

# Zaawansowane programowanie w C++ (ZPR)

## Wykład 11 – C++ i współbieżność

Robert Nowak

25L

# Równoległość - obliczenia podczas obsługi urządzeń

- ▶ bardzo wolna reakcja człowieka
- ▶ wolne urządzenia wejścia - wyjścia (np. drukarki)
- ▶ bardzo szybkie procesory

Aplikacja wielowątkowa:

- ▶ pozwala reagować na zlecenia użytkownika podczas obliczeń
- ▶ lepiej wykorzystuje dostępną moc obliczeniową
- ▶ może obsługiwać wiele zleceń równolegle
- ▶ **poprawnie zaprojektowana** jest szybsza na platformach wieloprocessorowych (wielordzeniowych)

Wątek („lekki proces”) – niezależne ciągi instrukcji w ramach procesu

- ▶ wątki współdzielą kod, dane oraz zasoby.
- ▶ mechanizm przełączania nie wprowadza dużych narzutów

# C++ pozwala implementować aplikacje wielowątkowe

Współdzielone	Niezależne
<ul style="list-style-type: none"><li>▶ obiekty dynamiczne</li><li>▶ obiekty automatyczne, jeżeli dostęp przez wskaźnik</li><li>▶ obiekty globalne</li><li>▶ zasoby systemu operacyjnego (pliki, okna itp)</li><li>▶ kod wykonywany</li></ul>	<ul style="list-style-type: none"><li>▶ rejestry</li><li>▶ ciąg wykonywanych instrukcji</li><li>▶ stos</li><li>▶ zmienne automatyczne</li></ul>

Program ma zawsze jeden wątek (funkcja `main()` ) - wątek główny lub inicjujący. Może tworzyć dodatkowe wątki.

# Wątki w C++

Zabronione funkcje biblioteki standardowej (przechowują stan pomiędzy wołaniami):

funkcja	biblioteka	zamiennik
<b>rand</b>	cstdlib	random, Boost.Random
<b>strtok</b>	cstring	regex, Boost.Tokenizer, Boost.Regex
<b>asctime</b> <b>ctime gmtime</b> <b>localtime</b>	ctime	chrono, Boost.Chrono, Boost.Date_Time

- ▶ należy używać bibliotek przeznaczonych do pracy wielowątkowej
- ▶ standard C++03 nie dostarczał mechanizmów tworzenia i synchronizacji wątków, standard C++11 ma te mechanizmy,
- ▶ można było używać wątków mechanizmami dostarczanyymi przez system operacyjny

# Biblioteka Boost.DateTime

nazwa	opis	wielkość
date	data (gregoriańska) od 1 stycznia 1400 do 31 grudnia 9999	4B
date_duration	długość w czasie (w dniach)	4B
date_period	przedział czasu	8B
ptime	punkt w czasie (date+time_duration). Dokładność: <i>ms</i> (lub <i>ns</i> ).	8 lub 12B
time_duration	długość w czasie	4 lub 8B
time_period	przedział czasu	16 lub 24B

```
gregorian::date d(2011,gregorian::Apr,20); //data 20 kwietnia 2011
posix_time::ptime t1(d,posix_time::time_duration(18,0,0) );//18:00
posix_time::ptime t2(d, posix_time::hours(12)+posix_time::seconds(4));
posix_time::time_duration td = t2 - t1; //odejmowanie, 4 sekundy
posix_time::ptime t3 = t2 + td*10; //mnożenie przez liczbę
```

# Biblioteka std::chrono (Boost.Chrono)

Biblioteka wykorzystywana w boost::thread (od wersji 1.49)

nazwa	opis
time_point	reprezentuje punkt w czasie
duration	długość w czasie, przechowuje liczbę cykli oraz długość cyklu
clock	dostarcza funkcji now, bieżący punkt w czasie

```
#include <chrono>
```

```
duration<long, milli > d1; //licznik milisekund używając typu long
```

```
duration<int, ratio<60> > d2; //licznik minut używając typu int
```

```
this_thread::sleep_for( milliseconds(1000)); //argument typu duration
```

```
system_clock::time_point time_limit = system_clock::now() + seconds(1);
```

```
this_thread::sleep_until(time_limit); //wykorzystuje time_point
```

