

Programowanie wielowątkowe w C++

Trener: Krystian Piękoś

Informacje organizacyjne

- Czas trwania szkolenia:
 - + 9:00 - 16:00
- Materiały szkoleniowe:
 - + <https://infotraining.bitbucket.io/cpp-thd>
 - + repozytorium GIT
- Przerwy
- Lista obecności
- Ankieta

Parallel vs. Concurrent

Co to jest współbieżność?

Co to jest współbieżność?

- Dwa działania lub więcej wykonywane jednocześnie

Co to jest współbieżność?

- Dwa działania lub więcej wykonywane jednocześnie
- Jeden procesor

Co to jest współbieżność?

- Dwa działania lub więcej wykonywane jednocześnie
- Jeden procesor
 - + szybkie przełączanie zadań - multitasking

Co to jest współbieżność?

- Dwa działania lub więcej wykonywane jednocześnie
- Jeden procesor
 - + szybkie przełączanie zadań - multitasking
- Wiele procesorów

Co to jest współbieżność?

- Dwa działania lub więcej wykonywane jednocześnie
- Jeden procesor
 - + szybkie przełączanie zadań - multitasking
- Wiele procesorów
 - + sprzętowa równoległość

Rodzaje współpracy

Rodzaje współbieżności

- Multiprocessing

Rodzaje współbieżności

- Multiprocessing
- Multithreading

Rodzaje współbieżności

- Multiprocessing
- Multithreading
- Vector parallelism (SIMD)

Proces vs. Wątek

Proces

- Egzemplarz wykonywanego programu
- Kontener wątków w chronionej przestrzeni adresowej
- System operacyjny przydziela procesowi odpowiednie zasoby
- Za zarządzanie procesami odpowiada jądro systemu operacyjnego
 - + system operacyjny zarządza priorytetami procesów

Komunikacja między procesami

- Sygnały
- Gniazda (sockets)
- Pliki
- Potoki
- Pamięć współdzielona
- Kolejki - *message queues*

Wątek

niezależny ciąg instrukcji wykonywany współbieżnie
w ramach jednego procesu

Wątki i zasoby

- Wszystkie wątki działające w danym procesie współdzielą przestrzeń adresową oraz zasoby systemowe
 - + pamięć (heap, static storage)
 - + pliki wymagane przez aplikację
 - + cykle CPU (multitasking)
 - + gniazda (sockets)

Natywne API dla wątków

- MS Windows - Win32 API
- Linux, BSD - pthread

Cross-platform C++ API

- Boost.Thread
- Threading Building Blocks - Intel
- Standard C++ Library - od C++11

g++ - opcje kompilacji

```
g++ -std=c++11 -pthread main.cpp
```

Po co używać współbieżności

Po co używać współbieżności

- Wydajność

Po co używać współbieżności

- Wydajność
 - + Chcemy wykorzystać całą moc maszyny wieloprocesorowej

Po co używać współbieżności

- Wydajność
 - + Chcemy wykorzystać całą moc maszyny wieloprocesorowej
 - + Chcemy podnieść skalowalność aplikacji

Po co używać współbieżności

- Wydajność
 - + Chcemy wykorzystać całą moc maszyny wieloprocesorowej
 - + Chcemy podnieść skalowalność aplikacji
 - + „The free lunch is over”

Po co używać współbieżności

- Wydajność
 - + Chcemy wykorzystać całą moc maszyny wieloprocesorowej
 - + Chcemy podnieść skalowalność aplikacji
 - + „The free lunch is over”
- Lepsza responsywność aplikacji

Po co używać współbieżności

- Wydajność
 - + Chcemy wykorzystać całą moc maszyny wieloprocesorowej
 - + Chcemy podnieść skalowalność aplikacji
 - + „The free lunch is over”
- Lepsza responsywność aplikacji
 - + Unikanie blokujących operacji I/O

Po co używać współbieżności

- Wydajność
 - + Chcemy wykorzystać całą moc maszyny wieloprocesorowej
 - + Chcemy podnieść skalowalność aplikacji
 - + „The free lunch is over”
- Lepsza responsywność aplikacji
 - + Unikanie blokujących operacji I/O
 - + Blokowanie interfejsu użytkownika

Kiedy nie używać współbieżności

Kiedy nie używać współbieżności

- gdy korzyści nie są warte kosztów

Problemy

Problemy

- Kod wielowątkowy jest dużo bardziej skomplikowany

Problemy

- Kod wielowątkowy jest dużo bardziej skomplikowany
 - + trudniejszy do pisania, czytania i testowania niż kod jednowątkowy

Problemy

- Kod wielowątkowy jest dużo bardziej skomplikowany
 - + trudniejszy do pisania, czytania i testowania niż kod jednowątkowy
- Większa złożoność, więcej błędów, które są trudne do wykrycia (często trudne do odtworzenia)

Problemy

- Kod wielowątkowy jest dużo bardziej skomplikowany
 - + trudniejszy do pisania, czytania i testowania niż kod jednowątkowy
- Większa złożoność, więcej błędów, które są trudne do wykrycia (często trudne do odtworzenia)
- Narzuty związane z zarządzaniem wątkami mogą drastycznie obniżyć wydajność

Problemy

- Kod wielowątkowy jest dużo bardziej skomplikowany
 - + trudniejszy do pisania, czytania i testowania niż kod jednowątkowy
- Większa złożoność, więcej błędów, które są trudne do wykrycia (często trudne do odtworzenia)
- Narzuty związane z zarządzaniem wątkami mogą drastycznie obniżyć wydajność
 - + thread oversubscription + context switching

Problemy

- Kod wielowątkowy jest dużo bardziej skomplikowany
 - + trudniejszy do pisania, czytania i testowania niż kod jednowątkowy
- Większa złożoność, więcej błędów, które są trudne do wykrycia (często trudne do odtworzenia)
- Narzuty związane z zarządzaniem wątkami mogą drastycznie obniżyć wydajność
 - + thread oversubscription + context switching
 - + cache ping-pong

Problemy

- Kod wielowątkowy jest dużo bardziej skomplikowany
 - + trudniejszy do pisania, czytania i testowania niż kod jednowątkowy
- Większa złożoność, więcej błędów, które są trudne do wykrycia (często trudne do odtworzenia)
- Narzuty związane z zarządzaniem wątkami mogą drastycznie obniżyć wydajność
 - + thread oversubscription + context switching
 - + cache ping-pong
 - + false sharing

Testowanie aplikacji wielowątkowych

Testowanie aplikacji wielowątkowych

- Kod jednowątkowy może być testowany jednostkowo (unit tests)

Testowanie aplikacji wielowątkowych

- Kod jednowątkowy może być testowany jednostkowo (unit tests)
 - + powtarzalne wyniki dla testów wykonywanych w izolacji

Testowanie aplikacji wielowątkowych

- Kod jednowątkowy może być testowany jednostkowo (unit tests)
 - + powtarzalne wyniki dla testów wykonywanych w izolacji
- Kod wielowątkowy jest trudno testowalny jednostkowo

Testowanie aplikacji wielowątkowych

- Kod jednowątkowy może być testowany jednostkowo (unit tests)
 - + powtarzalne wyniki dla testów wykonywanych w izolacji
- Kod wielowątkowy jest trudno testowalny jednostkowo
 - + błędy (np. race conditions) nie są deterministyczne i są ciężko reprodukowalne

