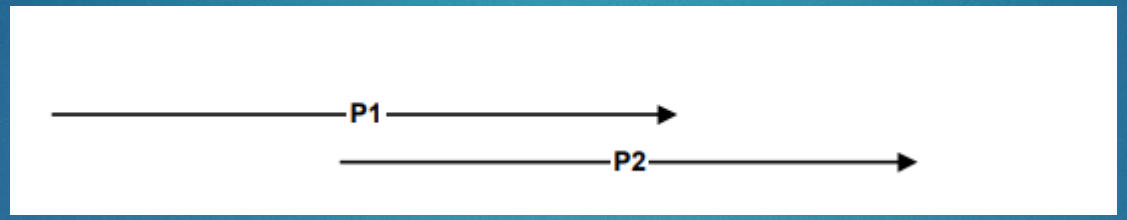
Wielowątkowość

Czym jest współbieżność?

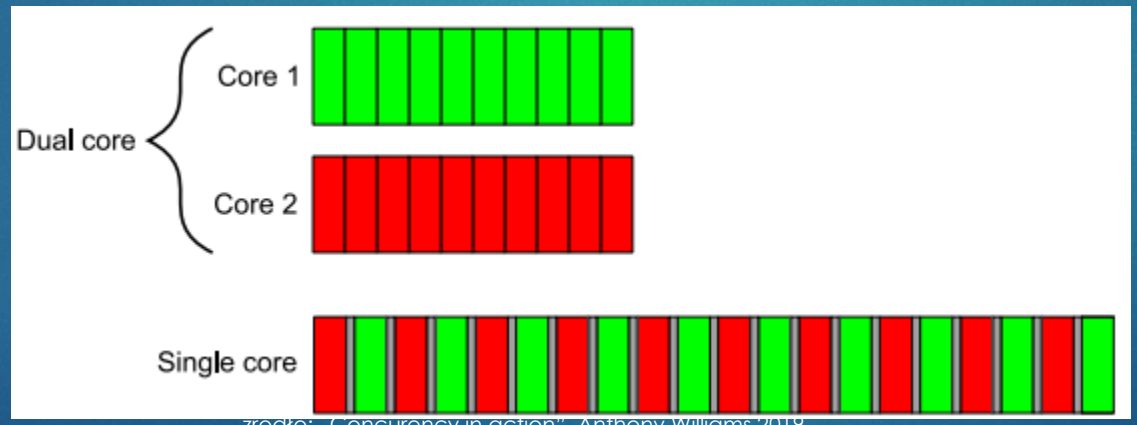
- Dwa procesy są współbieżne jeżeli jeden z nich rozpoczyna się przed zakończeniem drugiego.



- współbieżność polega na jednoczesnym wykonywaniu co najmniej dwóch czynności.

Współbieżność w systemach komputerowych

- „Symulacja” współbieżności w systemach jednoprocesorowych przy wykorzystaniu mechanizmu przełączania zadań (ang. Task switching),
- Komputery/ serwery wieloprocesorowe/ wielordzeniowe obsługujące wiele aplikacji jednoczenie.



źródło: „Concurency in action”, Anthony Williams 2019.

Modele współbieżności

- Współbieżność z wieloma procesami:
  - Osobne procesy (komunikacja przez sygnały, pliki, potoki, gniazda itd.),
  - Osobna przestrzeń pamięci, bezpieczeństwo przetwarzania wielowątkowego,
  - Długi i skomplikowany proces uruchamiania nowego procesu.
- Współbieżność z wieloma wątkami:
  - Uruchamianie wielu wątków w obrębie jednego procesu, które współdzielą pamięć (współdzielona przestrzeń adresowa),
  - Należy samemu zagwarantować bezpieczeństwo dostępu do współdzielonej pamięci,
  - Uruchomienie wątku jest szybsze niż uruchomienie nowego procesu,
  - Prostsze i szybsze metody komunikowania się pomiędzy watkami w obrębie procesu.

Kiedy stosować współbieżność?

- Podział zagadnień (np. odtwarzacz Blu-ray),
- Wydajność:
  - Zrównoleglanie zadań (podział zadania na części),
  - Zrównoleglanie danych (wykonywanie tych samych zadań, na różnych fragmentach danych)

Kiedy nie stosować współbieżności?

- Gdy zbyt wiele wątków może obniżyć wydajność zamiast ja zwiększyć (koszt uruchomienia wątku, koszt zasobów – każdy watek zajmuje pamięć typowo 8MB.),
- Gdy wzrost wydajności nie jest proporcjonalny do woźnego wysiłku i złożoności kodu (koszt utrzymania kodu jest równie istotny).

