Programowanie Współbieżne w C++

(Praktyczne Aspekty Inżynierii Oprogramowania)

Sławek Zborowski
Bartek 'BaSz' Szurgot

autorzy:
Bartek 'BaSz' Szurgot
Damian Bogel

Nokia Wrocław

26 kwietnia 2016

Część 1

Założenia

- Umiejętność programowania
- Znajomość C++
- Podstawy C++14
- Zdrowy rozsądek
- Otwarty umysł. . . :)

http://images.techhive.com/images/idge/imported/imageapi/2014/03/child-with-computer-100257326-primary.idge.jpg

Prawo Moore’a

Prawo Moore'a

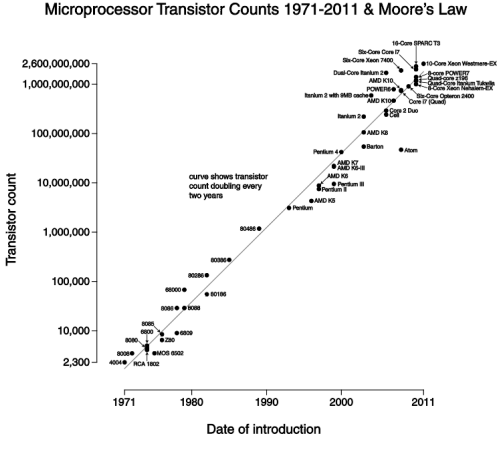
Prawo Moore'a

Liczba tranzystorów w procesorze podwaja się co około dwa lata.

Prawo Moore'a

Prawo Moore'a

Liczba tranzystorów w procesorze podwaja się co około dwa lata.



http://upload.wikimedia.org/wikipedia/commons/0/00/Transistor_Count_and_Moore's_Law_-_2011.svg

Programowanie sekwencyjne

- Na początek klasycznie :)

Programowanie sekwencyjne

- Na początek klasycznie :)
- Wyliczanie wartości funkcji:

```
1  const auto range = 10.5;
2  const auto eps   = 0.001;
3  for(double x = -range; x <= +range; x += eps)
4    cout << "f(" << x << ")␣=␣" << complicatedFunction(x) << endl;
```

Programowanie sekwencyjne

- Na początek klasycznie :)
- Wyliczanie wartości funkcji:

```
1  const auto range = 10.5;
2  const auto eps   = 0.001;
3  for(double x = -range; x <= +range; x += eps)
4    cout << "f(" << x << ")␣=␣" << complicatedFunction(x) << endl;
```

- Ile potrwa wykonywanie programu?
- $t = \frac{2range+1}{eps} t_{func}$
- Gdzie t_{func} = czas wykonania `complicatedFunction(x)`

Programowanie sekwencyjne

- Na początek klasycznie :)
- Wyliczanie wartości funkcji:

```
1 const auto range = 10.5;
2 const auto eps   = 0.001;
3 for(double x = -range; x <= +range; x += eps)
4   cout << "f(" << x << ") = " << complicatedFunction(x) << endl;
```

- Ile potrwa wykonywanie programu?
- $t = \frac{2range+1}{eps} t_{func}$
- Gdzie t_{func} = czas wykonania `complicatedFunction(x)`
- Ile trwa t_{func}?
  - 1 mikrosekundę? ($t = 22ms$)
  - 1 sekundę? ($t > 6h$)
- t_{func} będzie maleć z czasem?

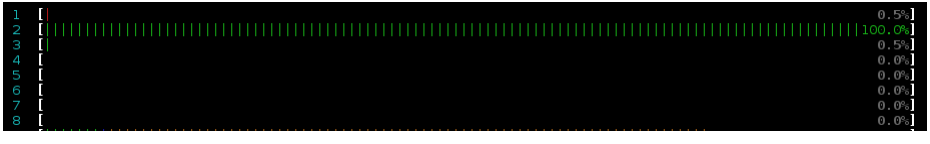
Podgląd wykonania

- Rok 2005:
  - $t_{func} = 100ms$
  - Obciążenie systemu:

Podgląd wykonania

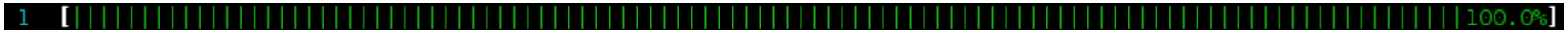
- Rok 2005:
  - $t_{func} = 100ms$
  - Obciążenie systemu:

- Rok 2015:
  - $t_{func} = 70ms$
  - Obciążenie systemu:

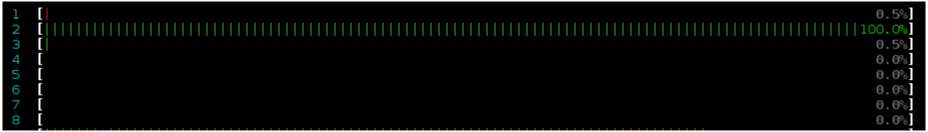


Podgląd wykonania

- Rok 2005:
  - $t_{func} = 100ms$
  - Obciążenie systemu:

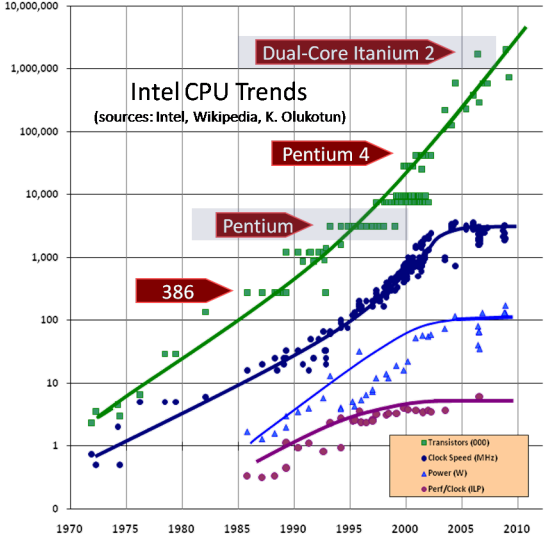


- Rok 2015:
  - $t_{func} = 70ms$
  - Obciążenie systemu:



- Jakie wnioski?
  - Przybyło rdzeni
  - $\frac{7}{8}$ rdzeni (87.5%) nie robi **nic**!
  - Ogromne rezerwy mocy do wykorzystania!

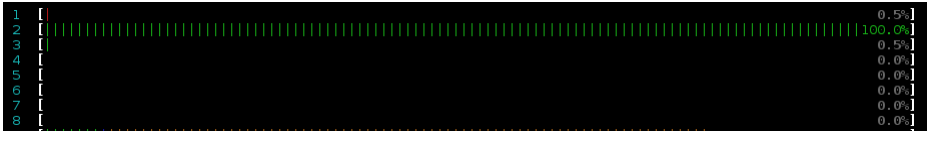
Prawo Moore'a w praktyce



http://www.gotw.ca/images/CPU.png

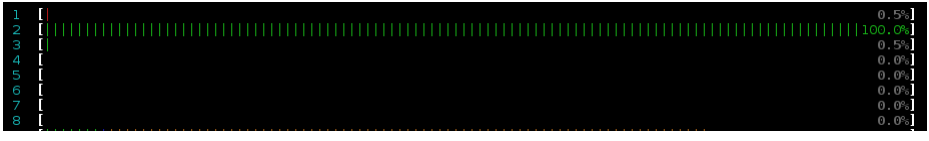
Problem

- Obecne obciążenie procesora:

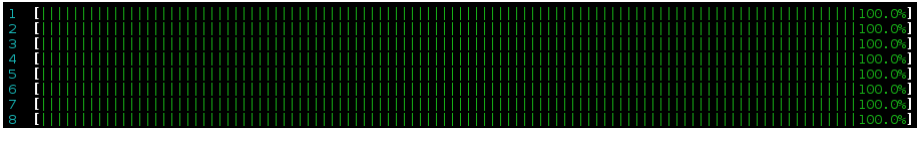


Problem

- Obecne obciążenie procesora:



- Oczekiwane obciążenie procesora:



- Jak ów cel osiągnąć?
- Napisać program używający wszystkich rdzeni jednocześnie!