Testowanie aplikacji wielowątkowych

- Kod jednowątkowy może być testowany jednostkowo (unit tests)
 - + powtarzalne wyniki dla testów wykonywanych w izolacji
- Kod wielowątkowy jest trudno testowalny jednostkowo
 - + błędy (np. race conditions) nie są deterministyczne i są ciężko reprodukowalne
 - + problemy pojawiają często przy dużym obciążeniu lub z wielokrotnieniu wątków

Testowanie aplikacji wielowątkowych

Dobrze jest zapewnić możliwość skalowania ilości wątków do jednego (single threaded code) aby wykluczyć inne przyczyny błędów

Wątki - std::thread

Zarządzanie wątkami

Zarządzanie wątkami

Zarządzanie wątkami

- Klasa `std :: thread` jest odpowiedzialna za tworzenie obiektów, które uruchamiają wątki systemowe i zarządzają nimi

Zarządzanie wątkami

- Klasa `std :: thread` jest odpowiedzialna za tworzenie obiektów, które uruchamiają wątki systemowe i zarządzają nimi
- Każda instancja klasy `std :: thread` reprezentuje

Zarządzanie wątkami

- Klasa `std :: thread` jest odpowiedzialna za tworzenie obiektów, które uruchamiają wątki systemowe i zarządzają nimi
- Każda instancja klasy `std :: thread` reprezentuje

Zarządzanie wątkami

- Klasa `std :: thread` jest odpowiedzialna za tworzenie obiektów, które uruchamiają wątki systemowe i zarządzają nimi
- Każda instancja klasy `std :: thread` reprezentuje
 - + pojedynczy wątek, utworzony przez system operacyjny

Zarządzanie wątkami

- Klasa `std :: thread` jest odpowiedzialna za tworzenie obiektów, które uruchamiają wątki systemowe i zarządzają nimi
- Każda instancja klasy `std :: thread` reprezentuje
 - + pojedynczy wątek, utworzony przez system operacyjny
 - + lub tzw. **wątek pusty** (*not-a-thread*)

Wątek pusty

Wątek pusty

Wątek pusty

- Instancja `std :: thread` utworzona konstruktorem domyślnym

Wątek pusty

- Instancja `std :: thread` utworzona konstruktorem domyślnym
- lub obiekt wątku po wykonaniu na nim operacji `std :: move()`

std::thread + std::move

std::thread + std::move

std::thread + std::move

- Obiekty wątków nie są kopiowalne - *non-copyable*

std::thread + std::move

- Obiekty wątków nie są kopiowalne - *non-copyable*
- Wątki systemowe mogą być przenoszone między obiektami

std::thread + std::move

- Obiekty wątków nie są kopiowalne - *non-copyable*
- Wątki systemowe mogą być przenoszone między obiektami

std::thread + std::move

- Obiekty wątków nie są kopiowalne - *non-copyable*
- Wątki systemowe mogą być przenoszone między obiektami
 - + semantyka przenoszenia - *move semantics*

std::thread + std::move

- Obiekty wątków nie są kopiowalne - *non-copyable*
- Wątki systemowe mogą być przenoszone między obiektami
 - + semantyka przenoszenia - *move semantics*
 - + w celu transferu wątku, który jest l-value należy użyć funkcji `std :: move`

Tworzenie wątku (1)

```
#include <thread>

void my_thread_func()
{
    std::cout << "My first thread ..." << std::endl;
}

int main()
{
    std::thread t(&my_thread_func);
    t.join();
}
```

Tworzenie wątku (1)

```
#include <thread>

void my_thread_func()
{
    std::cout << "My first thread ..." << std::endl;
}

int main()
{
    std::thread t(&my_thread_func);
    t.join();
}
```

Tworzenie wątku (1)

```
#include <thread>

void my_thread_func()
{
    std::cout << "My first thread ..." << std::endl;
}

int main()
{
    std::thread t(&my_thread_func);
    t.join();
}
```

Tworzenie wątku (1)

```
#include <thread>

void my_thread_func()
{
    std::cout << "My first thread ..." << std::endl;
}

int main()
{
    std::thread t(&my_thread_func);
    t.join();
}
```

Tworzenie wątku (2)

```
class BackgroundTask
{
public:
    void operator()() const
    {
        std::cout << "Hello from a thread..." << std::endl;
    }
};

int main()
{
    BackgroundTask bt;
    std::thread t1(bt);
    std::thread t2(BackgroundTask());
    t1.join();
    t2.join();
}
```

Tworzenie wątku (2)

```
class BackgroundTask
{
public:
    void operator()() const
    {
        std::cout << "Hello from a thread..." << std::endl;
    }
};

int main()
{
    BackgroundTask bt;
    std::thread t1(bt);
    std::thread t2(BackgroundTask());
    t1.join();
    t2.join();
}
```

Tworzenie wątku (2)

```
class BackgroundTask
{
public:
    void operator()() const
    {
        std::cout << "Hello from a thread..." << std::endl;
    }
};

int main()
{
    BackgroundTask bt;
    std::thread t1(bt);
    std::thread t2(BackgroundTask());
    t1.join();
    t2.join();
}
```

Tworzenie wątku (3)

```
std::thread t([] {  
    std::cout << "My first thread ..." << std::endl;  
});  
  
t.join(); f
```

Dołączanie do wątku

Dołączenie do wątku

- Aby poczekać na zakończenie wykonywania zadania przez dany wątek, należy wywołać na jego rzecz metodę `join()`

Dołączenie do wątku

- Aby poczekać na zakończenie wykonywania zadania przez dany wątek, należy wywołać na jego rzecz metodę `join()`
- `join()` blokuje (wstrzymuje wywołanie) bieżący wątek, aż do czasu ukończenia zadania wykonywanego przez wskazany wątek

Dołączanie do wątku

```
int main()
{
    BackgroundTask bt;

    std::thread t(bt);
    t.join(); // wstrzymanie wykonania wątku głównego (main),
              // aż do czasu zakończenia zadania w wątku t

    assert(!thd.joinable());
}
```

Dołączanie do wątku

```
int main()
{
    BackgroundTask bt;

    std::thread t(bt);
    t.join(); // wstrzymanie wykonania wątku głównego (main),
              // aż do czasu zakończenia zadania w wątku t

    assert(!thd.joinable());
}
```

C++ Core Guidelines - join

Prefer `gsl::joining_thread` over `std::thread`

Destruktor wątku

Destruktor wątku

- Jeśli obiekt wątku jest skojarzony z wątkiem systemowym wywołanie metody `joinable()` zwraca `true`.

Destruktor wątku

- Jeśli obiekt wątku jest skojarzony z wątkiem systemowym wywołanie metody `joinable()` zwraca `true`.
- Destrukcja obiektu wątku jest bezpieczna, jeśli obiekt wątku nie jest skojarzony z wątkiem systemowym.