# std::thread - wątki w C++

```
#include <thread>
//Funkcja główna wątku użytkownika
void my_thread() { /* ... */ }
//Można też użyć funktora
class MyThread {
public:
    //tutaj implementacja funkcji wątku użytkownika
    void operator()() { /* ... */ }
};
int main() {
    std::thread thrd(&my_thread); //Utworzenie i uruchomienie wątku
    try {
        thrd.join(); //Bieżący watek czeka na zakończenie wątku thrd
    } catch(...) { } //wyjątek zgłaszany, gdy wątek już nie istnieje
    return 0;
}

//liczba procesorów (rdzeni) dostępnych na platformie
unsigned int n = std::thread::hardware_concurrency();
```

# Skalowalność - zapewnienie wydajnej pracy współbieżnej

Wątki mogą pracować niezależnie gdy:

- ▶ odczytują współdzielone dane
- ▶ zapisują własne (lokalne) dane

Zapis współdzielonych danych jest kłopotliwy (spowalnia)

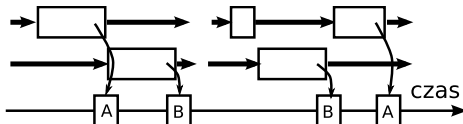
Poprawne wykorzystanie współbieżności gdy:

- ▶ można podzielić problem na niezależne pod-problemy, np. ten sam algorytm na różnych danych, ale wtedy lepiej stosować operacje wektorowe, maszynę SIMD (jeden strumień instrukcji, wiele strumienie danych), np. GPU;
- ▶ kod, który jest podatny na wyścigi jest w sekcji krytycznej, istotne: jak najmniej czasu spędzanego w sekcji krytycznej;
- ▶ algorytm używa instrukcji atomowych, istotne: konflikty są rzadkie.



# Wyścigi (race conditions)

Jeżeli wątki zapisują współdzielony obiekt, to może wystąpić niezdefiniowane zachowanie



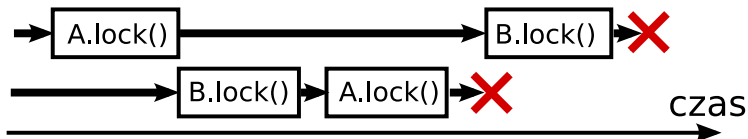
Jedno z rozwiązań: synchronizacja (blokady)

```
mutex m;
m.lock();
//Sekcja krytyczna
m.unlock();
```

```
std::mutex mutex_resource;
{
    std::lock_guard guard(mutex_resource);
    //sekcja krytyczna
}
```

# Zakleszczenia (deadlock)

każdy z wątków czeka na jakiś inny (jest blokowany). Żaden z nich nie może dalej pracować.



Rozwiązanie:

- ▶ zajmowanie blokad zawsze w tej samej kolejności
- ▶ stosowanie innych mechanizmów synchronizujących

# Samo-zakleszczenie

brak zwalniania sekcji krytycznej (np. w algorytmach rekurencyjnych)

```
struct Foo {  
    mutex m;  
    void recFun(){  
        std::lock_guard l(m);  
        /* ... */  
        recFun(); //Wołanie rekurencyjne  
    };  
};
```

Rozwiązanie: **std::recursive\_mutex**

# Rodzaje prostych blokad w bibliotece standardowej

- ▶ `std::mutex`
  - ▶ **lock** wolny: zajmuje, zajęty: blokuje
  - ▶ **try\_lock** wolny: zajmuje (zwraca true), zajęty (zwraca false)
- ▶ `std::recursive_mutex`
  - ▶ **lock** jak wyżej
  - ▶ **try\_lock** jak wyżej
- ▶ `std::timed_mutex`
  - ▶ **lock** jak wyżej
  - ▶ **try\_lock** jak wyżej
  - ▶ **try\_lock\_for**(time duration)  
wolny: zajmuje (zwraca true), zajęty: blokuje na dany czas
  - ▶ **try\_lock\_until**(time point)  
wolny: zajmuje (zwraca true), zajęty: blokuje na dany czas

# Skalowalność z użyciem blokad (sekcji krytycznych)

Wyróżnia się „szybką” i „wolną” ścieżkę w funkcjach wykonywanych w wątkach

- ▶ brak blokad na ”szybkiej ścieżce” (blokady, wykorzystywane nawet tylko do odczytu zapisują stan)
- ▶ usuwanie współdzielonych obiektów do zapisu (lepiej informację wyjściową przechowywać lokalnie, a później scalać wynik)