Skrótowo o wątkach

- Wątek (ang. *thread*) vs. proces (ang. *process*)
- Proces może mieć wiele wątków
- Wątek – aka. „lekki proces”

Skrótowo o wątkach

- Wątek (ang. *thread*) vs. proces (ang. *process*)
- Proces może mieć wiele wątków
- Wątek – aka. „lekki proces”
- Pamięć:
  - Wspólna przestrzeń adresowa
  - Stos (ang. *stack*) – per wątek
  - Sterta (ang. *heap*) – współdzielona

Skrótowo o wątkach

- Wątek (ang. *thread*) vs. proces (ang. *process*)
- Proces może mieć wiele wątków
- Wątek – aka. „lekki proces”
- Pamięć:
  - Wspólna przestrzeń adresowa
  - Stos (ang. *stack*) – per wątek
  - Sterta (ang. *heap*) – współdzielona
- Czas procesora:
  - Przeznaczony dla wątku
  - Wątek per rdzeń
  - Podlega wywłaszczaniu (ang. *preemption*)
- Współbieżność

Dlaczego wątki?

1. Maksymalne wykorzystanie krzemu
   - Szybsze obliczenia
   - Oszczędność prądu
     - Szybciej uzyskiwany wynik
     - Przejście w tryb uśpienia
   - Szybka komunikacja
     - Wspólna przestrzeń adresowa wątków
     - Przekazywanie wskaźników
     - Brak konieczności IPC

Dlaczego wątki?

1. Maksymalne wykorzystanie krzemu
   - Szybsze obliczenia
   - Oszczędność prądu
     - Szybciej uzyskiwany wynik
     - Przejście w tryb uśpienia
   - Szybka komunikacja
     - Wspólna przestrzeń adresowa wątków
     - Przekazywanie wskaźników
     - Brak konieczności IPC
2. Minimalizacja opóźnień
   - Wolne I/O w osobnym wątku
   - Responsywność (GUI)

Część 2

Nim zaczniemy...

https://github.com/nokia-wroclaw/paro

Budujemy!

```
cmake .
```

Uruchamianie wątku

- Wątek jest obiektem
- `#include <thread>`

Uruchamianie wątku

- Wątek jest obiektem
- `#include <thread>`

```
1 // foo()
2 std::thread t1(&foo);
3
4 // bar(1, 2)
5 std::thread t2(&bar, 1, 2);
6
7 // someObject.method(1, 2, 3)
8 std::thread t3(&SomeClass::method, someObject, 1, 2, 3);
```

Uruchamianie wątku

- Wątek jest obiektem
- `#include <thread>`

```
1 // foo()
2 std::thread t1(&foo);
3
4 // bar(1, 2)
5 std::thread t2(&bar, 1, 2);
6
7 // someObject.method(1, 2, 3)
8 std::thread t3(&SomeClass::method, someObject, 1, 2, 3);
```

Uruchamianie wątku

- Wątek jest obiektem
- `#include <thread>`

```
1 // foo()
2 std::thread t1(&foo);
3
4 // bar(1, 2)
5 std::thread t2(&bar, 1, 2);
6
7 // someObject.method(1, 2, 3)
8 std::thread t3(&SomeClass::method, someObject, 1, 2, 3);
```

Zadanie

- Program: `stworzenie_watku.cpp`
- Zadanie:
  1. Utworzyć wątek `greeter`
  2. `make stworzenie_watku`
  3. Uruchomić
  4. Co się dzieje na koniec programu?
- `http://en.cppreference.com/w/cpp/thread/thread`

Co dalej z wątkiem?

- . . . kaboom!
- Co robić?

Co dalej z wątkiem?

- . . . kaboom!
- Co robić?
- Nic?
  1. `std::thread::~thread()`
  2. `std::thread::joinable()` zwróci `true`
  3. `std::terminate()`
- Słabo. . .

Co dalej z wątkiem?

- . . . kaboom!
- Co robić?
- Nic?
  1. `std::thread::~thread()`
  2. `std::thread::joinable()` zwróci `true`
  3. `std::terminate()`
- Słabo. . .
- Poczekać na zakończenie?
- `std::thread::join()`

Co dalej z wątkiem?

- . . . kaboom!
- Co robić?
- Nic?
  1. `std::thread::~thread()`
  2. `std::thread::joinable()` zwróci `true`
  3. `std::terminate()`
- Słabo. . .
- Poczekać na zakończenie?
- `std::thread::join()`
- Odłączyć?
- `std::thread::detach()`
  - Nie czekamy!
  - Koniec `main()` niszczy wątek

Co dalej z wątkiem?

- . . . kaboom!
- Co robić?
- Nic?
  1. `std::thread::~thread()`
  2. `std::thread::joinable()` zwróci `true`
  3. `std::terminate()`
- Słabo. . .
- Poczekać na zakończenie?
- `std::thread::join()`
- Odłączyć?
- `std::thread::detach()`
  - Nie czekamy!
  - Koniec `main()` niszczy wątek

Wniosek

Na każdym wątku trzeba zawołać `join()` albo `detach()`. Zawsze.

Zadanie

- Program: `stworzenie_watku.cpp`
- Zadanie:
  1. Usunąć `std::this_thread::sleep_for()`
  2. Dodać `std::thread::join()`
  3. Sprawdzić wynik
- `http://en.cppreference.com/w/cpp/thread/thread/join`

Co z argumentami?

- Mając `void foo(const std::string&);`
- Znajdź różnicę:
  - `foo(x)`
  - `std::thread(foo, x)`
- Co może być inaczej?

Co z argumentami?

- Mając `void foo(const std::string&);`
- Znajdź różnicę:
  - `foo(x)`
  - `std::thread(foo, x)`
- Co może być inaczej?
- Argumenty:
  - `foo(x)` – referencja
  - `std::thread(foo, x)` – kopia

Co z argumentami?

- Mając `void foo(const std::string&);`
- Znajdź różnicę:
  - `foo(x)`
  - `std::thread(foo, x)`
- Co może być inaczej?
- Argumenty:
  - `foo(x)` – referencja
  - `std::thread(foo, x)` – kopia
- Przekazanie referencji – `std::ref()`

```
1 int x = 5;
2 std::thread t(baz, std::ref(x));
3 t.join();
```

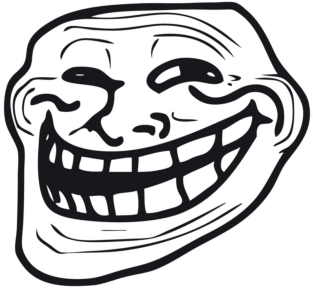
Zadanie

- Program: `argumenty_watku.cpp`
- Zadanie:
  1. `make argumenty_watku`
  2. Uruchom.
  3. Jakie adresy się wyświetlają? Czemu?
  4. Odkomentuj linie na końcu pliku.
  5. Czemu występuje błąd kompilacji?
  6. Napraw błąd i uruchom program.
- `http://en.cppreference.com/w/cpp/utility/functional/ref`

Wartość zwracana. . .

Wartość zwracana...

. . . jest ignorowana! :-)



Wyjątki a wątki

- W ciele wątku:
  - „**`[except.handle]`** If no matching handler is found, the function `std::terminate()` is called (...).”

Wyjątki a wątki

- W ciele wątku:
  - „**[except.handle]** If no matching handler is found, the function `std::terminate()` is called (...).”
  - Wniosek?
  - Trzeba obsługiwać wyjątki w wątkach!