Destruktor wątku

- Jeśli obiekt wątku jest skojarzony z wątkiem systemowym wywołanie metody `joinable()` zwraca `true`.
- Destrukcja obiektu wątku jest bezpieczna, jeśli obiekt wątku nie jest skojarzony z wątkiem systemowym.
- W przeciwnym wypadku wywołana jest funkcja `std :: terminate()`

Destruktor wątku

Destruktor wątku

- Obiekt wątku nie jest skojarzony z wątkiem systemowym:

Destruktor wątku

- Obiekt wątku nie jest skojarzony z wątkiem systemowym:
 - + jeśli został utworzony za pomocą konstruktora domyślnego

Destruktor wątku

- Obiekt wątku nie jest skojarzony z wątkiem systemowym:
 - + jeśli został utworzony za pomocą konstruktora domyślnego
 - + został przeniesiony do innego obiektu - np. jest po wywołaniu `std :: move()`

Destruktor wątku

- Obiekt wątku nie jest skojarzony z wątkiem systemowym:
 - + jeśli został utworzony za pomocą konstruktora domyślnego
 - + został przeniesiony do innego obiektu - np. jest po wywołaniu `std :: move()`
 - + została wcześniej wywołana operacja `join()`

Destruktor wątku

- Obiekt wątku nie jest skojarzony z wątkiem systemowym:
 - + jeśli został utworzony za pomocą konstruktora domyślnego
 - + został przeniesiony do innego obiektu - np. jest po wywołaniu `std :: move()`
 - + została wcześniej wywołana operacja `join()`
 - + została wcześniej wywołana operacja `detach()`

Destrukcja wątku

Standard C++ wymusza jawne wywołanie `join()` lub `detach()` przed wywołaniem destruktora wątku, który jest skojarzony z wątkiem systemowym.

Przekazywanie parametrów do wątków

3 sposoby

Sposób 1

- Przekazując argumenty jako kolejne (po funkcji lub obiekcie funkcyjnym) parametry konstruktora `std :: thread`
- Podane argumenty są kopiowane lub przenoszone do uruchamianego wątku.

```
void do_work(ThreadSafeQueue<Data>& q, std::string value)
{
    Data data = process_data(value);
    q.push(data)
}

// ...

ThreadSafeQueue<Data> data_queue;
const std::string value = "some data";

std::thread thd{&do_work, std::ref(data_queue), value};

// ...
thd.join();
```

```
void do_work(ThreadSafeQueue<Data>& q, std::string value)
{
    Data data = process_data(value);
    q.push(data)
}

// ...

ThreadSafeQueue<Data> data_queue;
const std::string value = "some data";

std::thread thd{&do_work, std::ref(data_queue), value};

// ...
thd.join();
```

```
void do_work(ThreadSafeQueue<Data>& q, std::string value)
{
    Data data = process_data(value);
    q.push(data)
}

// ...

ThreadSafeQueue<Data> data_queue;
const std::string value = "some data";

std::thread thd{&do_work, std::ref(data_queue), value};

// ...
thd.join();
```

Przekazanie referencji

- Jeśli wymagane jest przekazanie referencji do funkcji uruchamianej w nowym wątku, należy użyć standardowych wrapperów dla:
 - + referencji - `std :: ref()`
 - + referencji do stałej - `std :: cref()`

Sposób 2

- Wykorzystując obiekt domknięcia, który przechwytuje zmienne będące argumentami wywoływanej funkcji

```
void do_work(ThreadSafeQueue<Data>& q, std::string value)
{
    Data data = process_data(value);
    q.push(data)
}
```

```
// ...

ThreadSafeQueue<Data> data_queue;
const std::string value = "some data";

std::thread thd{&data_queue, value} { do_work(data_queue, value); }};

thd.join();
```

```
void do_work(ThreadSafeQueue<Data>& q, std::string value)
{
    Data data = process_data(value);
    q.push(data)
}
```

```
// ...

ThreadSafeQueue<Data> data_queue;
const std::string value = "some data";

std::thread thd{&data_queue, value} { do_work(data_queue, value); }};

thd.join();
```

Sposób 3

Konstrukcja obiektu funkcyjnego przekazywanego do konstruktora instancji
std :: thread:

```
class Processor
{
    ThreadSafeVector& data_;
    const int slot_index_;
public:
    Processor(ThreadSafeVector& data, int index)
        : data_{data}, slot_index_{index}
    {}

    void operator()()
    {
        process(data_, slot_index_);
    }
};

ThreadSafeVector vec(2);
const int slot_index = 1;

Processor processor_1{vec, slot_index};
std::thread thd{processor_1};

thd.join();
```

```
class Processor
{
    ThreadSafeVector& data_;
    const int slot_index_;
public:
    Processor(ThreadSafeVector& data, int index)
        : data_{data}, slot_index_{index}
    {}

    void operator()()
    {
        process(data_, slot_index_);
    }
};

ThreadSafeVector vec(2);
const int slot_index = 1;

Processor processor_1{vec, slot_index};
std::thread thd{processor_1};

thd.join();
```

```
class Processor
{
    ThreadSafeVector& data_;
    const int slot_index_;
public:
    Processor(ThreadSafeVector& data, int index)
        : data_{data}, slot_index_{index}
    {}

    void operator()()
    {
        process(data_, slot_index_);
    }
};

ThreadSafeVector vec(2);
const int slot_index = 1;

Processor processor_1{vec, slot_index};
std::thread thd{processor_1};

thd.join();
```

Problem wiszących referencji

Przy przekazywaniu parametrów do wątków należy unikać wiszących referencji
(dangling references)

C++ Core Guidelines [CP.31]

std::thread & move semantics

Klasa std :: thread implementuje tylko semantykę przenoszenia

```
void task() { /* implementation */ }

std::thread create_thread()
{
    std::thread thd(&task);
    return thd;
}

std::vector<std::thread> threads;

// implicit move of r-value
threads.push_back(create_thread());

// explicit move of l-value
std::thread thd(&task);
threads.push_back(std::move(thd));

for(auto& thd : threads)
    thd.join();
```

```
void task() { /* implementation */ }

std::thread create_thread()
{
    std::thread thd(&task);
    return thd;
}

std::vector<std::thread> threads;

// implicit move of r-value
threads.push_back(create_thread());

// explicit move of l-value
std::thread thd(&task);
threads.push_back(std::move(thd));

for(auto& thd : threads)
    thd.join();
```

wątki mogą być zwracane z funkcji

```
void task() { /* implementation */ }

std::thread create_thread()
{
    std::thread thd(&task);
    return thd;
}

std::vector<std::thread> threads;

// implicit move of r-value
threads.push_back(create_thread());

// explicit move of l-value
std::thread thd(&task);
threads.push_back(std::move(thd));

for(auto& thd : threads)
    thd.join();
```

implicit move

```
void task() { /* implementation */ }

std::thread create_thread()
{
    std::thread thd(&task);
    return thd;
}

std::vector<std::thread> threads;

// implicit move of r-value
threads.push_back(create_thread());

// explicit move of l-value
std::thread thd(&task);
threads.push_back(std::move(thd));

for(auto& thd : threads)
    thd.join();
```

explicit move

Wątki + wyjątki

Obsługa wyjątków w wątkach

Obsługa wyjątków w wątkach

Jeżeli z funkcji uruchomionej w osobnym wątku wydostanie się wyjątek, zostanie wywołana funkcja `std :: terminate()`

Obsługa wyjątków w wątkach

Jeżeli z funkcji uruchomionej w osobnym wątku wydostanie się wyjątek, zostanie wywołana funkcja `std :: terminate()`

W celu prawidłowej obsługi wyjątków, należy przechwycić rzucony wyjątek i jeżeli istnieje taka potrzeba, przekazać go do wątku rodzica wykorzystując klasę `std :: exception_ptr`

Klasa std::exception_ptr

Klasa std::exception_ptr

- Umożliwia przechowanie wskaźnika do obiektu wyjątku, który został zgłoszony instrukcją `throw` i przechwycony funkcją `std :: current_exception()`

Klasa std::exception_ptr

- Umożliwia przechowanie wskaźnika do obiektu wyjątku, który został zgłoszony instrukcją `throw` i przechwycony funkcją `std :: current_exception()`
- Instancja `std :: exception_ptr` może być przekazana do innej funkcji, również takiej, która uruchomiona jest w osobnym wątku

Klasa std::exception_ptr

- Umożliwia przechowanie wskaźnika do obiektu wyjątku, który został zgłoszony instrukcją `throw` i przechwycony funkcją `std :: current_exception()`
- Instancja `std :: exception_ptr` może być przekazana do innej funkcji, również takiej, która uruchomiona jest w osobnym wątku
- Obsługa przekazanego przez wskaźnik wyjątku jest możliwa przy pomocy funkcji `std :: rethrow_exception()`, która powoduje ponowne rzucenie wyjątku

Klasa std::exception_ptr

Klasa std::exception_ptr

- Domyślnie skonstruowany obiekt `std :: exception_ptr` ma wartość `nullptr`

Klasa std::exception_ptr

- Domyślnie skonstruowany obiekt `std :: exception_ptr` ma wartość `nullptr`
- Dwa obiekty są uznawane za równe, jeśli są puste lub wskazują na ten sam obiekt wyjątku.