*//Przykład: singleton wielowątkowy*

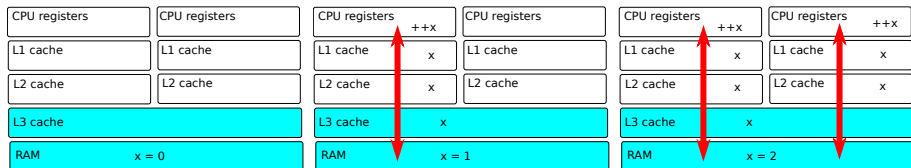
```
Singleton& Singleton::getInstance() {  
    if(!pInstance_) { //wzorzec podwójnego sprawdzania  
        std::lock_guard guard(mutex_); //tworzenie w sekcji krytycznej  
        if(!pInstance_)  
            pInstance_ = new Singleton;  
    }  
    return *pInstance_;  
}
```

# Instrukcje atomowe, std::atomic (C++11), Boost.Atomic

- ▶ niektóre operacje są transakcjami (wykonują się w całości, albo wcale)
- ▶ za pomocą takich operacji możemy tworzyć algorytmy współbieżne bez blokad (lock-free)
- ▶ takie algorytmy mogą działać szybciej, niż tworzenie sekcji krytycznych.

Operacje atomowe: gwarantuje sprzęt (procesor), ale nie tylko!

```
int y = 0;
++y; //nie jest atomowa!
std::atomic<int> x(0);
++x; //instrukcja atomowa, blokujący dostęp do pewnego obszaru pamięci
```



# Typy danych, które mogą być atomowe

Atomowe mogą być wszystkie typy, dla których kopiowanie jest trywialne (obiekt to ciągły obszar pamięci, nie ma funkcji wirtualnych).

```
std::atomic<int> i=0; //OK
std::atomic<double> d=0.0; //OK
struct A { long x; long y; };
std::atomic<A> a; //OK, nie zawsze lock-free!
```

`std::atomic_flag` – gwarancja, że jest lock-free (to nie `atomic<bool>`)

Operacje atomowe na typach atomowych:

<code>is_lock_free</code>	bada, czy typ nie używa blokad
<code>(constructor)</code>	tworzy obiekt
<code>operator=</code>	
<code>store</code>	zast. wartość atomową, argument nieatomowy
<code>load</code>	zwraca wartość atomową jako obiekt nieatomowy
<code>compare_exchange_weak</code>	podstawowa operacja dla algorytmów lock-free
<code>compare_exchange_strong</code>	podstawowa operacja dla algorytmów lock-free

Dla niektórych typów (np. `atomic<int>`) mamy też inne operacje atomowe.

# Modele synchronizacji pamięci

Parametr metod load, store obiektów atomowych, np.

```
atomic<int> x;
x.store(1, memory_order_relaxed);
```

nazwa	opis
memory_order_relaxed	brak gwarancji
memory_order_consume	zazwyczaj load
memory_order_acquire	zazwyczaj load
memory_order_release	zazwyczaj store
memory_order_acq_rel	jednocześnie 'acquire' i 'release';
memory_order_seq_cst	j.w. + wspólna kolejność, <b>domyślny</b> .

```
//Wątek 1
```

```
x.store(1);
y.store(2);
```

```
//Wątek 2
```

```
if(y.load() == 2) {
    assert( x.load() == 1);
    y.store(1); }
```

```
//Wątek 3
```

```
if( y.load() == 1)
    assert( x.load() == 1 );
```



# Parametry platformy związane ze współbieżnością

- ▶ `std::thread::hardware_concurrency` – omówiony wcześniej
- ▶ `std::hardware_destructive_interference_size` (C++17) – minimalny odstęp pomiędzy obiektami, aby gwarantować różne linie w pamięci podręcznej (cache L1)
- ▶ `std::hardware_constructive_interference_size` (C++17) – maksymalny rozmiar obiektu, który jest możliwy do przechowywania w tej samej linii pamięci podręcznej (cache L1)

# Obiekty związane z wątkami

`std::thread_local` (C++11), `boost::thread_specific`  
obiekty związane z bieżącym wątkiem

```
int random_dice() {
    thread_local std::mt19937 engine; //obiekt związany z wątkiem
    //dla każdego wątku to inny obiekt, nie potrzebna synchronizacja
    uniform_int_distribution<mt19937::result_type> dist(1,6);
    return dist(engine);
}

void thread_fun() {
    for(int i=0; i<10; ++i)
        random_dice();
}

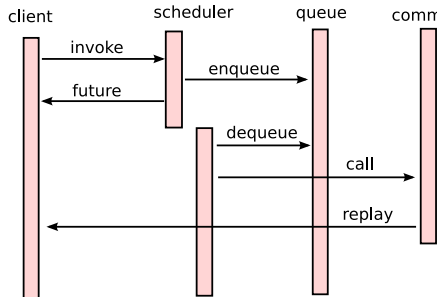
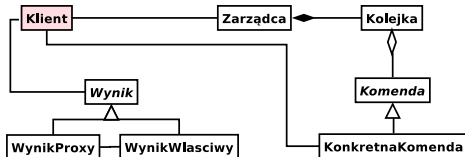
int main() {
    std::thread th1(thread_fun);
    std::thread th2(thread_fun);

    th1.join(); th2.join();
    return 0;
}
```

# Aktywny obiekt - asynchroniczne wołanie komend

Komenda wykonuje się w niezależnym wątku, wersje:

- ▶ komendy nie zwracające wyniku,
- ▶ komendy zwracające wynik.



