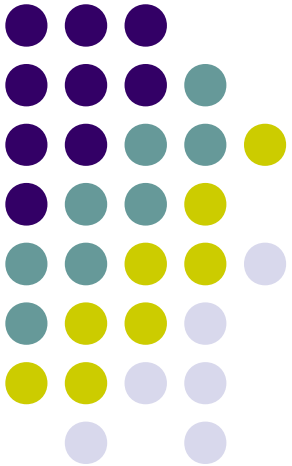
Średniozaawansowane programowanie w C++

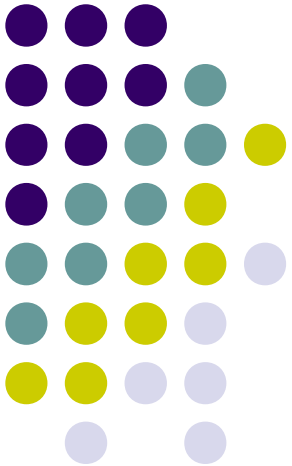
Wykład #4

3 listopada 2016 r.



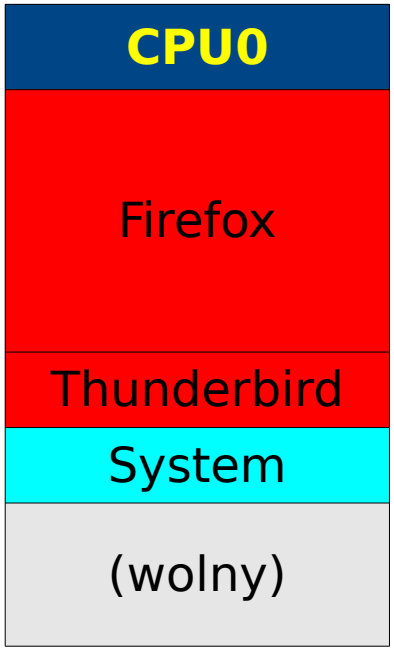
Systemy wieloprocesorowe

Czy więcej znaczy lepiej?