Klasa std::exception_ptr

- Domyślnie skonstruowany obiekt `std :: exception_ptr` ma wartość `nullptr`
- Dwa obiekty są uznawane za równe, jeśli są puste lub wskazują na ten sam obiekt wyjątku.
- Instancja `std :: exception_ptr` jest konwertowalna do wartości logicznej

```
void background_work(std::exception_ptr& excpt)
{
    try
    {
        may_throw();

        // ...
    }
    catch( ... )
    {
        excpt = std::current_exception();
    }
}
```

```
void background_work(std::exception_ptr& excpt)
{
    try
    {
        may_throw();

        // ...
    }
    catch( ... )
    {
        excpt = std::current_exception();
    }
}
```

```
void background_work(std::exception_ptr& except)
{
    try
    {
        may_throw();

        // ...
    }
    catch( ... )
    {
        except = std::current_exception();
    }
}
```



```
int main()
{
    std::exception_ptr thd_exception;

    std::thread thd(&background_work, std::ref(thd_exception));
    thd.join();

    try
    {
        std::rethrow_exception(e);
    }
    catch(const std::runtime_error& e)
    {
        // ... handling an error
    }
}
```

```
int main()
{
    std::exception_ptr thd_exception;

    std::thread thd(&background_work, std::ref(thd_exception));
    thd.join();

    try
    {
        std::rethrow_exception(e);
    }
    catch(const std::runtime_error& e)
    {
        // ... handling an error
    }
}
```

object that can store an exception thrown in another thread

```
int main()
{
    std::exception_ptr thd_exception;

    std::thread thd(&background_work, std::ref(thd_exception));
    thd.join();

    try
    {
        std::rethrow_exception(e);
    }
    catch(const std::runtime_error& e)
    {
        // ... handling an error
    }
}
```

is passed by reference to spawned thread


```
int main()
{
    std::exception_ptr thd_exception;

    std::thread thd(&background_work, std::ref(thd_exception));
    thd.join();

    try
    {
        std::rethrow_exception(e);
    }
    catch(const std::runtime_error& e)
    {
        // ... handling an error
    }
}
```

handling an exception in the main thread

Funkcje i klasy pomocnicze

std::thread::hardware_concurrency()

- Statyczna metoda zwracająca ilość dostępnych wątków sprzętowych
- Zwykle podawana jest ilość procesorów, rdzeni, itp. Jeżeli informacja nie jest dostępna zwraca wartość 0

std::thread::hardware_concurrency()

- Statyczna metoda zwracająca ilość dostępnych wątków sprzętowych
- Zwykle podawana jest ilość procesorów, rdzeni, itp. Jeżeli informacja nie jest dostępna zwraca wartość 0

```
auto hardware_threads_count =  
    std::max(1u, std::thread::hardware_concurrency());  
  
std::vector<std::thread> threads(hardware_threads_count);
```

Przestrzeń nazw std::this_thread

Przestrzeń nazw std::this_thread

- `sleep_for(const chrono::duration<R, P>& sleep_duration)`

Przestrzeń nazw std::this_thread

- `sleep_for(const chrono::duration<R, P>& sleep_duration)`
+ wstrzymuje wykonanie bieżącego wątku na (przynajmniej) określony interwał czasu

Przestrzeń nazw std::this_thread

- `sleep_for(const chrono::duration<R, P>& sleep_duration)`
+ wstrzymuje wykonanie bieżącego wątku na (przynajmniej) określony interwał czasu
- `sleep_until(const chrono::time_point<C, D>& sleep_time)`

Przestrzeń nazw std::this_thread

- `sleep_for(const chrono::duration<R, P>& sleep_duration)`
 - + wstrzymuje wykonanie bieżącego wątku na (przynajmniej) określony interwał czasu
- `sleep_until(const chrono::time_point<C, D>& sleep_time)`
 - + blokuje wykonanie wątku przynajmniej do podanego jako parametr punktu czasu

Przestrzeń nazw std::this_thread

- `sleep_for(const chrono::duration<R, P>& sleep_duration)`
 - + wstrzymuje wykonanie bieżącego wątku na (przynajmniej) określony interwał czasu
- `sleep_until(const chrono::time_point<C, D>& sleep_time)`
 - + blokuje wykonanie wątku przynajmniej do podanego jako parametr punktu czasu
- `yield()`

Przestrzeń nazw std::this_thread

- `sleep_for(const chrono::duration<R, P>& sleep_duration)`
 - + wstrzymuje wykonanie bieżącego wątku na (przynajmniej) określony interwał czasu
- `sleep_until(const chrono::time_point<C, D>& sleep_time)`
 - + blokuje wykonanie wątku przynajmniej do podanego jako parametr punktu czasu
- `yield()`
 - + funkcja umożliwiająca podjęcie próby wywłaszczenia bieżącego wątku i przydzielenia czasu procesora innemu wątkowi

Przestrzeń nazw std::this_thread

- `sleep_for(const chrono::duration<R, P>& sleep_duration)`
 - + wstrzymuje wykonanie bieżącego wątku na (przynajmniej) określony interwał czasu
- `sleep_until(const chrono::time_point<C, D>& sleep_time)`
 - + blokuje wykonanie wątku przynajmniej do podanego jako parametr punktu czasu
- `yield()`
 - + funkcja umożliwiająca podjęcie próby wywłaszczenia bieżącego wątku i przydzielenia czasu procesora innemu wątkowi
- `get_id()`

Przestrzeń nazw std::this_thread

- `sleep_for(const chrono::duration<R, P>& sleep_duration)`
 - + wstrzymuje wykonanie bieżącego wątku na (przynajmniej) określony interwał czasu
- `sleep_until(const chrono::time_point<C, D>& sleep_time)`
 - + blokuje wykonanie wątku przynajmniej do podanego jako parametr punktu czasu
- `yield()`
 - + funkcja umożliwiająca podjęcie próby wywłaszczenia bieżącego wątku i przydzielenia czasu procesora innemu wątkowi
- `get_id()`
 - + zwraca obiekt typu `std::thread::id` reprezentujący identyfikator bieżącego wątku

Identyfikator wątku

`std :: thread :: id` jest lekką, trywialnie kopiowalną klasą, która opakowuje unikalny identyfikator wątku.

Identyfikator wątku

`std :: thread :: id` jest lekką, trywialnie kopiowalną klasą, która opakowuje unikalny identyfikator wątku.