Wyjątki a wątki

- W ciele wątku:
  - „**[except.handle]** If no matching handler is found, the function `std::terminate()` is called (...).”
  - Wniosek?
  - Trzeba obsługiwać wyjątki w wątkach!
- Po uruchomieniu wątku:

```
1 std::thread t(foo);
2 bar(1, 2); // what if bar() throws?
3 t.join();
```

Co z wyjątkami?

- Program: `wyjatki.cpp`
- Zadanie:
  1. `make wyjatki`
  2. Co się dzieje?
  3. ...

Co z wyjątkami?

- Program: `wyjatki.cpp`
- Zadanie:
  1. `make wyjatki`
  2. Co się dzieje?
  3. ...
  4. Naprawić `thread_guard`'em

RAII

Resource Acquisition Is Initialization

Co z wyjątkami?

- Program: `wyjatki.cpp`
- Zadanie:
  1. `make wyjatki`
  2. Co się dzieje?
  3. ...
  4. Naprawić `thread_guard`'em

RAII

Resource Acquisition Is Initialization

```
1  struct thread_guard final
2  {
3    explicit thread_guard(std::thread &t):
4      thread_(t)
5    {}
6
7    ~thread_guard()
8    {
9      if( thread_.joinable() )
10       thread_.join();
11   }
12
13   // ...
14 private:
15   std::thread& thread_;
16 };
```

Jeszcze jeden, mały problem...

```
1  #include <thread>
2  #include <iostream>
3  #include <functional>
4
5  void abc(int &a) { a = 2; }
6  void def(int &a) { a = 3; }
7
8  int main()
9  {
10   int x = 1;
11   std::thread t1(abc, std::ref(x));
12   std::thread t2(def, std::ref(x));
13
14   t1.join();
15   t2.join();
16
17   std::cout << x << std::endl; // what will be displayed?
18 }
```

Jeszcze jeden, mały problem...

```
1  #include <thread>
2  #include <iostream>
3  #include <functional>
4
5  void abc(int &a) { a = 2; }
6  void def(int &a) { a = 3; }
7
8  int main()
9  {
10   int x = 1;
11   std::thread t1(abc, std::ref(x));
12   std::thread t2(def, std::ref(x));
13
14   t1.join();
15   t2.join();
16
17   std::cout << x << std::endl; // what will be displayed?
18 }
```

Jeszcze jeden, mały problem...

```
1  #include <thread>
2  #include <iostream>
3  #include <functional>
4
5  void abc(int &a) { a = 2; }
6  void def(int &a) { a = 3; }
7
8  int main()
9  {
10   int x = 1;
11   std::thread t1(abc, std::ref(x));
12   std::thread t2(def, std::ref(x));
13
14   t1.join();
15   t2.join();
16
17   std::cout << x << std::endl; // what will be displayed?
18 }
```

Jeszcze jeden, mały problem...

```
1  #include <thread>
2  #include <iostream>
3  #include <functional>
4
5  void abc(int &a) { a = 2; }
6  void def(int &a) { a = 3; }
7
8  int main()
9  {
10   int x = 1;
11   std::thread t1(abc, std::ref(x));
12   std::thread t2(def, std::ref(x));
13
14   t1.join();
15   t2.join();
16
17   std::cout << x << std::endl; // what will be displayed?
18 }
```

Jeszcze jeden, mały problem...

```
1  #include <thread>
2  #include <iostream>
3  #include <functional>
4
5  void abc(int &a) { a = 2; }
6  void def(int &a) { a = 3; }
7
8  int main()
9  {
10   int x = 1;
11   std::thread t1(abc, std::ref(x));
12   std::thread t2(def, std::ref(x));
13
14   t1.join();
15   t2.join();
16
17   std::cout << x << std::endl; // what will be displayed?
18 }
```

Jeszcze jeden, mały problem...

```
1  #include <thread>
2  #include <iostream>
3  #include <functional>
4
5  void abc(int &a) { a = 2; }
6  void def(int &a) { a = 3; }
7
8  int main()
9  {
10   int x = 1;
11   std::thread t1(abc, std::ref(x));
12   std::thread t2(def, std::ref(x));
13
14   t1.join();
15   t2.join();
16
17   std::cout << x << std::endl; // what will be displayed?
18 }
```

Krótka przerwa

10 minut

Część 3

Dzielona zmienna - wspólny obszar pamięci

- `char cstr[100+1];`
- Jaki wynik?

```
1 // thread #1
2 strcpy(cstr,
3        "alice␣has␣a␣cat");
```

```
1 // thread #2
2 strcpy(cstr,
3        "bruce␣is␣a␣␣builder");
```

Dzielona zmienna - wspólny obszar pamięci

- `char cstr[100+1];`
- Jaki wynik?

```
1 // thread #1
2 strcpy(cstr,
3        "alice␣has␣a␣cat");
```

```
1 // thread #2
2 strcpy(cstr,
3        "bruce␣is␣a␣␣builder");
```

- Przykładowe wyniki:
  - `cstr = "alice has a cat"`
  - `cstr = "bruce is a  builder"`

Dzielona zmienna - wspólny obszar pamięci

- `char cstr[100+1];`
- Jaki wynik?

```
1 // thread #1
2 strcpy(cstr,
3        "alice has a cat");
```

```
1 // thread #2
2 strcpy(cstr,
3        "bruce is a  builder");
```

- Przykładowe wyniki:
  - `cstr = "alice has a cat"`
  - `cstr = "bruce is a  builder"`
  - ...
  - `cstr = "bruce has a cat"`
  - `cstr = "alice is a  builder"`
  - `cstr = "alice is  a cat"`
  - `cstr = "brice is aa cat"`
  - itd...

Dzielona zmienna - wspólny obszar pamięci

- `char cstr[100+1];`
- Jaki wynik?

```
1 // thread #1
2 strcpy(cstr,
3        "alice has a cat");
```

```
1 // thread #2
2 strcpy(cstr,
3        "bruce is a  builder");
```

- Przykładowe wyniki:
  - `cstr = "alice has a cat"`
  - `cstr = "bruce is a  builder"`
  - ...
  - `cstr = "bruce has a cat"`
  - `cstr = "alice is a  builder"`
  - `cstr = "alice is  a cat"`
  - `cstr = "brice is aa cat"`
  - itd...
- Co z osobnymi zmiennymi?
- Może typy proste?

Dzielona zmienna - typy proste

- `int a,b;`
- Jaki wynik?

```
1 // thread #1
2 a = 1;
3 b = a;
```

```
1 // thread #2
2 b = 2;
3 a = b;
```

Dzielona zmienna - typy proste

- `int a,b;`
- Jaki wynik?

```
1 // thread #1
2 a = 1;
3 b = a;
```

```
1 // thread #2
2 b = 2;
3 a = b;
```

- Możliwości (a,b):
  - „Na logikę”: (1,1), (2,2)

Dzielona zmienna - typy proste

- `int a,b;`
- Jaki wynik?

```
1 // thread #1
2 a = 1;
3 b = a;
```

```
1 // thread #2
2 b = 2;
3 a = b;
```

- Możliwości (a,b):
  - „Na logikę”: (1,1), (2,2)
  - W praktyce także: **(2,1)**, **(1,2)**

Dzielona zmienna - typy proste

- `int a,b;`
- Jaki wynik?

```
1 // thread #1
2 a = 1;
3 b = a;
```

```
1 // thread #2
2 b = 2;
3 a = b;
```

- Możliwości (a,b):
  - „Na logikę”: (1,1), (2,2)
  - W praktyce także: **(2,1)**, **(1,2)**
- Co robić?
- Jak żyć?