Proces oraz wątek

- Proces służy do organizowania wykonywania programu. W skład jednego programu wchodzi jeden lub więcej procesów. Zatem proces jest to cały kontekst niezbędny do wykonania programu.
- Zmianie w wyniku wykonywania procesu ulega między innymi segment danych, segment stosu, stan rejestrów procesora.
- W momencie wykonywania procesu system operacyjny przydziela procesowi niezbędne zasoby (pamięć, czas procesora itp.).
- Synchronizacja, sposób obsługi procesów itp. Kontrolowana jest przez system operacyjny.
- W obrębie każdego procesu istnieje jeden lub więcej wątków.
- Wątki tego samego procesu współdzielą większość przestrzeni adresowej (segment kodu i danych, otwarte pliki Itp.).
- Przełączanie kontekstu wątku jest stosunkowo szybki i nie obciążające system operacyjny.
- Tworzenie wątku wymaga mniej zasobów do działania i jest szybsze niż tworzenie procesu.
- Łatwa (ale również niebezpieczna) komunikacja pomiędzy wątkami w obrębie jednego procesu.
- Każdy wątek posiada odrębny stos (adres powrotu z funkcji oraz zmienne lokalne).

C++ i obsługa wielowątkowości

- Standard C++11/14/17/20 – Wprowadzenie i rozwój bibliotek odpowiadających za obsługę wielowątkowości,
- Przed C++11 trzeba było korzystać z różnych bibliotek, lub ręcznie odwoływać się do interfejsów API udostępniających mechanizmy wielowątkowe,
- C++11 wprowadził również nowy model pamięci przystosowany do przetwarzania wielowątkowego na wielu platformach.
- Wprowadzono zarządzanie wątkami (ang. Thread), ochronę współdzielonych danych, synchronizacje operacji wykonywanych przez wątki, wykonywanie niskopoziomowych operacji atomowych itp.

„Hello World”

```
1  #include <iostream>
2  #include <thread>
3
4  void hello() {
5    std::cout << "Hello World"\n";
6  }
7  int main() {
8    std::thread t(hello);
9    t.join();
10 }
11
```

Zarządzanie wątkami

- std::thread,
- std::thread::join(),
- std::thread::detach(),
- std::thread::joinable(),

```
before starting, joinable: false
after starting, joinable: true
after joining, joinable: false
```

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

void foo()
{
    std::this_thread::sleep_for(std::chrono::seconds(1));
}

int main()
{
    std::thread t;
    std::cout << "before starting, joinable: " << std::boolalpha << t.joinable()
              << '\n';

    t = std::thread(foo);
    std::cout << "after starting, joinable: " << t.joinable()
              << '\n';

    t.join();
    std::cout << "after joining, joinable: " << t.joinable()
              << '\n';
}
```

Co przekazać do std::thread?

- Funkcje,
- Funktor (obiekt funkcyjny) – obiekt, który możemy wywołać tak jak funkcję (np. lambda),
- Wskaźnik na funkcje lub wskaźnik do funkcji składowej,
- Obiekt funkcyjny jest **kopiowany** do obszaru

  pamięci należącej do nowo utworzonego wątku.

Co przekazać do std::thread?

```
1  struct Bar {
2      void operator()() {
3          std::cout << "Hello World";
4      }
5  }
6
7  void foo() {
8      std::cout << "Hello World";
9  }
10
11 int main() {
12
13     std::thread t1([]() {
14         "Hello World"
15     });
16
17     std::thread t2(foo);
18
19     Bar bar;
20     std::thread t3(bar);
21
22 }
23
```

```
void foo() {
    std::cout << "Hello World\n";
}

class Bar {
public:
    void foo() { std::cout << "Hello World\n"; };
};

int main() {

    std::thread t(*foo);
    t.join();

    class Bar bar;
    std::thread t1(&Bar::foo, bar);
    t1.join();

    return 0;
}
```

std::thread::detach()

- Zadanie 1:
  - Jakie zagrożenia kryje kod po prawej?
  - Jak poprawić kod?