Celem klasy jest możliwość użycia jej jako klucza w kontenerach asocjacyjnych

Klasa std::thread::id

Klasa std::thread::id

- Domyślny konstruktor tworzy obiekt identyfikujący tzw. pusty wątek (Not-A-Thread)

Klasa std::thread::id

- Domyślny konstruktor tworzy obiekt identyfikujący tzw. pusty wątek (Not-A-Thread)
- Instancje `thread :: id` są kopiowalne, porównywalne oraz hashowalne

Klasa std::thread::id

- Domyślny konstruktor tworzy obiekt identyfikujący tzw. pusty wątek (Not-A-Thread)
- Instancje `thread :: id` są kopiowalne, porównywalne oraz hashowalne
 - + w klasie zdefiniowany jest pełen zestaw operatorów porównania

Klasa std::thread::id

- Domyślny konstruktor tworzy obiekt identyfikujący tzw. pusty wątek (Not-A-Thread)
- Instancje `thread :: id` są kopiowalne, porównywalne oraz hashowalne
 - + w klasie zdefiniowany jest pełen zestaw operatorów porównania
 - + biblioteka standardowa definiuje pomocniczą klasę `std :: hash<std :: thread :: id>`, która umożliwia przechowanie wątku w kontenerach hashujących (np. `unordered_map`)

Klasa std::thread::id

- Domyślny konstruktor tworzy obiekt identyfikujący tzw. pusty wątek (Not-A-Thread)
- Instancje `thread :: id` są kopiowalne, porównywalne oraz hashowalne
 - + w klasie zdefiniowany jest pełen zestaw operatorów porównania
 - + biblioteka standardowa definiuje pomocniczą klasę `std :: hash<std :: thread :: id>`, która umożliwia przechowanie wątku w kontenerach hashujących (np. `unordered_map`)
- Klasa posiada operator wyjścia do strumienia `operator <<`

Synchronizacja

Poprawność programów współbieżnych

Poprawność programów współbieżnych

- Poprawny program współbieżny musi spełniać dwie właściwości:

Poprawność programów współbieżnych

- Poprawny program współbieżny musi spełniać dwie właściwości:
 - + Bezpieczeństwa - *Safety*

Poprawność programów współbieżnych

- Poprawny program współbieżny musi spełniać dwie właściwości:
 - + Bezpieczeństwa - *Safety*
 - + Żywotności - *Liveness*

Bezpieczeństwo

- *Safety*
 - + nigdy nie doprowadza do niepożądanego stanu (*nothing bad will ever happens*)
 - + bezpieczeństwo może być naruszone przez:
 - data-race
 - deadlock

Żywotność

- *Liveness*

- + zapewnia, że każde pożądane zdarzenie w końcu zajdzie
- + żywotność może być naruszona poprzez:
 - deadlock
 - livelock
 - zagłodzenie wątków

Thread safety

- Segment kodu jest „thread-safe” jeśli manipuluje współdzielonym stanem w taki sposób, że gwarantowane jest bezpieczne (prawidłowe) wykonanie go przez wiele wątków pracujących w tym samym czasie
- Oznacza to, że kod „thread-safe” musi w odpowiedni sposób chronić współdzielony między wątkami stan programu

Unsafe code

- nieprawidłowo chroni współdzielone dane oraz zasoby (data race)
- zależy od przechowywanego między wywołaniami stanu - np. funkcja strtok(), rand()
- wywołuje kod, który jest „unsafe”

Safe code

- Niezmienność danych (*immutability*)
 - + stan obiektu nie może być zmieniony po jego utworzeniu
 - + obiekty mogą być bezpiecznie współdzielone, jeśli wątki mogą jedynie odczytywać ich stan
- Stosowanie jawnego blokowania (*explicit locking*)
 - + muteksów, do ochrony danych współdzielonych
 - + operacje synchronizacji mogą znacznie spowolnić działanie programu
- Atomowość (*atomicity*)
 - + stosowanie zmiennych i flag atomowych

Shared mutable state is an evil of all multithreading

Immutability

- Zmienne, które są niezmienne (immutable) mogą być bezpiecznie współdzielone bez blokad
- `const` w C++11 znaczy thread-safe

Współdzielanie kontenerów

- Dozwolony jest współbieżny odczyt (operacje read-only) np:
 - + wywołanie metod `begin()`, `end()`, `rbegin()`, `rend()`, `front()`, `back()`, `lower_bound()`, `upper_bound()`, `equal_range()`, `at()`
 - + operator `[]` - z wyjątkiem kontenerów asocjacyjnych
 - + dostęp za pomocą iteratorów, jeśli nie modyfikujemy stanu wskazywanych elementów
- Dozwolony jest współbieżny dostęp do różnych elementów kontenera za wyjątkiem obiektów `bitset` oraz `vector<bool>`

Współdzielenie strumieni

- Dla formatowanych operacji wejścia oraz wyjścia na obiektach strumieni standardowych (`cin`, `cout`, `cerr`, ...)
 - + dozwolony jest dostęp współbieżny
 - + może wystąpić przemieszanie kolejności znaków (interleaved characters)

Shared mutable satte ias n eivl ofall multihraeding

Muteksy

Sekcja krytyczna

- Fragment kodu programu, który w danej chwili może być wykonywany tylko przez jeden wątek
- Sekcja krytyczna zapewnia właściwość **wzajemnego wykluczenia**

Podstawowe obiekty blokad

- Muteksy (*mutual exclusion*) - obiekty implementujące właściwość wzajemnego wykluczania
- Semafora - binarne i zliczające
- Blokady współdzielone (blokady readers-writers)
- Zmienne warunkowe

Muteks

- Standardową realizacją wzajemnego wykluczenia jest wykorzystanie obiektu blokady (**muteksu**) zawierającego operacje
 - + `lock()`
 - + `unlock()`

Muteksy w bibliotece standardowej

Muteksy w bibliotece standardowej

- Zapewnia implementację operacji wzajemnego wykluczenia (*mutual exclusion*)

Muteksy w bibliotece standardowej

- Zapewnia implementację operacji wzajemnego wykluczenia (*mutual exclusion*)
- Umożliwia ochronę współdzielonych zmiennych przed sytuacją wyścigu (*race condition*)

Muteksy w bibliotece standardowej

- Zapewnia implementację operacji wzajemnego wykluczenia (*mutual exclusion*)
- Umożliwia ochronę współdzielonych zmiennych przed sytuacją wyścigu (*race condition*)
 - + wątek pozyskuje mutex (wchodzi do sekcji krytycznej) wywołując na rzecz mutexu metodę `lock()`

Muteksy w bibliotece standardowej

- Zapewnia implementację operacji wzajemnego wykluczenia (*mutual exclusion*)
- Umożliwia ochronę współdzielonych zmiennych przed sytuacją wyścigu (*race condition*)
 - + wątek pozyskuje muteks (wchodzi do sekcji krytycznej) wywołując na rzecz muteksu metodę `lock()`
 - + wątek opuszcza sekcję krytyczną - zwalnia muteks - wywołując metodę `unlock()`

Koncepty obiektów blokad

Koncepty obiektów blokad

- BasicLockable

Koncepty obiektów blokad

- BasicLockable
- Lockable

Koncepty obiektów blokad

- BasicLockable
- Lockable
- TimedLockable

Koncepty obiektów blokad

- BasicLockable
- Lockable
- TimedLockable
- SharedLockable

Koncept BasicLockable

```
template<typename L>
concept BasicLockable =
requires (L m) {
    m.lock();
    m.unlock();
};
```

Koncept BasicLockable

```
template<typename L>
concept BasicLockable =
requires (L m) {
    m.lock();
    m.unlock();
};
```

Koncept Lockable

```
template<typename L>
concept Lockable = BasicLockable<L> &&
requires (L m) {
    { m.try_lock(); } → std::convertible_to<bool>;
};
```

