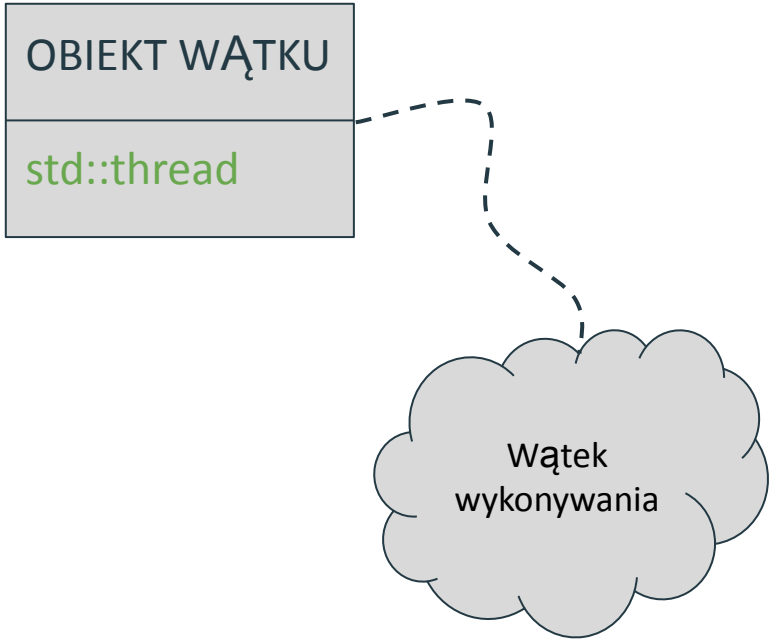
# Wątki w C++
# Biblioteka <thread>

Przemysław Dąbrowski

# Biblioteka <thread>

Udostępnia nam 2 klasy:

- std::thread - obiekt wątku reprezentujący pojedynczy wątek wykonywania
- std::jthread - to samo co std::thread ale pozwala na zatrzymanie wątku i automatycznie wykonuje join() przy destrukcji



# Biblioteka <thread>

Przestrzeń nazw std::this_thread z przydatnymi funkcjami:

- std::this_thread::get_id() - zwraca identyfikator wątku
- std::this_thread::yield() - wątek ustępuje pierwszeństwa innym np. czeka na ich zakończenie
- std::this_thread::sleep_untill() - usypia wątek aż do podanego momentu w czasie
- std::this_thread::sleep_for() - usypia wątek na podany okres

Wszystkie te funkcje dotyczą wątku w którym je wywołano

# std::thread, std::jthread

Rozpoczynają wykonywanie natychmiast po konstrukcji.

thread(F&& f, Args&&... args);

jthread(F&& f, Args&&... args);

Wartość zwracana przez funkcję f jest ignorowana.

Przekazanie wartości przez wskaźnik, referencję(std::ref(arg)) lub std::promise.

# Przykład 1

```cpp
void podprogram1(int x, int* y, int& z)// inkrementuje i wypisuje zmienne co 100ms
{
   std::cout << "Start podprogram1 w watku id: "<< std::this_thread::get_id() << std::endl;
   while(z<9)
   {   //inkrementacja przekazanych zmiennych
       ++x;
       ++(*y);
       ++z;
       std::cout << x << (*y) << z << std::endl; // wypisanie zmiennych
       std::this_thread::sleep_for(std::chrono::milliseconds(100)); // uśpij na 100ms
   }
   std::cout << "End podprogram1 w watku id: "<< std::this_thread::get_id() << std::endl;
}
int main()
{
   std::cout << "Main watek id: " << std::this_thread::get_id() << std::endl; // Main również ma wątek

   int a = 0, b = 0, c = 0; // Początkowe zmienne
   std::thread t1(podprogram1, a, &b, std::ref(c)); // Konstrukcja wątku dla funkcji podprogram1()
                                                    // i przekazanie parametrów
   std::cout << "Podprogram wykonuje sie, oczekiwanie na zakonczenie watku"<< std::endl;
   t1.join(); // zatrzymanie wątku main aż wątek t1 zakończy pracę
   std::cout << "watek zakonczyl prace: " << a << b << c << std::endl; // wypisanie zmiennych
}
```

```
Main watek id: 1
Podprogram wykonuje sie, oczeki
Start podprogram1 w watku id: 2
111
222
333
444
555
666
777
888
999
End podprogram1 w watku id: 2
watek zakonczyl prace: 099
```

# std::thread, std::jthread

Udostępniają metody:

- get_id() - zwraca identyfikator wątku
- joinable() - sprawdza czy jest wykonującym wątkiem
- hardware_concurrency() - zwraca wspieraną liczbę wątków współbieżnych, tylko wskazówka nie limit

# std::thread, std::jthread

Udostępniają operacje:

- join() - wstrzymuje wykonanie programu aż wątek zakończy pracę
- detach() - rozłącza obiekt wątku od wątku wykonywania, zasoby zwolnione po zakończeniu
- swap(std::thread& other) - zamienia się wątkami wykonywania z innym obiektem wątku

# Przykład 2

```
void foo(){std::this_thread::sleep_for(std::chrono::seconds(1));}
void bar(){std::this_thread::sleep_for(std::chrono::seconds(1));}
int main()
{
   std::cout << "Liczba wsp. watkow: " << std::thread::hardware_concurrency() <<  std::boolalpha << std::endl;

   std::thread t1;
   std::cout << "przed rozpoczeciem, joinable: "<< t1.joinable() << '\n';
   t1 = std::thread(foo);
   std::cout << "po rozpoczeciu, joinable: "<< t1.joinable() << '\n';

   t1.join();
   std::cout << "po t1.join(), joinable: "<< t1.joinable() << '\n';
   t1 = std::thread(foo);
   t1.detach();
   std::cout << "po t1.detach(), joinable: "<< t1.joinable() << '\n';

   t1 = std::thread(foo);
   std::thread t2(bar);
   std::cout << "watek t1 id: " << t1.get_id() << '\n' << "watek t2 id: " << t2.get_id() << '\n';

   t1.swap(t2);
   std::cout << "after t1.swap(t2):" << '\n' << "watek t1 id: " << t1.get_id() << '\n' << "watek t2 id: "
   << t2.get_id() << '\n';
   t1.join();
   t2.join();
}
```

```
Liczba wsp. watkow: 4
przed rozpoczeciem, joinable: false
po rozpoczeciu, joinable: true
po t1.join(), joinable: false
po t1.detach(), joinable: false
watek t1 id: 4
watek t2 id: 5
after t1.swap(t2):
watek t1 id: 5
watek t2 id: 4
```

# std::thread, std::jthread

Żadne dwa obiekty thread/jthread nie mogą przedstawiać tego samego wątku wykonywania.

Obiekty thread/jthread mogą nie przedstawiać żadnego wątku, np. po utworzeniu domyślnym konstruktorem lub po wywołaniu thread::join() czy thread::detach().

# std::jthread

Zawiera dodatkowy współdzielony stan zatrzymania powiązany z std::stop_source, do którego jest dostęp z innych wątków.

Udostępnia metody:

- request_stop() - zgłasza żądanie zatrzymania do stanu zatrzymania
- get_stop_token() - zwraca std::stop_token powiązany ze stanem zatrzymania
- get_stop_source() - zwraca std::stop_source powiązane ze stanem zatrzymania

# std::jthread

std::stop_token umożliwia sprawdzenie czy zostało zgłoszone żądanie zatrzymania do powiązanego stanu zatrzymania poprzez stop_requested().

std::stop_source umożliwia zgłoszenie żądania zatrzymania poprzez request_stop(), sprawdzenie stanu przez stop_requested(), oraz get_token() zwraca std::stop_token powiązany ze stanem zatrzymania.

# Przykład 3

```
void foo(const std::stop_token& st, int& x)
{
   while(!st.stop_requested())
   {
       ++x;
       std::cout << x << std::endl;
       std::this_thread::sleep_for(std::chrono::milliseconds(450));
   }
   std::cout << "Wykryto stop requested w foo\n";
}
void bar(const std::stop_token& st, std::stop_source ss)
{
   std::this thread::sleep_for(std::chrono::seconds(2));
   ss.request stop();
   std::cout << "stop request w bar\n";
}
int main()
{
   int a=0;
   std::jthread jt1(foo,std::ref(a));
   std::stop token stopToken = jt1.get stop token();
   std::stop_source stopSource = jt1.get_stop_source();

   std::jthread jt2(bar, stopSource);
   while(!stopToken.stop requested()){}
   std::cout << "Wykryto stop requested w main\n";
}
```

```
1
2
3
4
5
stop request w bar
Wykryto stop requested w main
Wykryto stop requested w foo
```

## std::thread

Konstruktor: thread(F&& f, Args&&... args)

