Zadania asynchroniczne

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

Zadania asynchroniczne - Agenda

- Zadanie domowe dla chętnych: generalizacja funkcji schedule
- std::async
- Przykład: polityki uruchamiania
- Polityki uruchamiania (launch policies)
- Zadanie 1: problem braku polityki
- Brak polityki = Niezdefiniowane zachowanie
- Zagadka
- std::packaged_task

Zadanie domowe dla chętnych: generalizacja funkcji schedule (level hard)

```
std::future<int> schedule(std::function<int()> func)
{
    std::promise<int> p;
    std::future<int> f = p.get_future();
    auto wrapped_func = [func] (std::promise<int> p) {
        try {
            p.set_value(func());
        } catch(...) {
            p.set_exception(std::current_exception());
        }
    };
    std::thread t(wrapped_func, std::move(p));
    t.detach();
    return f;
}
```

- Zadanie 1. Zmień funkcję `schedule()` tak, aby mogła przyjąć funkcję każdego typu, a więc zachowywała się podobnie do `std::async()`
- Zadanie 2. Dodaj dodatkowy parametr – `std::launch policy`, który określi czy funkcja ma zostać wykonana od razu, czy dopiero na żądanie
- Wzoruj się na : https://en.cppreference.com/w/cpp/thread/async

std::async

```
void promise_future_approach() {
    std::promise<int> promise;
    std::future<int> future = promise.get_future();
    auto function = [] (std::promise<int> promise)
    {
        // ...
        promise.set_value(10);
    };
    std::thread t(function, std::move(promise));
    std::cout << future.get() << std::endl;
    t.join();
}
```

```
void async_approach() {
    auto function = [] ()
    {
        // ...
        return 20;
    };
    std::future<int> future = std::async(function);
    std::cout << future.get() << std::endl;
}
```

- `#include <future>`
- `std::async()`
- Opakowuje funkcję, która może zostać wywołana asynchronicznie
- Zwraca odpowiednie `std::future<T>`
- Obsługuje wyjątki poprzez std::promise/std::future
- Automatycznie tworzy wątki jeśli potrzeba
- Można wybrać rodzaj zachowania poprzez parametr `policy` (`async, deffered`)

```
$> ./02_async
10
20
```

std::async

- std::async to wysoko-poziomowe rozwiązanie (w końcu!), które automatycznie zarządza wywołaniami asynchronicznymi z podstawowymi mechanizmami synchronizacji
- Najwygodniejsza forma odpalania zadań:
  - obsługa wartości zwracanych
  - obsługa wyjątków
  - synchronizacja (blokujące `get()` i `wait()` na `std::future`)
  - scheduler - automatyczne kolejkowanie zadań realizowane poprzez implementację biblioteki standardowej
  - możliwość manualnego wybrania rodzaju odpalenia (natychmiastowe, asynchroniczne - `async`, opóźnione, synchroniczne - `deferred`)

Przykład: polityki uruchamiania

```cpp
#include <future>
#include <vector>
#include <iostream>
#include <chrono>
using namespace std;

int main()
{
    auto f1 = async([] {
        cout << "f1 started\n";
        this_thread::sleep_for(1s);
        return 42;
    });
    cout << "f1 spawned\n";

    auto f2 = async(
    launch::async, []{
        cout << "f2 started\n";
        this_thread::sleep_for(1s);
        return 2 * 42;
    });
    cout << "f2 spawned\n";

    auto f3 = async(
    launch::deferred, []{
        cout << "f3 started\n";
        this_thread::sleep_for(1s);
        return 3 * 42;
    });
    cout << "f3 spawned\n";

    cout << "Getting f1 result\n";
    auto v1 = f1.get();
    cout << "Got f1 result\n";

    cout << "Getting f2 result\n";
    auto v2 = f2.get();
    cout << "Got f2 result\n";

    cout << "Getting f3 result\n";
    auto v3 = f3.get();
    cout << "Got f3 result\n";

    vector<int> numbers = {
        v1, v2, v3
    };
    for (const auto & item : numbers)
        cout << item << '\n';

    return 0;
}
```