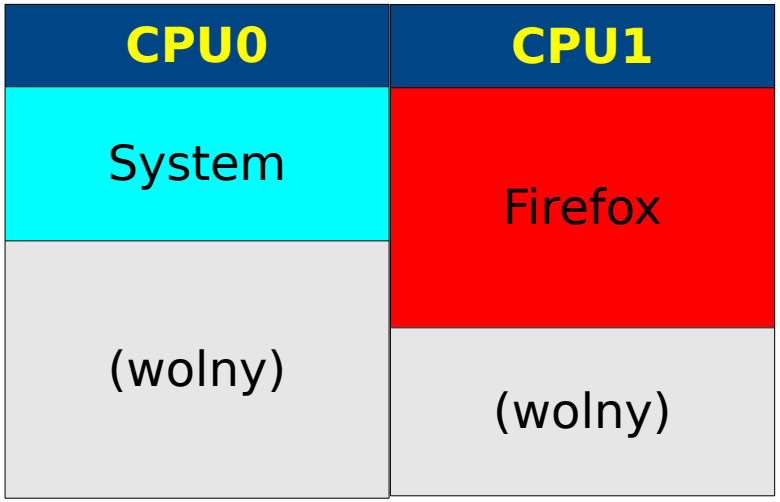


Zarządzanie procesorami przez OS

System jednoprocesorowy



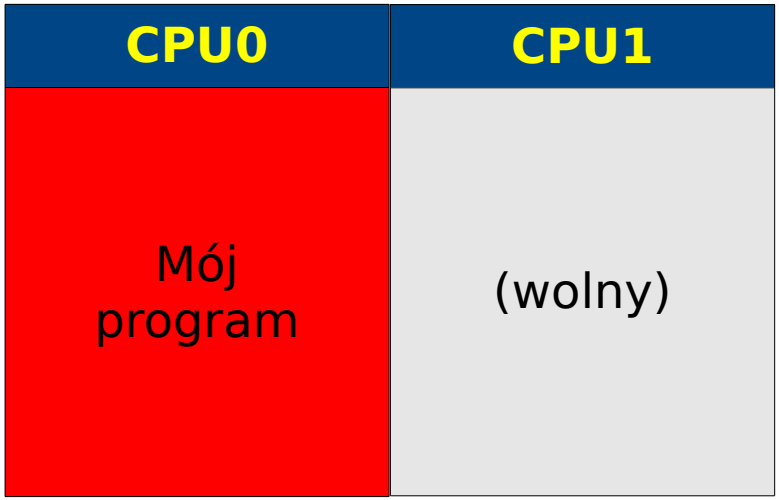
System dwuprocesorowy



Aplikacje jednowątkowe

System jednoprocesorowy

System dwuprocesorowy

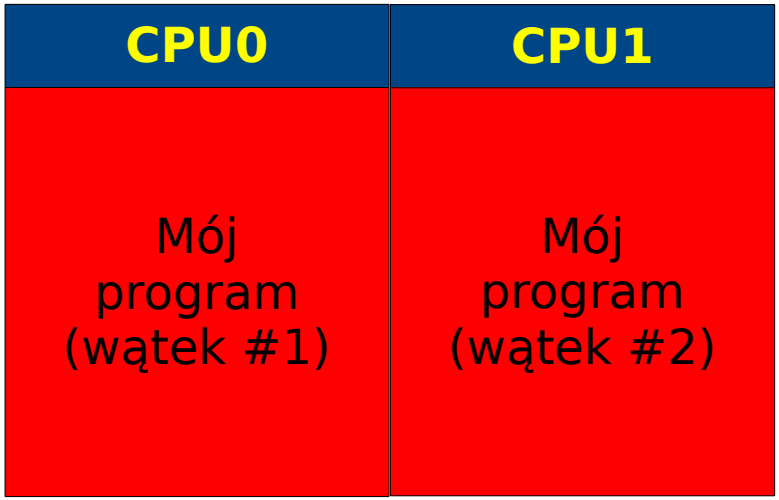


Aplikacje niezaprojektowane jako wielowątkowe będą działać szybciej na komputerach z jednym szybkim rdzeniem, niż z dwoma wolniejszymi!

Aplikacje wielowątkowe

System jednoprocesorowy

System dwuprocesorowy

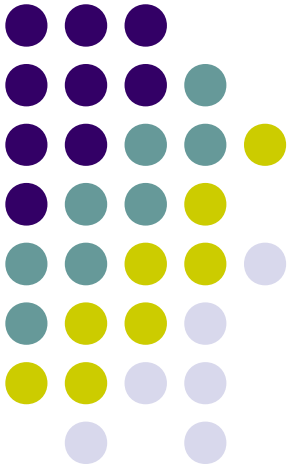


Dopiero jawne wydzielenie drugiego wątku prowadzi do skorzystania z zalet procesorów wielordzeniowych!