Koncept Lockable

```
template<typename L>
concept Lockable = BasicLockable<L> &&
requires (L m) {
    { m.try_lock(); } → std::convertible_to<bool>;
};
```

std::mutex

std::mutex

- implementuje koncept **Lockable** zapewniając podstawowy mechanizm synchronizacji, który może być użyty do implementacji bezpiecznego dostępu do współdzielonego zasobu

std::mutex

- implementuje koncept `Lockable` zapewniając podstawowy mechanizm synchronizacji, który może być użyty do implementacji bezpiecznego dostępu do współdzielonego zasobu
- wywołujący wątek posiada muteks od momentu udanego wywołania metody `lock()` lub `try_lock()` do wywołanie metody `unlock()`

std::mutex

- implementuje koncept `Lockable` zapewniając podstawowy mechanizm synchronizacji, który może być użyty do implementacji bezpiecznego dostępu do współdzielonego zasobu
- wywołujący wątek posiada muteks od momentu udanego wywołania metody `lock()` lub `try_lock()` do wywołanie metody `unlock()`
- jeśli wątek posiada muteks, wszystkie inne wątki wywołujące `lock()` będą blokowane lub zwrócią `false` po wywołaniu `try_lock()`

std::mutex

- implementuje koncept `Lockable` zapewniając podstawowy mechanizm synchronizacji, który może być użyty do implementacji bezpiecznego dostępu do współdzielonego zasobu
- wywołujący wątek posiada muteks od momentu udanego wywołania metody `lock()` lub `try_lock()` do wywołanie metody `unlock()`
- jeśli wątek posiada muteks, wszystkie inne wątki wywołujące `lock()` będą blokowane lub zwrócią `false` po wywołaniu `try_lock()`
- wywołujący wątek nie może posiadać muteksu przed wywołaniem `lock()` lub `try_lock()` - `std :: mutex` implementuje nie-rekursywną wersję muteksu

std::recursive_mutex

std::recursive_mutex

- implementuje rekursywną wersję konceptu Lockable

std::recursive_mutex

- implementuje rekursywną wersję konceptu Lockable
- ten sam wątek może wielokrotnie pozyskać muteks poprzez wywołanie metody `lock()` lub `try_lock()`

std::recursive_mutex

- implementuje rekursywną wersję konceptu Lockable
- ten sam wątek może wielokrotnie pozyskać muteks poprzez wywołanie metody `lock()` lub `try_lock()`
- aby zwolnić muteks wątek musi odpowiednią ilość razy wywołać `unlock()`

std::recursive_mutex

- implementuje rekursywną wersję konceptu Lockable
- ten sam wątek może wielokrotnie pozyskać muteks poprzez wywołanie metody `lock()` lub `try_lock()`
- aby zwolnić muteks wątek musi odpowiednią ilość razy wywołać `unlock()`
- maksymalny poziom rekursji dla obiektu `std :: recursive_mutex` nie jest zdefiniowany przez standard, ale po przekroczeniu tej wartości:

std::recursive_mutex

- implementuje rekursywną wersję konceptu Lockable
- ten sam wątek może wielokrotnie pozyskać muteks poprzez wywołanie metody `lock()` lub `try_lock()`
- aby zwolnić muteks wątek musi odpowiednią ilość razy wywołać `unlock()`
- maksymalny poziom rekursji dla obiektu `std :: recursive_mutex` nie jest zdefiniowany przez standard, ale po przekroczeniu tej wartości:
 - + z metody `lock()` rzucony zostanie wyjątek `std :: system_error`

std::recursive_mutex

- implementuje rekursywną wersję konceptu Lockable
- ten sam wątek może wielokrotnie pozyskać muteks poprzez wywołanie metody `lock()` lub `try_lock()`
- aby zwolnić muteks wątek musi odpowiednią ilość razy wywołać `unlock()`
- maksymalny poziom rekursji dla obiektu `std :: recursive_mutex` nie jest zdefiniowany przez standard, ale po przekroczeniu tej wartości:
 - + z metody `lock()` rzucony zostanie wyjątek `std :: system_error`
 - + metoda `try_lock()` zwróci `false`

Menadżery blokad

Menadżery blokad

- Zarządzanie blokadami (obiektami muteksów) odbywa się za pomocą następujących klas implementujących RAII:

Menadżery blokad

- Zarządzanie blokadami (obiektami muteksów) odbywa się za pomocą następujących klas implementujących RAII:
 - + `lock_guard<Mutex>`

Menadżery blokad

- Zarządzanie blokadami (obiektami muteksów) odbywa się za pomocą następujących klas implementujących RAII:
 - + `lock_guard<Mutex>`
 - + `unique_lock<Mutex>`

Menadżery blokad

- Zarządzanie blokadami (obiektami muteksów) odbywa się za pomocą następujących klas implementujących RAII:
 - + `lock_guard<Mutex>`
 - + `unique_lock<Mutex>`
 - + `shared_lock<Mutex>`

std::lock_guard

```
template<typename _Mutex>
class lock_guard
{
public:
    typedef _Mutex mutex_type;

    explicit lock_guard(mutex_type& __m) : _M_device(__m)
    { _M_device.lock(); }

    lock_guard(mutex_type& __m, adopt_lock_t) : _M_device(__m)
    { } // calling thread owns mutex

    ~lock_guard()
    { _M_device.unlock(); }

    lock_guard(const lock_guard&) = delete;
    lock_guard& operator=(const lock_guard&) = delete;

private:
    mutex_type& _M_device;
};
```

std::lock_guard

```
template<typename _Mutex>
class lock_guard
{
public:
    typedef _Mutex mutex_type;

    explicit lock_guard(mutex_type& __m) : _M_device(__m)
    { _M_device.lock(); }

    lock_guard(mutex_type& __m, adopt_lock_t) : _M_device(__m)
    { } // calling thread owns mutex

    ~lock_guard()
    { _M_device.unlock(); }

    lock_guard(const lock_guard&) = delete;
    lock_guard& operator=(const lock_guard&) = delete;

private:
    mutex_type& _M_device;
};
```

konstruktor pozyskuje blokadę

std::lock_guard

```
template<typename _Mutex>
class lock_guard
{
public:
    typedef _Mutex mutex_type;

    explicit lock_guard(mutex_type& __m) : _M_device(__m)
    { _M_device.lock(); }

    lock_guard(mutex_type& __m, adopt_lock_t) : _M_device(__m)
    { } // calling thread owns mutex

    ~lock_guard()
    { _M_device.unlock(); }

    lock_guard(const lock_guard&) = delete;
    lock_guard& operator=(const lock_guard&) = delete;

private:
    mutex_type& _M_device;
};
```

destruktor zwalnia blokadę wywołując unlock()

std::lock_guard

```
template<typename _Mutex>
class lock_guard
{
public:
    typedef _Mutex mutex_type;

    explicit lock_guard(mutex_type& __m) : _M_device(__m)
    { _M_device.lock(); }

    lock_guard(mutex_type& __m, adopt_lock_t) : _M_device(__m)
    { } // calling thread owns mutex

    ~lock_guard()
    { _M_device.unlock(); }

    lock_guard(const lock_guard&) = delete;
    lock_guard& operator=(const lock_guard&) = delete;

private:
    mutex_type& _M_device;
};
```