# std::promise oraz std::future

Asynchroniczne przekazywanie obiektu:

- ▶ `std::future<T>` - wynik (typ lub wyjątek), z punktu widzenia odbiorcy (czytelnika)
- ▶ `std::promise<T>` - wynik z punktu widzenia nadawcy (pisarza)

```
Result long_computaton();  
//uruchamia w oddzielnym wątku  
future<Result> result = async( launch:async, long_computation );  
//...  
result.get(); //czeka na na wynik
```

- ▶ `std::launch::async` - wykonuje w oddzielnym wątku
- ▶ `std::launch::deferred` - wykonuje w tym samym wątku, leniwe wykonanie, gdy `get` na `future`

# STL i współbieżność

- ▶ bezpiecznie czytanie – wiele wątków może równocześnie czytać z kontenera
- ▶ bezpieczne pisanie do różnych kontenerów – różne wątki mogą równocześnie pisać do różnych kontenerów

(od C++17) dostarcza algorytmy równoległe i wektorowe

```
std::execution::sequenced_policy – sekwencyjne
std::execution::parallel_policy – można używać wątków
std::execution::unsequenced_policy – można oper. wektorowe
std::execution::parallel_unsequenced_policy
```

```
vector<int> v(100);
fill( execution::unseq, v.begin(), v.end(), 1); //ustawia wartości w kontenerze
std::mt19937_64 gen; //generator liczb pseudolosowych, Mersenne-Twister
std::uniform_int_distribution<> distr(0,99);
generate( v.begin(), v.end(), [&]() { return distr(gen); });
```

# CUDA, Thrust, Boost.Compute

**Thrust** Umożliwia używanie GPU z architekturą CUDA w C++.

Kontenery: `thrust::host_vector` – wektor w pamięci operacyjnej  
`thrust::device_vector` – wektor na GPU

**Boost.Compute** Umożliwia używanie CPU i/lub GPU za pośrednictwem OpenCL.

- ▶ niezależna od dostawcy sprzętu (ponieważ używa OpenCL)
- ▶ bezpośredni dostęp do równoległego sprzętu (nie jest to warstwa abstrakcji)
- ▶ pozwala wykorzystywać typowe kontenery (np. `vector<T>`)
- ▶ pozwala wykorzystywać typowe algorytmy (np. `std::transform`)

```
g++ compute.cpp -lOpenCL
```

# Struktury danych bez blokad (lock-free)

```
bool compare_exchange(atomic<T>* obj, T* exp, T val)
    if(obj==exp) { obj = val; return true;}
    else { exp = obj; return false; }
```

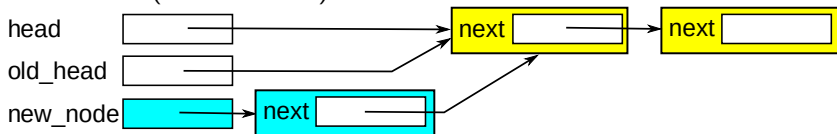
```
struct Node { int value; Node* next; };
std::atomic<Node*> head(nullptr);
```

```
void push_front(int val) { //może być wołane w różnych wątkach
    Node* old_head = head.load();
    Node* new_node = new Node {val,old_head};
    //założenie - konflikty występują rzadko
    while (! head.compare_exchange_weak(old_head,new_node))
        new_node->next = old_head;
}
```

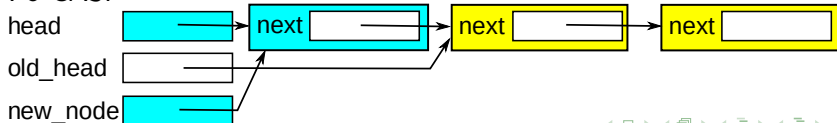
# Struktury danych bez blokad (lock-free) (2)

```
void push_front(int val) { //może być wołane w różnych wątkach
    Node* old_head = head.load();
    Node* new_node = new Node {val,old_head};
    //założenie - konflikty występują rzadko
    while (! head.compare_exchange_weak(old_head,new_node))
        new_node->next = old_head;
}
```

