

# Asynchroniczne wywołanie z `std::async`

dr inż. Ireneusz Szczęśniak

jesień 2017 roku

## Wsparcie równoległości w C++11

- Obsługa zadań, bez wtajemniczania użytkownika w szczegóły. `std::async` i `std::future`.
- Surowy dostęp do wątków. Użytkownik ma możliwość tworzenia, obsługi i niszczenia wątków w dowolny sposób. `std::thread`, `thread_local`.
- Obsługa zadań z poziomu wątków: `std::packaged_task`, `std::promise`, `shared_future`.
- Niskopoziomowe mechanizmy synchronizacji między wątkami: zmienna warunkowa (`std::condition_variable`) i mutex (`std::mutex`).
- Bardzo dobre wsparcie atomowych operacji i typu atomowego: `std::atomic<T>`.
- **Well begun is half done**. C++11 daje solidny fundament, na którym będzie budowane dalsze wsparcie wielowątkowości biblioteki standardowej C++, np. pule wątków.

## Asynchroniczne wykonanie zadania

- Asynchroniczne wykonanie pozwala na podzielenie obliczeń na zadania do równoległego wykonania.
- Zadanie posiada argumenty i **jedną** wartość obliczeń - tak jak funkcja.
- W C++11 możemy asynchronicznie wykonać funkcję, funkcję lambda, albo obiekt funkcyjny (inaczej nazywany funktorem).

## std::async

- `#include <future>`
- Funkcją `std::async` zlecamy asynchroniczne wykonanie zadania.
- Funkcja ma dwie postacie:
  - `future<T>`  
`async(polityka, funkcja, argumenty)`
  - `future<T>`  
`async(funkcja, argumenty)`
- Funkcja zwraca obiekt klasy `std::future<T>`, gdzie typ wyniku T jest taki sam, jak wartość zwracana przez funkcję do asynchronicznego wywołania.
- Polityka wykonania zadania może być:  
`std::launch::deferred` albo `std::launch::async`.

## Polityki wykonania

Obecnie są dostępne dwie polityki wykonania:

- `std::launch::async` - wykonanie zadania w innym wątku,
- `std::launch::deferred` - wykonanie zadania zostaje odłożone na przyszłość, najprawdopodobniej będzie wykonane w tym samym wątku, który oczekuje na wynik.

Domyślną polityką jest `std::launch::async` |  
`std::launch::deferred`.

## Komunikacja producent-konsument

- Korzystając z wywołania asynchronicznego nie musimy dbać o poprawne przekazanie wyniku między wątkami.
- Przekazanie wartości odbywa się na zasadzie producenta i konsumenta. Zadanie wykonywane asynchronicznie jest producentem, a kod oczekujący na wynik jest konsumentem.
- **Jedna** wartość obliczeń jest **przenoszona** od producenta do konsumenta. Kopiowanie nie odbywa się.
- Mimo, że to tylko jedna wartość, to może to być kontener, albo duży obiekt, więc dobrze, że jest przenoszony, a nie kopiowany.

## Przykłady wywołania

Przekazujemy funkcję albo obiekt funkcyjny "foo". Argumentem wywołania funkcji bądź funktora będzie "2":

```
auto f = std::async(foo, 2);  
std::future<T> f = std::async(foo, 2);
```

Wynikiem wywołania funkcji "async" jest obiekt "f", który jest klasy "future". Do obiektu "f" będzie przeniesiony wynik wykonania funkcji bądź operatora funkcyjnego. Wynikiem może być "void".

A tak czekamy na wynik:

```
T wynik = f.get();
```

## std::future<T>

```
#include <future>
```

Klasa obiektu, do którego przenoszony jest wynik zadania. Ten obiekt staje się właścicielem wyniku: tylko ten obiekt otrzyma wynik z zadania.

Najważniejsza funkcja składowa: `get` - zwraca wynik, ale blokuje do czasu zwrotu wyniku. Jeżeli `T = void`, to funkcja nie zwróci wyniku.

Blokowanie wątku jest OK, bo wątek jest usypiany do momentu wybudzenia, gdy wynik jest gotowy.

Funkcja `wait` czeka na wynik, `valid` sprawdza, czy jest wynik.



## Przekazywanie wyjątku

Zadanie (funkcja lub funktor) może rzucić wyjątek, który jest propagowany do miejsca oczekującego na wynik. Funkcja “get” klasy “future” rzuci ten sam wyjątek, który obsługujemy poza zadaniem.

```
try
{
    f.get();
} catch (wyjatek w)
{
    std::cout << "Przechwycony!" << std::endl;
}
```

## Podsumowanie

- Zadania wywoływane z `std::async` najlepszą formą uruchamiania zadań równoległych.
- Funkcja `std::async` jest techniką wysokopoziomową. Programista nie przejmuje się szczegółami.
- Biblioteka standardowa zajmuje się szczegółami zwracania wyniku i obsługi wyjątków.

Dziękuję za uwagę.