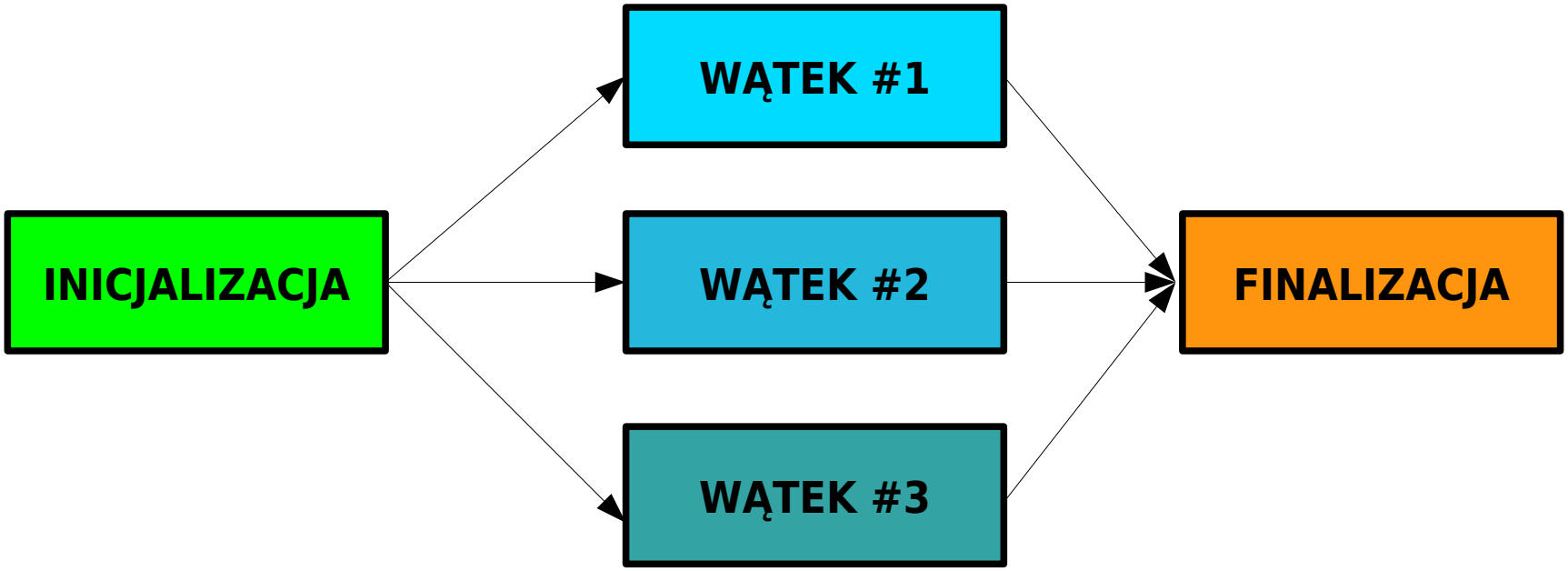
Obliczenia wielowątkowe

Idea

Implementacja



Program wielowątkowy



Nie wszystkie etapy programu można wykonywać wielowątkowo (np. wczytywanie dużego pliku)!
Etapy jedno i wielowątkowe mogą się wielokrotnie przeplatać.

std::thread

```
#include <thread>

void funkcja_drugiego_watku ()
{
        // (…) robi coś
}

int main ()
{
        // (…) wczytujemy dane itp.
        // otwieramy drugi wątek:
        std::thread drugi_watek (funkcja_drugiego_watku);
        // (…) coś dalej miziamy
        drugi_watek.join ();    // czekamy na zakończenie drugiego wątku
        // (…) kończymy obliczenia jednowątkowo
        return 0;
}
```

Zalety: obliczenie równoległe na ogół wykonują się szybko
Wady: funkcja_drugiego_wątku musi być void (void) – problemy ze sterowaniem wątkiem i przechwytywaniem jego wyników (konieczność użycia zmiennych globalnych).

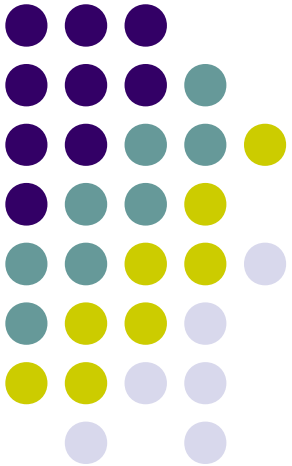
std::mutex

```
#include <mutex>

float tablica [100];    // tablica globalna

class KlasaWspoldzielona
{
        public:
                void kwadrat (int i) {
                        blokada.lock ();        // zakładanie blokady
                        tablica [i] *= tablica [i];
                        blokada.unlock ();      // zdejmowanie blokady
                }
        private:
                std::mutex blokada;
};
```

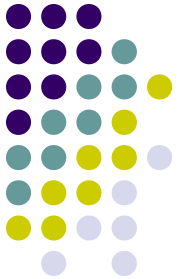
Blokada blokuje sekcję na wyłączność dla jednego wątku. Pozostałe wątki czekają w kolejce na zwolnienie blokady.
Dla wygody można używać std::unique_lock

Przykład

Obliczanie wartości średniej

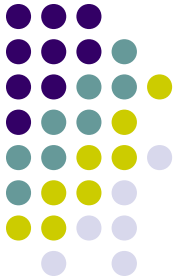


srednia.hpp



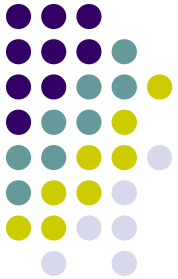
```
#ifndef _srednia_hpp_
#define _srednia_hpp_
#include <vector>

typedef std::vector<double>::const_iterator iter;

class Srednia
{
  public:
    // Konstruktor zapamiątuje „jego” zakres tablicy
    Srednia (const iter &pocz, const iter &konc);
    double wartosc () const {return srednia_;}
    void operator() (); // wykonuje obliczenia
    int ilosc () const {return n_;}
  private:
    double srednia_;
    int n_;     // liczba elementów w zakresie
    iter pocz_; // początek zakresu
    iter konc_; // pierwszy element za zakresem
};

#endif
```