- Uruchom examples/04_async_policies
- Spójrz na kod źródłowy
- Uruchom examples/05_async_ids
- Poeksperymentuj z ustawieniami polityk
- Zaobserwuj działania programów
- Wyciągnij wnioski ☺

```
$> ./04_async_policies
f1 spawned
f1 started
f2 spawned
f3 spawned
Getting f1 result
f2 started
Got f1 result
Getting f2 result
Got f2 result
Getting f3 result
f3 started
Got f3 result
42
84
126
```

Polityki uruchamiania (launch policies)

`async(std::launch policy, Function&& f, Args&&... args);`

- `std::launch::async` - wywołanie asynchroniczne, w osobnym wątku systemowym
- `std::lauch::deferred` - leniwie wykonuje funkcję f momencie pierwszego wywołania na obiekcie `future` metod `get()` lub `wait()`. Wykonanie jest synchroniczne, czyli wywołujący czeka na zakończenie funkcji `f`. Jeśli `get()` lub `wait()` nie zostaną zawołane funkcja `f` nie wykona się.

`async(Function&& f, Args&&... args);`

- Brak polityki - zachowuje się tak samo jak `async(std::launch::async | std::launch::deferred, f, args...)`. Implikacje:
  - nie wiadomo, czy `f` zostanie wykonane współbieżnie
  - nie wiadomo, czy `f` wykona się w innym czy tym samym wątku, który wywołuje `get()` lub `wait()` na `future`
  - można nie przewidzieć, czy `f` w ogóle się wykona, bo mogą istnieć ścieżki w kodzie, gdzie `get()` lub `wait()` nie zostanie zawołane (np. z powodu wyjątków)

Zadanie 1: problem braku polityki

```cpp
#include <iostream>
#include <future>
using namespace std;

void f() {
    this_thread::sleep_for(1s);
}

int main() {
    auto fut = async(f);

    while (fut.wait_for(100ms) !=
           future_status::ready) {
        // loop until f has finished running...
        // which may never happen!
        cout << "Waiting...\n";
    }
    cout << "Finally...\n";
}
```

- Undefined Behavior?
- Jeśli scheduler wybierze `std::launch::async` to wszystko jest w porządku
- Jeśli wybierze `std::launch::deferred` to `future_status` nigdy nie będzie miał wartości `ready` i mamy nieskończoną pętlę
- Wybrana polityka może zależeć od obecnego obciążenia systemu
- Napraw ten kod, aby program zawsze się zakończył. Zrób to bez specyfikowania polityki.

```
$> ./04_async_never_called
Waiting...
Waiting...
Waiting...
Waiting...
Waiting...
Waiting...
Waiting...
Waiting...
Waiting...
Waiting...
Finally...
```

Zadanie 1 - rozwiązanie

```cpp
#include <iostream>
#include <future>
using namespace std;

void f() {
    this_thread::sleep_for(1s);
}

int main() {
    auto fut = async(f);

    if (fut.wait_for(0s) == future_status::deferred) {
        cout << "Scheduled as deffered. "
             << "Calling wait() to enforce execution\n";
        fut.wait();
    } else {
        while (fut.wait_for(100ms) !=
               future_status::ready) {
            cout << "Waiting...\n";
        }
        cout << "Finally...\n";
    }
}
```

- Nie ma bezpośredniego sposobu sprawdzenia na `future` w jaki sposób zostanie/zostało uruchomione, ale...
- `wait_for()` zwraca 1 z 3 statusów:
  - `future_status::deferred`
  - `future_status::ready`
  - `future_status::timeout`
- Nie chcemy czekać na `wait_for()`, więc jeśli dla czasu 0 zwraca `deferred`, a nie `timeout` to uruchomienie jest odroczone i zaczeka na wywołanie `get()` lub `wait()`
- https://en.cppreference.com/w/cpp/thread/future/wait_for

```
$> ./04_async_never_called
Waiting...
Waiting...
Waiting...
Waiting...
Waiting...
Waiting...
Waiting...
Waiting...
Waiting...
Finally...
```