adaptowanie, tj. przejęcie prawa własności i tym samym odpowiedzialności za zwolnienie pozyskanej
już wcześniej blokady

```
template <typename Value_>
struct Synchronized {
    Value_ value;
    std::mutex mtx_value;
};

template <typename F_, typename Value_>
void apply(F_ f, Synchronized<Value_> &sync_value)
{
    std::lock_guard<std::mutex> lk{sync_value.mtx_value};
    f(sync_value.value);
}
```

```
Synchronized<std::vector<int>> sync_vec;

std::thread thd([&] {
    apply([](std::vector<int> &v) {
        v.push_back(1);
        v.push_back(2);
    }, sync_vec);
});
```

thd.join();

```
template <typename Value_>
struct Synchronized {
    Value_ value;
    std::mutex mtx_value;
};

template <typename F_, typename Value_>
void apply(F_ f, Synchronized<Value_> &sync_value)
{
    std::lock_guard<std::mutex> lk{sync_value.mtx_value};
    f(sync_value.value);
}
```

```
Synchronized<std::vector<int>> sync_vec;

std::thread thd([&] {
    apply([](std::vector<int> &v) {
        v.push_back(1);
        v.push_back(2);
    }, sync_vec);
});
```

thd.join();

```
template <typename Value_>
struct Synchronized {
    Value_ value;
    std::mutex mtx_value;
};

template <typename F_, typename Value_>
void apply(F_ f, Synchronized<Value_> &sync_value)
{
    std::lock_guard<std::mutex> lk{sync_value.mtx_value};
    f(sync_value.value);
}
```

```
Synchronized<std::vector<int>> sync_vec;

std::thread thd([&] {
    apply([](std::vector<int> &v) {
        v.push_back(1);
        v.push_back(2);
    }, sync_vec);
});
```

thd.join();

```
template <typename Value_>
struct Synchronized {
    Value_ value;
    std::mutex mtx_value;
};

template <typename F_, typename Value_>
void apply(F_ f, Synchronized<Value_> &sync_value)
{
    std::lock_guard<std::mutex> lk{sync_value.mtx_value};
    f(sync_value.value);
}
```

```
Synchronized<std::vector<int>> sync_vec;

std::thread thd([&] {
    apply([](std::vector<int> &v) {
        v.push_back(1);
        v.push_back(2);
    }, sync_vec);
});
```

thd.join();

```
template <typename Value_>
struct Synchronized {
    Value_ value;
    std::mutex mtx_value;
};

template <typename F_, typename Value_>
void apply(F_ f, Synchronized<Value_> &sync_value)
{
    std::lock_guard<std::mutex> lk{sync_value.mtx_value};
    f(sync_value.value);
}
```

```
Synchronized<std::vector<int>> sync_vec;

std::thread thd([&] {
    apply([](std::vector<int> &v) {
        v.push_back(1);
        v.push_back(2);
    }, sync_vec);
});
```

thd.join();

`std::unique_lock`

- Rozbudowana wersja managera blokad umożliwiająca:
 - + ochronę RAII przed wyciekami blokad
 - + opóźnione pozyskiwanie blokad - `deferred_lock`
 - + adaptowanie pozyskanej przez wątek blokady - `adopt_lock`
 - + transfer prawa własności
 - instancja `unique_lock` nie jest kopiowalna, ale jest przenaszalna (moveable)
 - + podejmowanie nieblokujących prób pozyskania blokady - `try_lock()`
 - + korzystanie z muteksów czasowych

`std::unique_lock`

- Jeśli instancja managera `std :: unique_lock` posiada blokadę:
 - + metoda `mutex()` zwraca wskaźnik do muteksu realizującego blokadę
 - + metoda `owns_lock()` zwraca wartość `true`
 - + w momencie niszczenia obiektu destruktor wywoła metodę `unlock()` na obiekcie muteksu

std::unique_lock - std::try_to_lock

- Nieblokujące próby pozyskania blokady -
`unique_lock(Lockable& m, std :: try_to_lock_t)`
 - + w konstruktorze wywoływana jest dla mutexu nieblokująca metoda `try_lock()`
 - + jeśli blokada zostanie pozyskana metoda `owns_lock()` zwraca true, w przeciwnym wypadku zwraca false

std::unique_lock + std::timed_mutex

```
std::timed_mutex mutex;
std::unique_lock<std::timed_mutex> lock(mutex, std::try_to_lock);

// ...
if (!lock.owns_lock())
{
    int count = 0;
    do
    {
        std::cout << "Thread does not own a lock..." 
            << " Tries to acquire a mutex..." 
            << std::endl;
    } while(!lock.try_lock_for(std::chrono::seconds(1)));
}
```

Deadlock

Deadlock

sytuacja, w której co najmniej dwa różne wątki czekają na siebie nawzajem, więc żadny nie może się zakończyć

Deadlock

Deadlock

- Do zakleszczenia dochodzi, jeśli spełnione są cztery warunki:

Deadlock

- Do zakleszczenia dochodzi, jeśli spełnione są cztery warunki:
 1. Wzajemne wykluczenie

Deadlock

- Do zakleszczenia dochodzi, jeśli spełnione są cztery warunki:
 1. Wzajemne wykluczenie
 - + w danym czasie tylko jeden wątek może korzystać z zasobu

Deadlock

- Do zakleszczenia dochodzi, jeśli spełnione są cztery warunki:
 1. Wzajemne wykluczenie
 - + w danym czasie tylko jeden wątek może korzystać z zasobu
 2. Trzymanie zasobu i oczekiwanie

Deadlock

- Do zakleszczenia dochodzi, jeśli spełnione są cztery warunki:

1. Wzajemne wykluczenie

- + w danym czasie tylko jeden wątek może korzystać z zasobu

2. Trzymanie zasobu i oczekiwanie

- + wątek utrzymuje jeden z zasobów, ale do ukończenia pracy potrzebne jest także zablokowanie innego zasobu

Deadlock

- Do zakleszczenia dochodzi, jeśli spełnione są cztery warunki:

1. Wzajemne wykluczenie

- + w danym czasie tylko jeden wątek może korzystać z zasobu

2. Trzymanie zasobu i oczekiwanie

- + wątek utrzymuje jeden z zasobów, ale do ukończenia pracy potrzebne jest także zablokowanie innego zasobu

3. Cykliczne oczekiwanie

Deadlock

- Do zakleszczenia dochodzi, jeśli spełnione są cztery warunki:

1. Wzajemne wykluczenie

- + w danym czasie tylko jeden wątek może korzystać z zasobu

2. Trzymanie zasobu i oczekiwanie

- + wątek utrzymuje jeden z zasobów, ale do ukończenia pracy potrzebne jest także zablokowanie innego zasobu

3. Cykliczne oczekiwanie

- + wątki w taki sposób żądają dostępu do zasobów, że powstaje cykliczny graf skierowany

Deadlock

- Do zakleszczenia dochodzi, jeśli spełnione są cztery warunki:

1. Wzajemne wykluczenie

- + w danym czasie tylko jeden wątek może korzystać z zasobu

2. Trzymanie zasobu i oczekiwanie

- + wątek utrzymuje jeden z zasobów, ale do ukończenia pracy potrzebne jest także zablokowanie innego zasobu

3. Cykliczne oczekiwanie

- + wątki w taki sposób żądają dostępu do zasobów, że powstaje cykliczny graf skierowany

4. Brak wywłaszczenia zasobu

Deadlock

- Do zakleszczenia dochodzi, jeśli spełnione są cztery warunki:

1. Wzajemne wykluczenie

- + w danym czasie tylko jeden wątek może korzystać z zasobu

2. Trzymanie zasobu i oczekiwanie

- + wątek utrzymuje jeden z zasobów, ale do ukończenia pracy potrzebne jest także zablokowanie innego zasobu