srednia.cpp



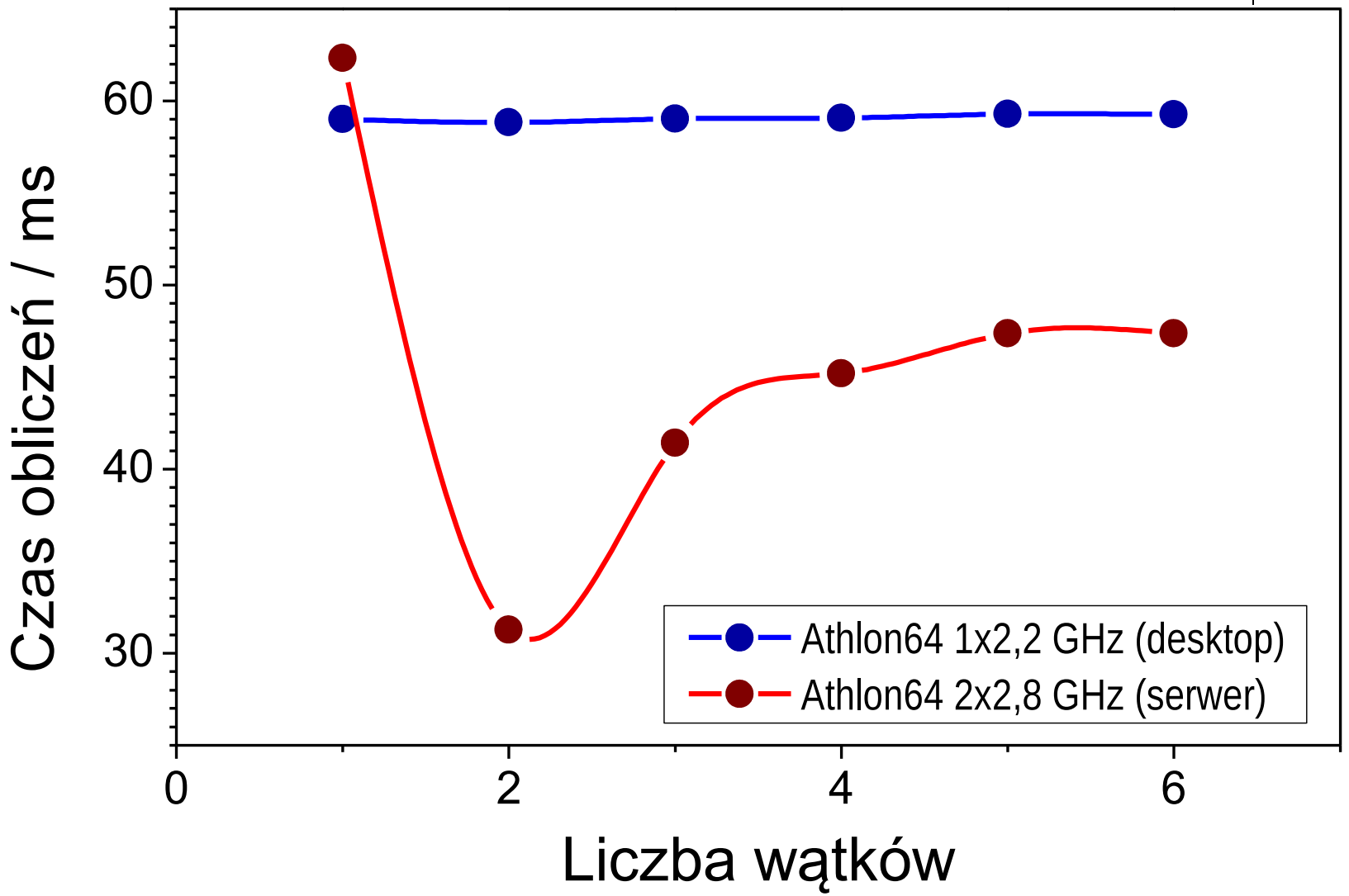
```
#include "srednia.hpp"
#include <algorithm>
#include <boost/lambda/lambda.hpp>

Srednia::Srednia (const iter &pocz, const iter &konc) :
        pocz_ (pocz), konc_ (konc)
{
  n_ = konc_ - pocz_;
}

void Srednia::operator() ()     // wykonuje obliczenia
{
  srednia_ = 0.0;
  std::for_each (pocz_, konc_, srednia_ += boost::lambda::_1);
  srednia_ /= n_;
}
```

main.cpp



```
#include <boost/thread.hpp>
#include "srednia.hpp"

double licz_srednia (const std::vector <double> &dane, int p) // p – liczba wątków
{
  std::vector <Srednia*> srednie;
  boost::thread_group watki;

  int n = dane.size();

  for (int i = 0; i < p; ++i) // przydziela zakresy poszczególnym obiektom
    srednie.push_back (new Srednia (dane.begin() + n*i/p, dane.begin() + n*(i+1)/p));
  for (int i = 0; i < p; ++i) // tworzy wątki
    watki.create_thread (std::ref (*(srednie [i])));


  watki.join_all (); // czeka na zakończenie wszystkich obliczeń

  double srednia = 0.0;
  for (int i = 0; i < p; ++i) // oblicza średnią średnich
    srednia += srednie[i]->wartosc() * srednie[i]->ilosc();
  srednia /= dane.size ();

  for (int i = 0; i < p; ++i)
    delete srednie[i];

  return srednia;
}
```