Brak polityki = Niezdefiniowane zachowanie

- If both the `std::launch::async` and `std::launch::deferred` flags are set in policy, it is up to the implementation whether to perform asynchronous execution or lazy evaluation.
- (Since C++14) If neither `std::launch::async` nor `std::launch::deferred`, nor any implementation-defined policy flag is set in policy, the behavior is undefined.
- Source: https://en.cppreference.com/w/cpp/thread/async

- Zawsze używaj `std::async()` ze ściśle zdefiniowaną polityką:
  - `std::launch::async`
  - `std::launch::deferred`
  - `std::launch::async | std::launch::deferred`

Zagadka

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

int main() {
    std::string x = "x";

    std::async(std::launch::async, [&x]() {
        x = "y";
    });
    std::async(std::launch::async, [&x]() {
        x = "z";
    });

    std::cout << x;
}
```

- Co wyświetli się na ekranie?
  - x
  - y
  - z
  - To zależy (od czego?)

Zagadka (i odpowiedź)

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

int main() {
    std::string x = "x";

    std::async(std::launch::async, [&x]() {
        x = "y";
    });
    std::async(std::launch::async, [&x]() {
        x = "z";
    });

    std::cout << x;
}
```

```
$> ./06_riddle
z
```

- Co wyświetli się na ekranie?
  - x
  - y
  - z
  - To zależy (od czego?)
- Odpowiedź:
  - z
- Wyjaśnienie:
  - jeśli obiekt future jest tymczasowy, to w destruktorze czeka dopóki zadanie się nie skończy. Drugie zadanie zostanie więc uruchomione po pierwszym i pomimo, że będą one w innych wątkach to ich wykonanie będzie zsynchronizowane
  - https://en.cppreference.com/w/cpp/thread/future/~future
  - http://cppquiz.org/quiz/question/48
  - std::futures from std::async aren't special! – Scott Meyers
- Wnioski:
  - Jeśli chcesz mieć wywołania asynchroniczne, to wynik działania std::async musisz zapisać w zmiennej std::future

std::packaged_task

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

auto globalLambda = [](int a, int b) {
    std::this_thread::sleep_for(std::chrono::seconds(1));
    return std::pow(a, b);
};

void localPackagedTask() {
    std::packaged_task<int(int,int)> task(globalLambda);
    auto result = task.get_future();
    task(2, 9);
    std::cout << "getting result:\t" << result.get() << '\n';
}

void remotePackagedTask() {
    std::packaged_task<int(int,int)> task(globalLambda);
    auto result = task.get_future();
    std::thread t(std::move(task), 2, 9);
    t.detach();
    std::cout << "getting result:\t" << result.get() << '\n';
}

void remoteAsync() {
    auto result = std::async(std::launch::async,
                             globalLambda, 2, 9);
    std::cout << "getting result:\t" << result.get() << '\n';
}
```

- `#include <future>`
- `std::packaged_task`
- Obiekt pomocniczy, przy pomocy którego zaimplementowana jest funkcja `std::async()`
- Opakowuje funkcję, która może zostać wywołana asynchronicznie
- Zwraca odpowiednie `std::future<T>`
- Obsługuje wyjątki poprzez std::promise/std::future
- Nie uruchamia się automatycznie
- Wymaga jawnego wywołania
- Wywołanie można przekazać do innego wątku

```
$> ./07_packaged_task
```

Przydatne linki

- std::async on cppreference.com
- std::packaged_task on cppreference.com
- std::futures from std::async aren't special! – Scott Meyers
- The difference between std::async and std::packaged_task

CODERS SCHOOL

https://coders.school

Łukasz Ziobroń
lukasz@coders.school