Model pamięci w C++14

- Wymagana synchronizacja!
- Model pamięci definiuje kiedy

Model pamięci w C++14

- Wymagana synchronizacja!
- Model pamięci definiuje kiedy
- Założenia:
  - Najmniejsza jednostka - 1 bajt
  - Nie potrzeba synchronizacji dla rozłącznych obszarów

Model pamięci w C++14

- Wymagana synchronizacja!
- Model pamięci definiuje kiedy
- Założenia:
  - Najmniejsza jednostka - 1 bajt
  - Nie potrzeba synchronizacji dla rozłącznych obszarów
- „Wyścig danych” (ang. *„Data race”*) gdy:
  - *N* wątków
  - Dostęp do tej samej lokacji w pamięci
  - W tym samym czasie
  - Co najmniej jeden wątek pisze

Model pamięci w C++14

- Wymagana synchronizacja!
- Model pamięci definiuje kiedy
- Założenia:
  - Najmniejsza jednostka - 1 bajt
  - Nie potrzeba synchronizacji dla rozłącznych obszarów
- „Wyścig danych” (ang. *„Data race”*) gdy:
  - *N* wątków
  - Dostęp do tej samej lokacji w pamięci
  - W tym samym czasie
  - Co najmniej jeden wątek pisze
- Dzięki temu:
  - Odczyty (bez zapisów) – bezpieczne
  - `const` implikuje bezpieczeństwo wielowątkowe

Wyścig danych czy nie?

```
1 // thread #1
2 strcpy(cstr,
3        "alice has a cat");
```

```
1 // thread #2
2 strcpy(cstr,
3        "bruce is a  builder");
```

Wyścig danych czy nie?

```
1 // thread #1
2 strcpy(cstr,
3        "alice has a cat");
```

```
1 // thread #2
2 strcpy(cstr,
3        "bruce is a  builder");
```

- Wyścig na zmiennej `cstr` – 2 wątki piszą
- Wymagana synchronizacja!

Wyścig danych czy nie?

```
1 // thread #1
2 strcpy(cstr,
3        "alice␣has␣a␣cat");
```

```
1 // thread #2
2 strcpy(cstr,
3        "bruce␣is␣a␣␣builder");
```

- Wyścig na zmiennej `cstr` – 2 wątki piszą
- Wymagana synchronizacja!

```
1 // thread #1
2 a = 1;
3 b = a;
```

```
1 // thread #2
2 b = 2;
3 a = b;
```

Wyścig danych czy nie?

```
1 // thread #1
2 strcpy(cstr,
3        "alice has a cat");
```

```
1 // thread #2
2 strcpy(cstr,
3        "bruce is a  builder");
```

- Wyścig na zmiennej `cstr` – 2 wątki piszą
- Wymagana synchronizacja!

```
1 // thread #1
2 a = 1;
3 b = a;
```

```
1 // thread #2
2 b = 2;
3 a = b;
```

- Wyścig na zmiennych a oraz b – 2 piszą i czytają
- Wymagana synchronizacja!

Wyścig danych czy nie?

```
1 // thread #1
2 strcpy(cstr,
3        "alice has a cat");
```

```
1 // thread #2
2 strcpy(cstr,
3        "bruce is a  builder");
```

- Wyścig na zmiennej `cstr` – 2 wątki piszą
- Wymagana synchronizacja!

```
1 // thread #1
2 a = 1;
3 b = a;
```

```
1 // thread #2
2 b = 2;
3 a = b;
```

- Wyścig na zmiennych a oraz b – 2 piszą i czytają
- Wymagana synchronizacja!

```
1 // thread #1
2 a = 1;
```

```
1 // thread #2
2 b = 2;
```

Wyścig danych czy nie?

```
1 // thread #1
2 strcpy(cstr,
3        "alice has a cat");
```

```
1 // thread #2
2 strcpy(cstr,
3        "bruce is a  builder");
```

- Wyścig na zmiennej `cstr` – 2 wątki piszą
- Wymagana synchronizacja!

```
1 // thread #1
2 a = 1;
3 b = a;
```

```
1 // thread #2
2 b = 2;
3 a = b;
```

- Wyścig na zmiennych a oraz b – 2 piszą i czytają
- Wymagana synchronizacja!

```
1 // thread #1
2 a = 1;
```

```
1 // thread #2
2 b = 2;
```

- Brak wyścigu – rozłączne obszary pamięci
- Nie trzeba synchronizować

Wyścig danych czy nie?

```
1 // thread #1
2 sin(x);
```

```
1 // thread #2
2 cos(x);
```

Wyścig danych czy nie?

```
1 // thread #1
2 sin(x);
```

```
1 // thread #2
2 cos(x);
```

- Brak wyścigu – same odczyty
- Nie trzeba synchronizować

Wyścig danych czy nie?

```
1 // thread #1
2 sin(x);
```

```
1 // thread #2
2 cos(x);
```

- Brak wyścigu – same odczyty
- Nie trzeba synchronizować

```
1 // thread #1
2 b = sin(a);
```

```
1 // thread #2
2 x = sin(b);
```

Wyścig danych czy nie?

```
1 // thread #1
2 sin(x);
```

```
1 // thread #2
2 cos(x);
```

- Brak wyścigu – same odczyty
- Nie trzeba synchronizować

```
1 // thread #1
2 b = sin(a);
```

```
1 // thread #2
2 x = sin(b);
```

- Wyścig na zmiennej b – 1 wątek czyta, 1 pisze
- Wymagana synchronizacja!

Wyścig danych czy nie?

```
1 // thread #1
2 sin(x);
```

```
1 // thread #2
2 cos(x);
```

- Brak wyścigu – same odczyty
- Nie trzeba synchronizować

```
1 // thread #1
2 b = sin(a);
```

```
1 // thread #2
2 x = sin(b);
```

- Wyścig na zmiennej b – 1 wątek czyta, 1 pisze
- Wymagana synchronizacja!

```
1 // thread #1
2 if(a) b = 1;
```

```
1 // thread #2
2 if(b) a = 2;
```

- Początkowo $a = b = 0$

Wyścig danych czy nie?

```
1 // thread #1
2 sin(x);
```

```
1 // thread #2
2 cos(x);
```

- Brak wyścigu – same odczyty
- Nie trzeba synchronizować

```
1 // thread #1
2 b = sin(a);
```

```
1 // thread #2
2 x = sin(b);
```

- Wyścig na zmiennej b – 1 wątek czyta, 1 pisze
- Wymagana synchronizacja!

```
1 // thread #1
2 if(a) b = 1;
```

```
1 // thread #2
2 if(b) a = 2;
```

- Początkowo $a = b = 0$
- Brak wyścigu – odczyty i zapisy nie zachodzą jednocześnie
- Nie trzeba synchronizować

Sequential consistency

- Wyścig danych == niezdefiniowane zachowanie
- (ang. *undefined behavior*, UB)

Sequential consistency

- Wyścig danych == niezdefiniowane zachowanie
- (ang. *undefined behavior*, UB)
- Optymalizacje kompilatora:
  - Są dozwolone ;)
  - Nie mogą wprowadzać wyścigu danych
  - Muszą zachować obserwowalne efekty

Sequential consistency

- Wyścig danych == niezdefiniowane zachowanie
- (ang. *undefined behavior*, UB)
- Optymalizacje kompilatora:
  - Są dozwolone ;)
  - Nie mogą wprowadzać wyścigu danych
  - Muszą zachować obserwowalne efekty
- W praktyce oznacza to:
  - Trzeba kompilatorowi wskazać obszary/zmienne dzielone
  - Kompilator ogranicza tam optymalizacje
  - Kompilator wstawia specjalne instrukcje dla procesora

Sequential consistency

- Wyścig danych == niezdefiniowane zachowanie
- (ang. *undefined behavior*, UB)
- Optymalizacje kompilatora:
  - Są dozwolone ;)
  - Nie mogą wprowadzać wyścigu danych
  - Muszą zachować obserwowalne efekty
- W praktyce oznacza to:
  - Trzeba kompilatorowi wskazać obszary/zmienne dzielone
  - Kompilator ogranicza tam optymalizacje
  - Kompilator wstawia specjalne instrukcje dla procesora
- Gwarancja bezpiecznego wykonania
- Możliwość wnioskowania o przepływie sterowania
- Zachowana przyczynowość
- Jak synchronizować?