```
1  #include <iostream>
2  #include <thread>
3
4  void do_something(int data) {
5      std::cout << "Data: " << data << "\n" << std::flush;
6  }
7
8  struct Foo {
9  public:
10     Foo(int& i): i_(i) {}
11     void operator()(){
12         for (int k = 0 ; k < 100000 ; ++k) {
13             do_something(++i_);
14         }
15     }
16
17 private:
18     int& i_;
19 };
20
21 void oops() {
22     int some_local_state = 0;
23     Foo foo(some_local_state);
24     std::thread t(foo);
25     std::cout << "Detach thread\n" << std::flush;
26     t.detach();
27 }
28
29 int main()
30 {
31     std::thread t(oops);
32     t.join();
33
34     return 0;
35 }
```

std::terminate
std::thread::~thread

- If *this has an associated thread (joinable() == true), std::terminate() is called.
- **ZAWSZE** należy zawołać std::join() lub std::detach() na wątku!
- Co z wyjątkami, jakieś pomysły?

std::exception

- Zadanie 2:
  - Jakie zagrożenia kryje listing poniżej?
  - Jak lepiej można rozwiązać poniższy problem?

```
3
4  void do_something(int data) {
5      std::cout << "Data: " << data << "\n" << std::flush;
6  }
7
8  struct Foo {
9  public:
10     Foo(int& i): i_(i) {}
11     void operator()(){
12         for (int k = 0 ; k < 10 ; ++k) {
13             do_something(++i_);
14         }
15     }
16 private:
17     int& i_;
18 };
19
20 void bar() {throw std::runtime_error("Error"); }
21
22 void oops() {
23     int some_local_state = 0;
24     Foo foo(some_local_state);
25     std::thread t(foo);
26     try {
27         bar();
28     } catch (...) {
29         std::cout << "ERROR" << std::endl;
30         t.join();
31         throw;
32     }
33     t.join();
34 }
35
36 int main()
37 {
38     oops();
39     return 0;
40 }
```

RAII (Resource Acquisition Is Initialization)

- Własna Klasa ThreadGuard – zapewniająca bezpieczeństwo złączania wątków.

```
class ThreadGuard
{
    std::thread& t;
public:
    explicit ThreadGuard(std::thread& t_) :
        t(t_)
    {}
    ~ThreadGuard()
    {
        if (t.joinable())
        {
            t.join();
        }
    }
    ThreadGuard(ThreadGuard const&) = delete;
    ThreadGuard& operator=(ThreadGuard const&) = delete;
};
struct func;
void f()
{
    int local = 0;
    Fun fun(local);
    std::thread t(fun);
    ThreadGuard g(t); // lub ThreadGuard(std::thread(fun));
    foo();
}
```

Przekazywanie argumentów do funkcji wątku

- Przekazywanie przez wartość

```
struct SomeStruct {

};

void bar(int x, std::string str, SomeStruct obj) {

}

void main() {
    std::thread t(bar, 10, "String", SomeStruct{});
    t.join();
}
```

Przekazywanie argumentów do funkcji wątku

- Przekazywanie przez wskaźnik, referencje.

```
void bar(int& x, int* y) {
    std::cout << "Inside fun: x = " << x << " | y = " << *y << std::endl;
    x = 20;
    *y = 30;
}

int main() {
    int x = 10;
    int y = 10;
    std::thread t(bar, std::ref(x), &y);
    t.join();
    std::cout << "Outside fun: x = " << x << " | y = " << y << std::endl;

    return 0;
}
```

Pułapki podczas przekazywania argumentów

- Zadanie 3: Jakie pułapki kryje kod poniżej?

```
void f(int i, std::string const& s);
void oops(int arg)
{
    char buffer[1024];
    sprintf(buffer, "%i", arg);
    std::thread t(f, 3, buffer);
    t.detach();
}
```

Pułapki podczas przekazywania argumentów

- Należy zwrócić uwagę na przekazywane argumenty
  - Jeżeli zmienna jest wskaźnikiem/ referencją, należy zadbać aby długość jej życia była dłuższa niż wątku, który na niej operuje.
  - Jeżeli istnieje ryzyko niejawnej konwersji, najlepiej od razu przekazać przekonwertowany argument.

```
void f(int i, std::string const& s);
void not_oops(int param)
{
    char buffer[1024];
    sprintf(buffer, "%i", param);
    std::thread t(f, 3, std::string(buffer));
    t.detach();
}
```

Przenoszenie wątków

- Tak jak np. std::unique_ptr, wątki mogą być jedynie przenoszone (ich kopiowanie nie miało by sensu, gdyż 2 obiekty zarządzałyby jednym wątkiem)
- Do przenoszenia wątków wykorzystujemy bibliotekę utility i funkcję std::move