Metody:

get_id() - zwraca id wątku

joinable() - sprawdza czy wątek aktywny

hardware_concurrency() - wspierana liczba współbieżnych wątków, wskazówka nie limit

join() - wstrzymuje wykonanie programu aż wątek zakończy pracę

detach() - rozłącza obiekt wątku od wątku wykonywania

swap(std::thread& other) - zamiana wątkami z obiektem other

## std::jthread dodatki

request_stop() - zgłasza żądanie zatrzymania

get_stop_token() - zwraca powiązany std::stop_token

get_stop_source() - zwraca powiązany std::stop_source

## std::this_thread

get_id() - zwraca id tego wątku

yield() - wątek ustępuje pierwszeństwa innym

sleep_untill() - usypia wątek aż do podanego momentu w czasie

sleep_for() - usypia wątek na podany okres czasu

## std::stop_token

stop_requested() - sprawdza czy zażądano zatrzymania

## std::stop_source

stop_requested() - sprawdza czy zażądano zatrzymania

request_stop() - zgłasza żądanie zatrzymania

get_stop_token() - zwraca powiązany std::stop_token

Zad. 1 (0.1)

Napisz funkcję z argumentem “int& arg”, która co 100 milisekund inkrementuje podany argument aż osiągnie wartość >=100. W main napisz program, który tworzy wątek dla powyższej funkcji z argumentem = 0. Gdy użytkownik wprowadzi coś do konsoli wypisz wprowadzony ciąg oraz aktualną wartość przekazanego argumentu (“hello”-->”hello 26”). Gdy wprowadzony ciąg = ”z” poczekaj na zakończenie pracy wątku i wypisz adekwatny komunikat gdy tak się stanie.

Zad. 2 (0.2)

std::vector<int> fib = {0,1};

Napisz funkcję f1, która przyjmuje referencję wektora fib jako argument i co 1ms oblicza i dodaje do wektora kolejny element ciągu Fibonacciego. Funkcja f1 musi mieć możliwość zatrzymania z innego wątku.

Napisz funkcję f2, która przyjmuje referencję wektora fib jako argument i co 2sek wypisuje liczbę wyliczonych elementów ciągu oraz pyta użytkownika Czy zakończyć dalsze obliczenia? gdy wprowadzono “t” zatrzymać wątek funkcji f1. W main utworzyć wątki(std::jthread) dla funkcji f1 i f2 i prawidłowo przekazać argumenty.

# MATERIAŁY POMOCNICZE

https://github.com/user-3141/pk4thread

```
std::vector<long long> GetFactors(long long n)
{
    std::vector<long long> factorization;
    for (long long d = 2; d * d <= n; d++) {
        while (n % d == 0) {
            factorization.push_back(d);
            n /= d;
        }
    }
    if (n > 1)
        factorization.push_back(n);
    return factorization;
}
```

```
std::vector<long long> numbers = {343533055656878036,
        5860350457319753452, 7893707073704431551,
        4368313902489077849, 1677941465804505455,
        3297139368709694577, 2233241309328876460,
        1429423566119835739, 6516264611898389790,
        2442897970716511172, 4090212017170405777,
        1153505978712351257, 612345904729275297,
        2967311862877440092, 7436859525682139307,
        8675552455193670309, 5341853797610081834,
        8839889067931964805, 2075530006345431709,
        6715788216943620268, 865949283066326930,
        240626470966425323, 3392424271198655643,
        4993821633716302940,  5808357568576893759,
        7063755934051908169, 5318084116412572738,
        7927635782097829856, 4394970874296051547,
        5103438212726077733,3864573979507605197,
        163382648627135698};
```

POMIAR CZASU

```
auto poczatek= std::chrono::high_resolution_clock::now();

//program

auto koniec = std::chrono::high_resolution_clock::now();
long long int czas = std::chrono::duration_cast<std::chrono::milliseconds > (koniec - poczatek).count();
//czas trwania w ms
```

Zad. 3 (0.2)

Napisz funkcję f przyjmującą jako argumenty wektor numbers, int beginIndex, int endIndex, wektor factors, która za pomocą funkcji GetFactors() wykonuje faktoryzację liczb z wektora numbers z przedziału wskazanego przez przekazane indeksy. W main utwórz jeden wątek dla funkcji f i zmierz czas wykonania faktoryzacji wszystkich liczb z numbers. Następnie rozdziel wykonywanie faktoryzacji na 2 wątki funkcji f i zmierz czas faktoryzacji. Porównaj zmierzone czasy.