Mutex

- MUTual EXclusion
- Synchronizacja przez wykluczanie/blokowanie

Mutex

- MUTual EXclusion
- Synchronizacja przez wykluczanie/blokowanie
- Nagłówek `mutex`
- Klasa `std::mutex`
- Metody:
  - `lock()` – nakładanie blokady
  - `unlock()` – zdejmowanie blokady
  - `...`

Mutex

- MUTual EXclusion
- Synchronizacja przez wykluczanie/blokowanie
- Nagłówek `mutex`
- Klasa `std::mutex`
- Metody:
  - `lock()` – nakładanie blokady
  - `unlock()` – zdejmowanie blokady
  - ...
- Przykład użycia:

```
1 m.lock();                    // <-- 1
2 str = "alice␣has␣a␣cat";     // critical section
3 m.unlock();                  // <-- 2
```

Zadanie

- Program: `podstawowa_synchronizacja.cpp`
- Zadanie:
  1. Skompilować
  2. Uruchomić kilkukrotnie
  3. Dlaczego program „losowo” zawodzi?
  4. Naprawić program przy użyciu blokad
- `http://en.cppreference.com/w/cpp/thread/mutex`

Sytuacje problematyczne

```
1  m.lock();                    // <-- 1
2  str = "alice␣has␣a␣cat";     // critical section
3  m.unlock();                  // <-- 2
```

- Co gdy przypisanie zgłosi wyjątek?
- Co gdy kod jest skomplikowany?
- Zawsze `unlock` jest wołany? Na pewno?

Sytuacje problematyczne

```
1 m.lock();                    // <-- 1
2 str = "alice␣has␣a␣cat";     // critical section
3 m.unlock();                  // <-- 2
```

- Co gdy przypisanie zgłosi wyjątek?
- Co gdy kod jest skomplikowany?
- Zawsze `unlock` jest wołany? Na pewno?
- RAII – ang. *Resource Acquisition Is Initialization*
  - Pobieranie zasobu w konstruktorze
  - Zwalnianie zasobu w destruktorze
  - Zarządzanie tylko jednym zasobem
- Jak to przełożyć na język blokad?

Blokady w stylu RAII

- Przykładowa implementacja:

```
1  struct LockGuard final
2  {
3    explicit LockGuard(std::mutex& m): m_(m)
4    { m_.lock(); }
5    ~LockGuard()
6    { m_.unlock(); }
7  private:
8    std::mutex& m_;
9  };
```

Blokady w stylu RAII

- Przykładowa implementacja:

```
1  struct LockGuard final
2  {
3    explicit LockGuard(std::mutex& m): m_(m)
4    { m_.lock(); }
5    ~LockGuard()
6    { m_.unlock(); }
7  private:
8    std::mutex& m_;
9  };
```

- Jest standardowa implementacja :-)
- `std::lock_guard<T>`

Blokady w stylu RAII - przykład

- Oryginalny kawałek kodu:

```
1  m.lock();                    // <-- 1
2  str = "alice␣has␣a␣cat";     // critical section
3  m.unlock();                  // <-- 2
```

Blokady w stylu RAII - przykład

- Oryginalny kawałek kodu:

```
1 m.lock();                    // <-- 1
2 str = "alice␣has␣a␣cat";     // critical section
3 m.unlock();                  // <-- 2
```

- Po zmianie na RAII:

```
1 const std::lock_guard<std::mutex> lock(m); // automatic variable
2 str = "alice␣has␣a␣cat";                   // critical section
```

Blokady w stylu RAII - przykład

- Oryginalny kawałek kodu:

```
1 m.lock();                    // <-- 1
2 str = "alice␣has␣a␣cat";     // critical section
3 m.unlock();                  // <-- 2
```

- Po zmianie na RAII:

```
1 const std::lock_guard<std::mutex> lock(m); // automatic variable
2 str = "alice␣has␣a␣cat";                   // critical section
```

- Oraz wprowadzeniu pomocniczej definicji:

```
1 const Lock lock(m);          // using Lock = std::lock_guard<std::mutex>;
2 str = "alice␣has␣a␣cat";     // critical section
```

Blokady w stylu RAII - przykład

- Oryginalny kawałek kodu:

```
1 m.lock();                    // <-- 1
2 str = "alice␣has␣a␣cat";     // critical section
3 m.unlock();                  // <-- 2
```

- Po zmianie na RAII:

```
1 const std::lock_guard<std::mutex> lock(m); // automatic variable
2 str = "alice␣has␣a␣cat";                   // critical section
```

- Oraz wprowadzeniu pomocniczej definicji:

```
1 const Lock lock(m);          // using Lock = std::lock_guard<std::mutex>;
2 str = "alice␣has␣a␣cat";     // critical section
```

- Zalety:
  - Mniej kodu
  - Gwarantowana poprawność
- Zawsze używaj RAII do blokad!

Zadanie

- Program: `podstawowa_synchronizacja.cpp`
- Zadanie:
  1. Skompilować poprawioną wersję
  2. Wprowadzić blokowanie w stylu RAII
- `http://en.cppreference.com/w/cpp/thread/lock_guard`

Oczekiwanie na zmianę

- Jak przekazać dane do wątku?

```
1 // thread #1
2 assert(not ready);
3 str = "important␣message";
4 Lock lock(m);
5 ready = true;
```

```
1 // thread #2
2 while(true)
3 {
4   Lock lock(m);
5   if(ready)
6     break;
7 }
8 cout << str << endl;
```

- Jakieś problemy?

Oczekiwanie na zmianę

- Jak przekazać dane do wątku?

```
1 // thread #1
2 assert(not ready);
3 str = "important␣message";
4 Lock lock(m);
5 ready = true;
```

```
1 // thread #2
2 while(true)
3 {
4   Lock lock(m);
5   if(ready)
6     break;
7 }
8 cout << str << endl;
```

- Jakieś problemy?
- Fatalna wydajność!
- Jeden procesor ciągle w użyciu
- Jak zaczekać na informację?

Zmienna warunkowa

- Jeden wątek czeka na zajście warunku
- Drugi wątek ustawia warunek
- `std::condition_variable cv;`

Zmienna warunkowa

- Jeden wątek czeka na zajście warunku
- Drugi wątek ustawia warunek
- `std::condition_variable cv;`

```
1 // thread #1
2 assert(not ready);
3 Lock lock(m);
4 str = "important_message";
5 ready = true;
6 cv.notify_one();
```

```
1 // thread #2
2 // using UniqueLock =
3 //       std::unique_lock<std::mutex>;
4 UniqueLock lock(m);
5 // blocks (note: spurious wakeups)
6 cv.wait(lock, [&](){ return ready; } );
7 cout << str << endl;
```

Zadanie

- Program: `przekazywanie_przez_kolejke.cpp`
- Zadanie:
  1. Skomunikować dwa wątki za pośrednictwem kolejki
  2. Zapewnić brak wyścigów danych
  3. Nie używać aktywnego czekania (ang. *busy loop*)
- `http://en.cppreference.com/w/cpp/thread/condition_variable`
- Dla ambitnych:
  - Przerobić kolejkę na szablon (ang. *template*)
  - Przygotować bezpieczne API: `push(T)`, `pop()`, `T& top()`

Krótka przerwa

10 minut

Część 4

A co gdybyśmy...

- Robienie kilku rzeczy jednocześnie?

A co gdybyśmy...

- Robienie kilku rzeczy jednocześnie?
- Misja – obiad:
  1. Ugotować ziemniaki – 10 minut
  2. Pokroić kurczaka – 5 minut
  3. Usmażyć kurczaka – 10 minut
  4. Zrobić sałatkę – 15 minut
  5. Zaparzyć herbatę – 10 minut

A co gdybyśmy...