```
int main() {
    std::thread t1;
    std::thread t2(foo);
    std::thread t3(std::move(t2));
    t1 = std::thread(bar);

    std::cout << std::boolalpha << "t1: " << t1.joinable() << std::endl;
    std::cout << std::boolalpha << "t2: " << t2.joinable() << std::endl;
    std::cout << std::boolalpha << "t3: " << t3.joinable() << std::endl;

    return 0;
}
```

Wybór liczby wątków podczas implementacji

- Zbyt dużo wątków – program działa wolniej,
- Zbyt mało wątków – brak wykorzystania potencjału,
- std::thread::hardware_concurrency()

Identyfikacja wątków

- std::this_thread::get_id()
- std::thread::id()
- Można porównywać id, można je wyświetlać poprzez std::cout
- Możemy wykorzystać je do identyfikacji poszczególnych wątków

```
1  std::thread::id master_thread;
2  void some_core_part_of_algorithm() {
3    if (std::this_thread::get_id() == master_thread) {
4      do_master_thread_work();
5    }
6    do_common_work();
7  }
```

Usypianie wątków

- std::this_thread::sleep_until
- std::this_thread::sleep_for

- std::chrono
- std::chrono_literals

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

int main()
{
    using namespace std::chrono_literals;
    std::cout << "Hello waiter\n" << std::flush;
    auto start = std::chrono::high_resolution_clock::now();
    std::this_thread::sleep_for(2s);
    auto end = std::chrono::high_resolution_clock::now();
    std::chrono::duration<double, std::milli> elapsed = end-start;
    std::cout << "Waited " << elapsed.count() << " ms\n";
}
```

Possible output:

```
Hello waiter
Waited 2000.12 ms
```

Zadanie 4

- Zaimplementuj program w którym 4 wątki będą wyświetlać swój std::this_thread::id() określoną liczbę razy.

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

using namespace std::chrono_literals;

template <typename TIME>
void deamon(int number, TIME time) {
    for (int i = 0; i < number; ++i) {
        std::cout << "Hi I'm thread with id: " << std::this_thread::get_id() << " Number: " << number << std::endl;
        std::this_thread::sleep_for(time);
    }
}

int main() {
    std::thread t1(deamon<decltype(1s)>, 20, 1s);
    std::thread t2(deamon<decltype(1500ms)>, 15, 1500ms);
    std::thread t3(deamon<decltype(2700ms)>, 10, 2700ms);

    t1.detach();
    t2.detach();
    t3.detach();

    deamon(4, 7s);
    return 0;
}
```

Zadanie 5

- Zaimplementuj algorytm std::accumulate wykorzystując wielowątkowość.

Defined in header <numeric>

```
template< class InputIt, class T >
T accumulate( InputIt first, InputIt last, T init );
```

```
template<class InputIt, class T>
constexpr // since C++20
T accumulate(InputIt first, InputIt last, T init)
{
    for (; first != last; ++first) {
        init = std::move(init) + *first; // std::move since C++20
    }
    return init;
}
```

Wskazówka 1

- Obliczenie wymaganych wątków
- Stworzenie kontenera przechowującego wyniki

```
const size_t hardwareThread = std::thread::hardware_concurrency();
const size_t neededThreads = std::min(size / minimumSize, hardwareThread);
const size_t chunkSize = size / neededThreads;
std::cout << "NeededThreads: " << neededThreads << std::endl;
std::cout << "ChunkSIze: " << chunkSize << std::endl;
std::vector<std::thread> threads(neededThreads - 1);
std::vector<T> results(neededThreads);
```

Wskazówka 2

- Implementacja funktora dla wątków
  - Funktor przyjmuje kolejne porcje danych
  - Należy obliczyć i przekazać mu iteratory odnoszące się do początku i końca zakresu na którym ma operować wątek.
  - Należy przekazać także przez referencje zmienną przechowującą wynik.

```
1  std::thread([](IT first, IT last, T& result)
2              {
3                  result = std::accumulate(first, last, T{});
4              }, begin, end, std::ref(results[i]));
```

Wskazówka 3

- Należy wykorzystać również aktualny wątek, na którym wywoływana jest funkcja, aby nie czekał bezczynnie na wywołanie pozostałych wątków.

```
auto begin = first;
for (size_t i = 0; i < neededThreads - 1; ++i) {
    auto end = std::next(begin, chunkSize);
    threads[i] = std::thread([](IT first, IT last, T& result)
        {
            result = std::accumulate(first, last, T{});
        }, begin, end, std::ref(results[i]));
    begin = end;
}
results[neededThreads - 1] = std::accumulate(begin, last, T{});
```

Czy zadanie było trudne?

- Co sprawiło największy problem?
- Czy można uprościć algorytm?
- Czy można zastosować inne mechanizmy ułatwiające implementacje?

Zadanie domowe

- 1) Zaimplementuj algorytm count_if wykorzystując wielowątkowość.
- POWODZENIA ☺

Dziękuję za uwagę