Testy wydajności

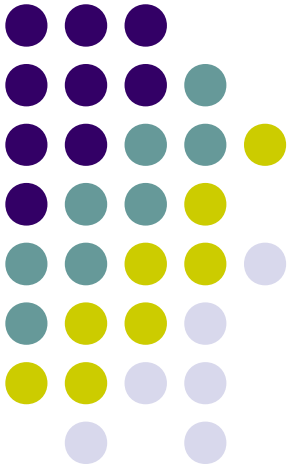


boost::posix_time

He who controls the past commands the future

He who commands the future, conquers the past.

Kane



Dostępne zegary

```
#include <boost/date_time/posix_time/posix_time.hpp>

using namespace boost::posix_time;
```

Zegar	Opis
second_clock::local_time	Czas lokalny z dokładnością do sekundy
second_clock::universal_time	Czas UTC z dokładnością do sekundy
microsec_clock::local_time	Czas lokalny z dokładnością do mikrosekundy
microsec_clock::universal_time	Czas UTC z dokładnością do mikrosekundy

Ww. funkcje zwracają obiekt daty/czasu typu *ptime*.

Dokładność zegarów mikrosekundowych w systemie Windows™ może być mniejsza niż deklarowana.

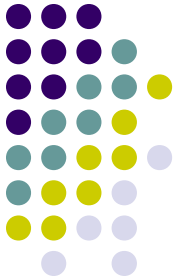
Funkcje jednostek czasu

```
using namespace boost::posix_time;
```

Funkcja	Jednostka wartości zwracanej
hours (long)	godzina
minutes (long)	minuta
seconds (long)	sekunda
milliseconds (long)	milisekunda
microseconds (long)	mikrosekunda
nanoseconds (long)	nanosekunda

```
time_duration siesta = hours (1) + minutes (30);
time_duration przerwa = milliseconds (10);
```

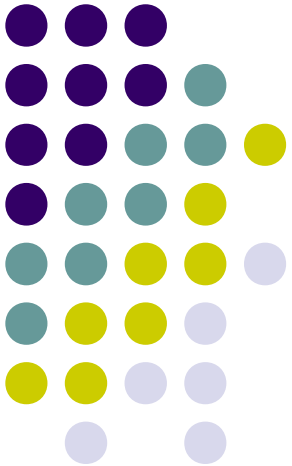
ptime / time_duration – przykłady



```
#include <boost/date_time/posix_time/posix_time.hpp>

using namespace boost::posix_time;

ptime czas_start  = microsec_clock::universal_time();
cout << "Pomiar rozpoczeto o " << to_simple_string (czas_start) << endl;
// ALBO: cout << "Pomiar rozpoczeto o " << czas_start << endl;

time_duration czas_trwania = minutes (2) + seconds (30);
// ALBO: time_duration czas_trwania (0, 2, 30, 0);

ptime czas_biezacy;
do {
        // (…) pomiar
        czas_biezacy = microsec_clock::universal_time();
}
while (czas_biezacy – czas_start < czas_trwania);
```

Więcej na:
http://www.boost.org/doc/libs/1_53_0/doc/html/date_time/posix_time.html

std::chrono



C++11

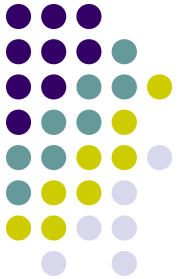
Dostępne zegary

```
#include <chrono>

using namespace std::chrono;
```

Zegar	Opis
system_clock	Wall clock time from the system-wide realtime clock
steady_clock	Monotonic clock that will never be adjusted
high_resolution_clock	The clock with the shortest tick period available

Metody *now*() ww. klas zwracają obiekt daty/czasu typu *std::chrono::time_point*.

time_point / duration – przykłady



```cpp
#include <iostream>
#include <chrono>
#include <ctime>

int main()
{
    std::chrono::time_point<std::chrono::system_clock> start, end;
    start = std::chrono::system_clock::now();
    std::cout << "f(42) = " << fibonacci(42) << '\n';
    end = std::chrono::system_clock::now();

    std::chrono::duration<double> elapsed_seconds = end-start;
    std::time_t end_time = std::chrono::system_clock::to_time_t(end);

    std::cout << "finished computation at " << std::ctime(&end_time)
              << "elapsed time: " << elapsed_seconds.count() << "s\n";
}
```

Więcej na:
http://en.cppreference.com/w/cpp/chrono

Programowanie jest fantastyczne!!!