- Robienie kilku rzeczy jednocześnie?
- Misja – obiad:
  1. Ugotować ziemniaki – 10 minut
  2. Pokroić kurczaka – 5 minut
  3. Usmażyć kurczaka – 10 minut
  4. Zrobić sałatkę – 15 minut
  5. Zaparzyć herbatę – 10 minut
- 50 minut?!
- A gdyby tak nie czekać?
- . . .

Zmiana frontu

- Załóżmy, że:
  - Zadanie ma pod-zadania
  - Znamy zależności między pod-zadaniami

Zmiana frontu

- Załóżmy, że:
  - Zadanie ma pod-zadania
  - Znamy zależności między pod-zadaniami
- Niektóre zadania uruchamiamy od razu:
  - Zależności już spełnione
  - Wynik potrzebny później

Zmiana frontu

- Załóżmy, że:
  - Zadanie ma pod-zadania
  - Znamy zależności między pod-zadaniami
- Niektóre zadania uruchamiamy od razu:
  - Zależności już spełnione
  - Wynik potrzebny później
- Przykład:
  - Wczytujemy plik z dysku
  - Używamy później, w trakcie działania
  - Czytanie nie musi blokować innych zadań!

Poznajcie std::async()!

```
1  #include <iostream>
2  #include <future>
3
4  std::string readConfig(std::string const& filename);
5
6  int main()
7  {
8    const auto filename = "config.txt";
9    std::future<std::string> future =
10     std::async(std::launch::async, readConfig, filename);
11   // time to do sth else here!
12   std::cout << "config:␣" << future.get() << std::endl;
13 }
```

Poznajcie std::async()!

```
1  #include <iostream>
2  #include <future>
3
4  std::string readConfig(std::string const& filename);
5
6  int main()
7  {
8    const auto filename = "config.txt";
9    std::future<std::string> future =
10     std::async(std::launch::async, readConfig, filename);
11   // time to do sth else here!
12   std::cout << "config: " << future.get() << std::endl;
13 }
```

Poznajcie std::async()!

```
1  #include <iostream>
2  #include <future>
3
4  std::string readConfig(std::string const& filename);
5
6  int main()
7  {
8    const auto filename = "config.txt";
9    std::future<std::string> future =
10     std::async(std::launch::async, readConfig, filename);
11   // time to do sth else here!
12   std::cout << "config: " << future.get() << std::endl;
13 }
```

Poznajcie std::async()!

```
1  #include <iostream>
2  #include <future>
3
4  std::string readConfig(std::string const& filename);
5
6  int main()
7  {
8    const auto filename = "config.txt";
9    std::future<std::string> future =
10     std::async(std::launch::async, readConfig, filename);
11   // time to do sth else here!
12   std::cout << "config: " << future.get() << std::endl;
13 }
```

Poznajcie std::async()!

```
1  #include <iostream>
2  #include <future>
3
4  std::string readConfig(std::string const& filename);
5
6  int main()
7  {
8    const auto filename = "config.txt";
9    std::future<std::string> future =
10     std::async(std::launch::async, readConfig, filename);
11   // time to do sth else here!
12   std::cout << "config: " << future.get() << std::endl;
13 }
```

Poznajcie std::async()!

```
1  #include <iostream>
2  #include <future>
3
4  std::string readConfig(std::string const& filename);
5
6  int main()
7  {
8    const auto filename = "config.txt";
9    std::future<std::string> future =
10     std::async(std::launch::async, readConfig, filename);
11   // time to do sth else here!
12   std::cout << "config:␣" << future.get() << std::endl;
13 }
```

Zadanie

- Program: `obiad.cpp`
- Zadanie: ugotować obiad :-)
- Pomocne funkcje:
  - `std::future::get()`
  - `auto f = std::async(std::launch::async, funkcja, arg1, arg2);`
- `http://en.cppreference.com/w/cpp/thread/async`
- `http://en.cppreference.com/w/cpp/thread/launch`
- `http://en.cppreference.com/w/cpp/thread/future`

std::async() - polityki uruchomienia

1. `std::launch::async`
   - Natychmiastowe uruchomienie
   - Inny (nowy?) wątek

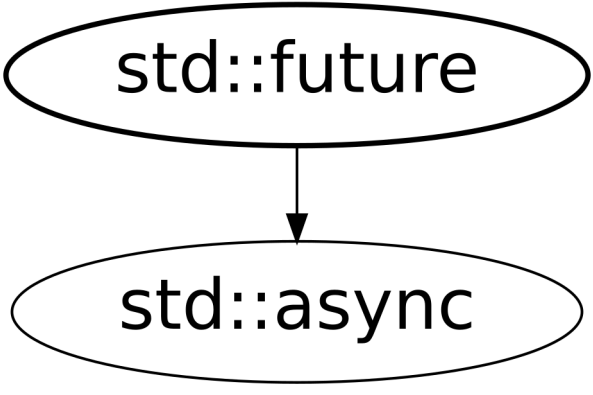
std::async() - polityki uruchomienia

1. `std::launch::async`
   - Natychmiastowe uruchomienie
   - Inny (nowy?) wątek
2. `std::launch::deferred`
   - Leniwa ewaluacja
   - Obliczanie przy okazji:
     - `std::future::wait()`
     - `std::future::get()`
   - Nie tworzy wątku

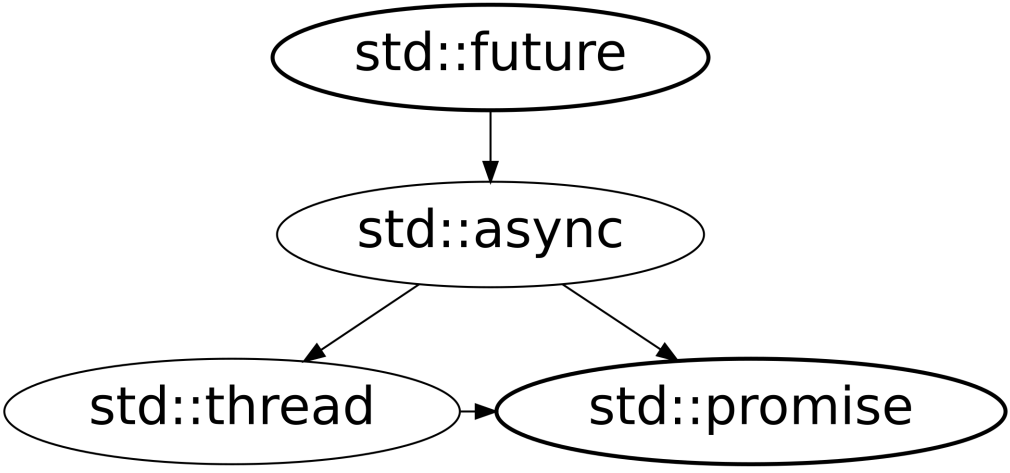
std::async() - polityki uruchomienia

1. `std::launch::async`
   - Natychmiastowe uruchomienie
   - Inny (nowy?) wątek
2. `std::launch::deferred`
   - Leniwa ewaluacja
   - Obliczanie przy okazji:
     - `std::future::wait()`
     - `std::future::get()`
   - Nie tworzy wątku
3. `std::launch::async | std::launch:deferred`
   - Domyślna wartość
   - „Implementacja zadecyduje”

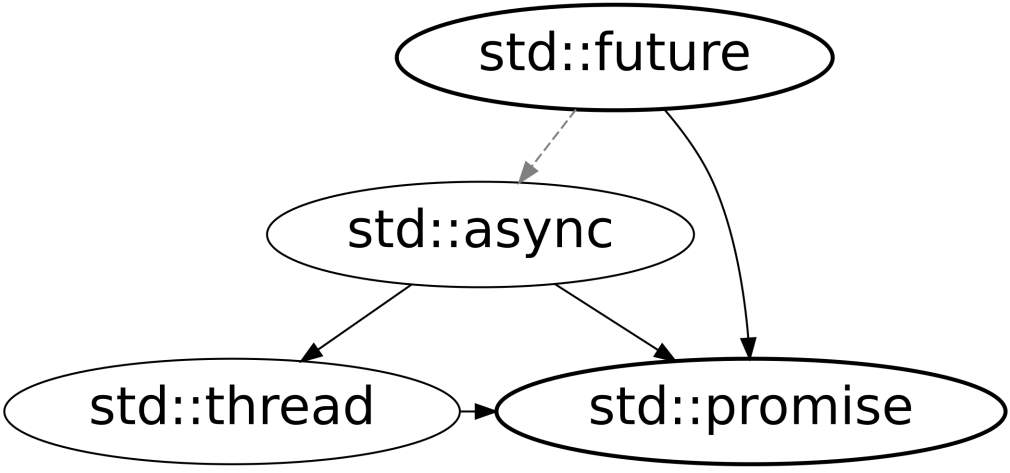
Jak działa std::async?



