ignore exceptions */ }
};
```

- Spróbuj naprawić problem z rzucaniem wyjątków w `call_once`
- Wg cppreference.com:

  If that invocation throws an exception, it is propagated to the caller of call_once, and the flag is not flipped so that another call will be attempted (such call to call_once is known as exceptional).

NIE DA SIĘ! (Przynajmniej u mnie ☺)

```
$> g++ 03_exceptional_exclusive_calls.cpp -lpthread -
fsanitize=thread
$> ./a.out
...

initializePierdyliard
Call once initializePierdyliard
initializeOne

(hang up)
```

Wyjątki w call_once – bug w implementacji biblioteki standardowej

- Jeśli `once_flag` jest w stanie "wywołana", `call_once` natychmiast zwraca - return (passive call)
- Jeśli `once_flag` jest w stanie "nie wywołana", `call_once` wykonuje przekazaną funkcję, przekazując do niej dalsze argumenty (active call)
  - Jeśli funkcja rzuci wyjątkiem to jest on propagowany dalej, a `once_flag` nie zostaje ustawiona w stanie "wywołana" (exceptional call), więc inny `call_once` może zostać wywołany (przynajmniej w teorii ☺) – bug w implementacji, przykład na cppreference.com też nie działa
  - Jeśli funkcja zakończy się normalnie, `once_flag` zostaje ustawiona w stan "wywołana" (returning call). Gwarantowane jest, że wszystkie inne wywołania będą pasywne.
- Kilka aktywnych zawołań na tej samej fladze `once_flag` jest kolejkowanych.
- Jeśli tej samej flagi używamy do współbieżnych wywołań różnych funkcji, to nie jest wyspecyfikowane, która funkcja zostanie zawołana.

Przydatne linki

- std::call_once on cppreference.com
- std::once_flag on cppreference.com
- STL bug in exception handling in call_once
- call_once vs mutex on stackoverflow
- Meyers Singleton on stackoverflow

CODERS SCHOOL

https://coders.school

Łukasz Ziobroń
lukasz@coders.school