Przed CAS (bez konfliktu):



Po CAS:

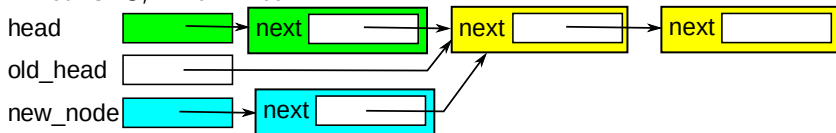




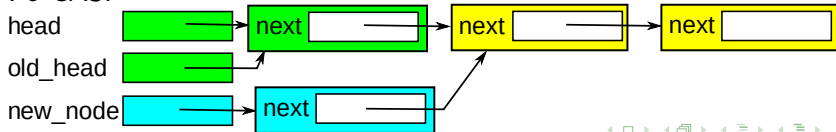
# Struktury danych bez blokad (lock-free) (3)

```
void push_front(int val) { //może być wołane w różnych wątkach
    Node* old_head = head.load();
    Node* new_node = new Node {val,old_head};
    //założenie - konflikty występują rzadko
    while (! head.compare_exchange_weak(old_head,new_node))
        new_node->next = old_head;
}
```

Przed CAS, z konfliktem:



Po CAS:



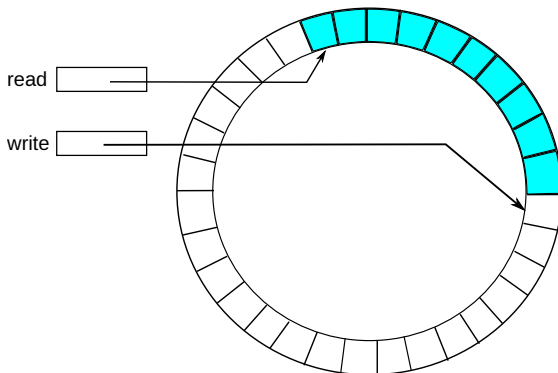
# Problem ABA w algorytmach bez blokad

W wypadku dwukrotnego odczytu lokacji pamięci, gdy odczytana wartość jest taka sama, to jest interpretowana jako 'nic się nie zmieniło'

- ▶ wątek  $P_1$  czyta wartość A, np. wskaźnik na element listy
- ▶ wątek  $P_2$  zmienia wartość z A na B, np. element jest usuwany
- ▶ wątek  $P_2$  zmienia wartość z B na A, nowy element listy jest dodany, ale w tym miejscu pamięci
- ▶ wątek  $P_1$  czyta wartość A, zakłada, że się nic nie zmieniło, a tutaj ten sam wskaźnik, ale może być inna wartość wskazywana

# Bufor cykliczny lock-free

Bufor cykliczny – bardzo dobry wybór w przypadku jeden producent danych, jeden konsument. Kolejka (FIFO) o ustalonym rozmiarze.



Można się zgłosić, zaimplementować i zbadać, w ramach zadania dodatkowego.

# Aplikacje współbieżne, podsumowanie

Mogą wystąpić wyścigi:

- ▶ błędy są niepowtarzalne,
- ▶ różne zachowanie się wersji debug od release,
- ▶ różne zachowanie się na platformach jedno i wielo-procesorowych.

Niezbędny poprawny projekt – nie da się wychwycić w statycznej analizie kodu (kompilacja) i w testach!

Zapobieganie wyścigom: sekcje krytyczne lub operacje atomowe

- ▶ operacje atomowe są wolniejsze niż te same operacje nieatomowe, nawet dla typów takich jak 'int' (czekają na dostęp do linii cache),
- ▶ operacje atomowe mogą być szybsze lub wolniejsze niż sekcje krytyczne
- ▶ sekcje krytyczne zawsze zmniejszają skalowalność

# Biblioteki związane ze współbieżnością

- ▶ **Boost.Interprocess** komunikacja pomiędzy procesami  
Przenośne mechanizmy komunikacji pomiędzy procesami (aplikacjami).
  - ▶ pamięć dzielona (shared memory)
  - ▶ mechanizmy synchronizujące w tej pamięci (semafony, zmienne warunkowe)
  - ▶ komunikacja przez pliki
  - ▶ kolejki komunikatów
- ▶ **OpenMP** - interfejs do tworzenia aplikacji wykorzystujących współbieżność
- ▶ **Boost.Asio** asynchroniczna obsługa urządzeń wejścia-wyjścia

***Dziękuję***