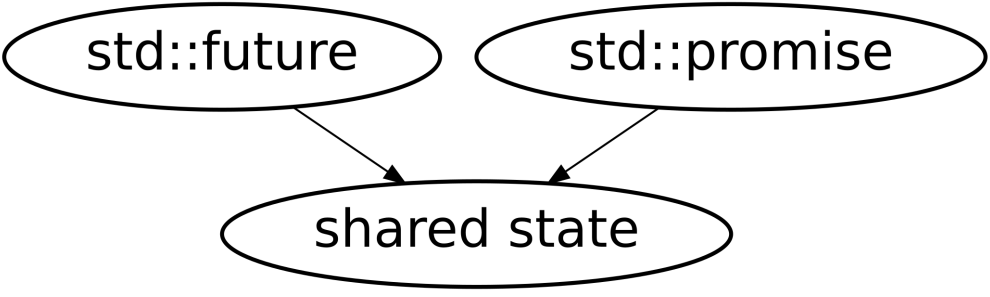
future, promise i async



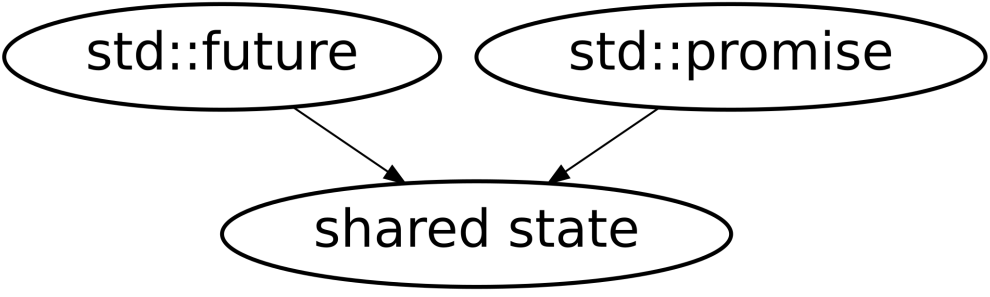
future, promise i async



std::future i std::promise

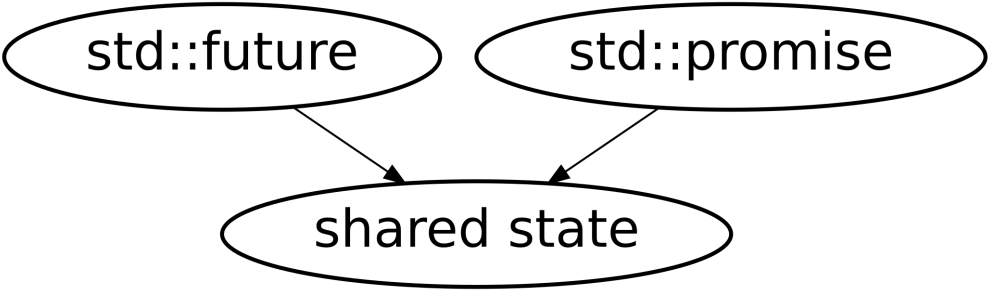


std::future i std::promise



- `std::promise` – nadawca
- `std::promise::set_value()`
- `std::promise::set_exception()`

std::future i std::promise



- `std::promise` – nadawca
- `std::promise::set_value()`
- `std::promise::set_exception()`

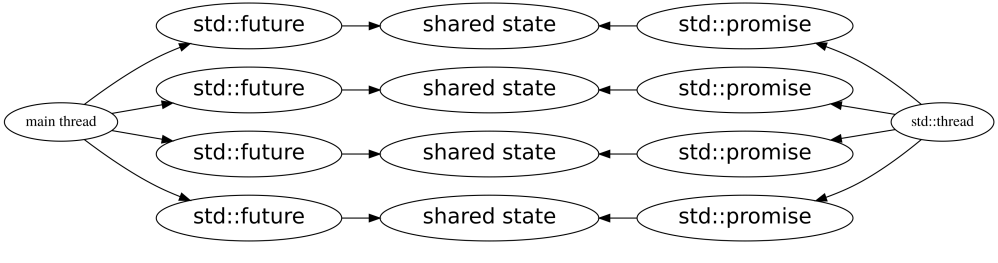
- `std::future` – odbiorca
- `std::future::get()`
- Może zablokować

Dużo obietnic

- Jak zrobić więcej rzeczy?

Dużo obietnic

- Jak zrobić więcej rzeczy?
  - Podać listę zadań
  - Oczekiwać wyników
  - Wątek przetwarza w tle
- Wizualnie:



Zadanie

- Program: `future_promise.cpp`
- Zadanie: Jeden wątek wypełniający dużo `std::promise`
  1. Utwórz wątek
  2. Przekaż do niego kolekcję `std::promise`
  3. Zachowaj powiązane z nimi `std::future`
  4. Wątek roboczy wypełnia `std::promise`
  5. Głównym wątek pyta o wyniki (`std::future::wait_for()`)

Zadanie

- Program: `future_promise.cpp`
- Zadanie: Jeden wątek wypełniający dużo `std::promise`
  1. Utwórz wątek
  2. Przekaż do niego kolekcję `std::promise`
  3. Zachowaj powiązane z nimi `std::future`
  4. Wątek roboczy wypełnia `std::promise`
  5. Głównym wątek pyta o wyniki (`std::future::wait_for()`)
- Wskazówki:
  1. `std::promise::get_future()` – pobiera future powiązanego z promise
  2. `std::promise` – nie kopiowalne, ale przenaszalne (`std::move`)
  3. `std::future::wait_for(std::chrono::milliseconds(0))` zwraca:
     - `std::future_status::ready` – znaczy gotowe!
     - Lub inną...;-)
  4. `std::promise::set_value()` – ustawia wartość
- `http://en.cppreference.com/w/cpp/thread/promise`
- `http://en.cppreference.com/w/cpp/thread/future`

Część 5

Więcej C++14!

- `std::lock()`
  - Zakładanie wielu blokad, bez zakleszczeń
  - `http://en.cppreference.com/w/cpp/thread/lock`

Więcej C++14!

- `std::lock()`
  - Zakładanie wielu blokad, bez zakleszczeń
  - `http://en.cppreference.com/w/cpp/thread/lock`
- `std::atomic<T>`
  - Synchronizacja bez „mutexów”
  - `http://en.cppreference.com/w/cpp/atomic/atomic`

Więcej C++14!

- `std::lock()`
  - Zakładanie wielu blokad, bez zakleszczeń
  - `http://en.cppreference.com/w/cpp/thread/lock`
- `std::atomic<T>`
  - Synchronizacja bez „mutexów”
  - `http://en.cppreference.com/w/cpp/atomic/atomic`
- `std::call_once()`
  - Jednorazowa inicjalizacja
  - `http://en.cppreference.com/w/cpp/thread/call_once`

Więcej C++14!

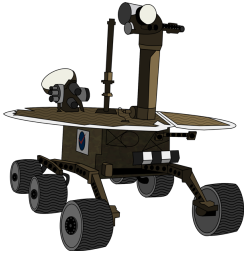
- `std::lock()`
  - Zakładanie wielu blokad, bez zakleszczeń
  - `http://en.cppreference.com/w/cpp/thread/lock`
- `std::atomic<T>`
  - Synchronizacja bez „mutexów”
  - `http://en.cppreference.com/w/cpp/atomic/atomic`
- `std::call_once()`
  - Jednorazowa inicjalizacja
  - `http://en.cppreference.com/w/cpp/thread/call_once`
- `std::shared_lock()`
  - „Czytelnicy i pisarze”
  - `http://en.cppreference.com/w/cpp/thread/shared_lock`

Problemy

- Zakleszczenie (ang. *deadlock*)
  - Ucztujący filozofowie
  - `https://en.wikipedia.org/wiki/Deadlock`

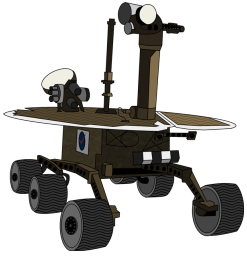
Problemy

- Zakleszczenie (ang. *deadlock*)
  - Ucztujący filozofowie
  - `https://en.wikipedia.org/wiki/Deadlock`
- Odwrócenie priorytetów (ang. *priority inversion*)
  - Marsjański łazik ;-)
  - `https://en.wikipedia.org/wiki/Priority_inversion`



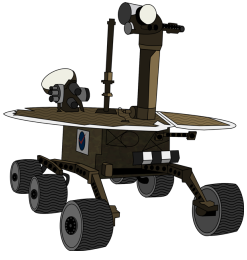
Problemy

- Zakleszczenie (ang. *deadlock*)
  - Ucztujący filozofowie
  - `https://en.wikipedia.org/wiki/Deadlock`
- Odwrócenie priorytetów (ang. *priority inversion*)
  - Marsjański łazik ;-)
  - `https://en.wikipedia.org/wiki/Priority_inversion`
- Aktywne zakleszczenie (ang. *livelock*)
  - Mijanie ludzi idących naprzeciw
  - `https://en.wikipedia.org/wiki/Deadlock#Livelock`



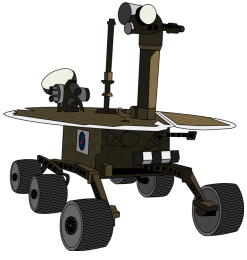
Problemy

- Zakleszczenie (ang. *deadlock*)
  - Ucztujący filozofowie
  - `https://en.wikipedia.org/wiki/Deadlock`
- Odwrócenie priorytetów (ang. *priority inversion*)
  - Marsjański łazik ;-)
  - `https://en.wikipedia.org/wiki/Priority_inversion`
- Aktywne zakleszczenie (ang. *livelock*)
  - Mijanie ludzi idących naprzeciw
  - `https://en.wikipedia.org/wiki/Deadlock#Livelock`
- Zagłodzenie (ang. *resource starvation*)
  - Niepoprawny algorytm szeregujący
  - `https://en.wikipedia.org/wiki/Resource_starvation`



Problemy

- Zakleszczenie (ang. *deadlock*)
  - Ucztujący filozofowie
  - `https://en.wikipedia.org/wiki/Deadlock`
- Odwrócenie priorytetów (ang. *priority inversion*)
  - Marsjański łazik ;-)
  - `https://en.wikipedia.org/wiki/Priority_inversion`
- Aktywne zakleszczenie (ang. *livelock*)
  - Mijanie ludzi idących naprzeciw
  - `https://en.wikipedia.org/wiki/Deadlock#Livelock`
- Zagłodzenie (ang. *resource starvation*)
  - Niepoprawny algorytm szeregujący
  - `https://en.wikipedia.org/wiki/Resource_starvation`
- Fałszywe współdzielenie (ang. *false sharing*)
  - Zależności na poziomie sprzętu
  - `https://en.wikipedia.org/wiki/False_sharing`



Inne zagadnienia

- Wzorce projektowe (ang. *design patterns*)

Inne zagadnienia

- Wzorce projektowe (ang. *design patterns*)
- Programowanie bez blokad (ang. *lock-free programming*)
  - Synchronizacja bez blokowania
  - Skomplikowane nawet dla ekspertów!
  - „Żonglowanie brzytwami”

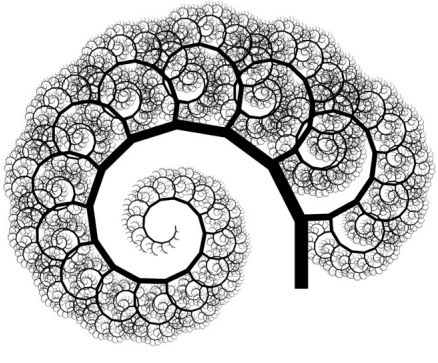
Inne zagadnienia

- Wzorce projektowe (ang. *design patterns*)
- Programowanie bez blokad (ang. *lock-free programming*)
  - Synchronizacja bez blokowania
  - Skomplikowane nawet dla ekspertów!
  - „Żonglowanie brzytwami”
- RCU (ang. *read copy update*)
  - Inne podejście do minimalizacji blokowania
  - Technika wykorzystywana w jądrze linuxa
- . . .
- . . .

Część 6

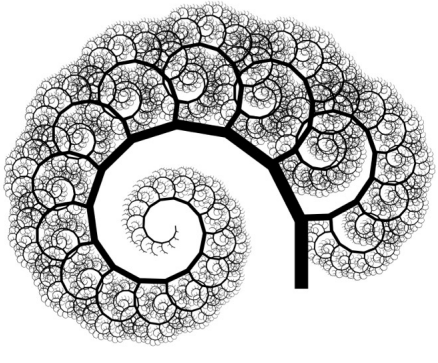
O programowaniu współbieżnym

- Wątki
  - Temat BARDZO szeroki
  - Dużo wiedzy (oprogramowanie + sprzęt)
  - Wymaga praktyki



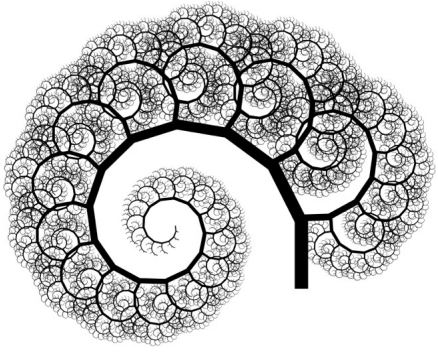
O programowaniu współbieżnym

- Wątki
  - Temat BARDZO szeroki
  - Dużo wiedzy (oprogramowanie + sprzęt)
  - Wymaga praktyki
- 3*h* - mission impossible!
  - Dużo książek o wątkach
  - Podstawa wielu doktoratów
  - Dobry temat na 2-semestralny kurs



O programowaniu współbieżnym

- Wątki
  - Temat BARDZO szeroki
  - Dużo wiedzy (oprogramowanie + sprzęt)
  - Wymaga praktyki
- 3*h* - mission impossible!
  - Dużo książek o wątkach
  - Podstawa wielu doktoratów
  - Dobry temat na 2-semestralny kurs
- Skromne wprowadzenie
- Niezbędne minimum teorii
- Kilka podstawowych mechanizmów



http://matthewjamestaylor.com/img/recursive-drawing/pubic-hair.jpg

Materiały dodatkowe

- „The free lunch is over”, *Herb Sutter*
- „Język C++ i przetwarzanie współbieżne w akcji”, *Anthony Williams*
- „Threads and shared variables in C++11”, *Hans Boehm*
- „Atomic<> weapons”, *Herb Sutter*
- „Eliminate false sharing”, *Herb Sutter*
- „Threading: dos and don’ts”, *Bartek ’BaSz’ Szurgot*
- „Lock-free programming”, *Herb Sutter*
- „Effective modern C++”, *Scott Meyers*
- `http://cppreference.com`

Pytania?