3. Cykliczne oczekiwanie

- + wątki w taki sposób żądają dostępu do zasobów, że powstaje cykliczny graf skierowany

4. Brak wywłaszczenia zasobu

- + wątki dobrowolnie nie rezygnują z przydzielonych im zasobów, zwolnienie zasobów możliwe jest po zakończeniu zadania

Deadlock - przykład

- Klasa ze stanem wewnętrznym, chroniona muteksem
- Chcemy napisać operator porównania.

```
class X
{
    mutable std::mutex mtx_;
    int data_;
public:
    bool operator<(const X& other) const
    {
        std::lock_guard<std::mutex> lk(mtx_);
        std::lock_guard<std::mutex> lk(other.mtx_);
        return data_ < other.data_;
    }
};
```

Deadlock - przykład

- Klasa ze stanem wewnętrznym, chroniona muteksem
- Chcemy napisać operator porównania.

```
class X
{
    mutable std::mutex mtx_;
    int data_;
public:
    bool operator<(const X& other) const
    {
        std::lock_guard<std::mutex> lk(mtx_);
        std::lock_guard<std::mutex> lk(other.mtx_);
        return data_ < other.data_;
    }
};
```

Deadlock

Thread A

```
if(x1 < x2)
```

```
x1 mtx.lock()
```

```
x2 mtx.lock() ← deadlock
```

Thread B

```
if(x2 < x1)
```

```
x2 mtx.lock()
```

Zapobieganie zakleszczeniom - std::lock()

- Aby zminimalizować ryzyko należy pozyskiwać blokady zawsze w tej samej kolejności
- Lub użyć funkcji `std :: lock()`
 - + gwarantuje zablokowanie wszystkich muteksów bez zakleszczenia niezależnie od kolejności ich pozyskiwania
 - + wymaga przekazania jako parametrów opóźnionych blokad typu `std :: unique_lock`

std::lock()

```
bool X::operator< (const X& other) const
{
    std::unique_lock<std::mutex> l1(mtx_, std::defer_lock);
    std::unique_lock<std::mutex> l2(other.mtx_, std::defer_lock);

    std::lock(l1, l2); // avoiding deadlock

    return some_data < other.some_data;
}
```

std::lock()

```
bool X::operator< (const X& other) const
{
    std::unique_lock<std::mutex> l1(mtx_, std::defer_lock);
    std::unique_lock<std::mutex> l2(other.mtx_, std::defer_lock);

    std::lock(l1, l2); // avoiding deadlock

    return some_data < other.some_data;
}
```

std::lock()

```
bool X::operator< (const X& other) const
{
    std::unique_lock<std::mutex> l1(mtx_, std::defer_lock);
    std::unique_lock<std::mutex> l2(other.mtx_, std::defer_lock);

    std::lock(l1, l2); // avoiding deadlock

    return some_data < other.some_data;
}
```

Bezpieczne pozyskanie blokad

std::lock()

```
bool X::operator< (const X& other) const
{
    std::unique_lock<std::mutex> l1(mtx_, std::defer_lock);
    std::unique_lock<std::mutex> l2(other.mtx_, std::defer_lock);

    std::lock(l1, l2); // avoiding deadlock

    return some_data < other.some_data;
}
```

Zwolnienie blokad

INFOTRAINING⁺⁺

Synchronizacja za pomocą zdarzeń

Zdarzenia

- W programach wielowątkowych często zachodzi potrzeba zsynchronizowania pracy wielu wykonywanych współbieżnie zadań
- Dzieje się tak w sytuacji, gdy jeden z pracujących wątków osiąga punkt, w którym nie może wykonać następnych operacji, dopóki inne wątki nie zakończą swojej części pracy, przygotowując dane potrzebne do zakończenia zadania
- Fakt, że oczekiwane dane są dostępne jest określany jako **zdarzenie** (*event*).

Synchronizacja za pomocą zdarzeń

Oczekiwanie na zdarzenie może być zaimplementowane jako:

Synchronizacja za pomocą zdarzeń

Oczekiwanie na zdarzenie może być zaimplementowane jako:

- *busy wait* - oczekивание на зданиее implementedowane jest za pomocą петли do-while

Synchronizacja za pomocą zdarzeń

Oczekiwanie na zdarzenie może być zaimplementowane jako:

- *busy wait* - oczekивание на зданиее implementedowane jest za pomocą pętli do-while
- *idle wait* - wątek oczekujący на зданиее przechodzi do stanu uśpienia, w którym nie zużywa cyklów CPU

Busy waits - implementacja z mutexem

```
volatile bool data_ready;  
std::mutex mtx;
```

```
void set_data_ready()  
{  
    prepare_data();  
    std::unique_lock<std::mutex> lk(mtx);  
    data_ready = true;  
}
```

```
void task()  
{  
    bool ready_flag = data_ready;  
  
    do  
    {  
        std::lock_guard<std::mutex> lk{mtx};  
        ready_flag = data_ready;  
    }  
    while(!ready_flag);  
  
    process_data();  
}
```

Busy waits - implementacja z mutexem

```
volatile bool data_ready;  
std::mutex mtx;
```

```
void set_data_ready()  
{  
    prepare_data();  
    std::unique_lock<std::mutex> lk(mtx);  
    data_ready = true;  
}
```

```
void task()  
{  
    bool ready_flag = data_ready;  
  
    do  
    {  
        std::lock_guard<std::mutex> lk{mtx};  
        ready_flag = data_ready;  
    }  
    } while(!ready_flag);  
  
    process_data();  
}
```

Busy waits - implementacja z std::atomic

```
std::atomic<bool> data_ready;
```

```
void set_data_ready()
{
    prepare_data();
    data_ready = true;
}
```

```
void task()
{
    while(!data_ready)
    {
        //..
        std::this_thread::yield();
    }

    process_data();
}
```

Busy waits - implementacja z std::atomic

```
std::atomic<bool> data_ready;
```

```
void set_data_ready()
{
    prepare_data();
    data_ready = true;
}
```

```
void task()
{
    while(!data_ready)
    {
        //..
        std::this_thread::yield();
    }

    process_data();
}
```

Busy waits - implementacja z std::atomic

```
std::atomic<bool> data_ready;
```

```
void set_data_ready()
{
    prepare_data();
    data_ready.store(true,
        std::memory_order_release);
}
```

```
void task()
{
    while(!data_ready.load(
        std::memory_order_acquire))
    {
        // ..
        std::this_thread::yield();
    }

    process_data();
}
```

Busy waits - implementacja z std::atomic

```
std::atomic<bool> data_ready;
```

```
void set_data_ready()
{
    prepare_data();
    data_ready.store(true,
        std::memory_order_release);
}
```

```
void task()
{
    while(!data_ready.load(
        std::memory_order_acquire))
    {
        // ..
        std::this_thread::yield();
    }

    process_data();
}
```

Idle waits - zmienne warunkowe

- Aby uniknąć aktywnego odpytywania się o spełnienie określonego warunku, efektywniej jest zablokować oczekujący wątek do momentu zajścia zdarzenia (spełnienia oczekiwanej warunku)
- Mechanizm taki jest wykorzystany w implementacji **zmiennych warunkowych** (*condition variables*)

Zmienne warunkowe

- Biblioteka standardowa C++11 dostarcza dwie implementacje zmiennych warunkowych:
 - + `std :: condition_variable`
 - + `std :: condition_variable_any`

Zmienne warunkowe

- Obie klasy współpracują z muteksem, aby zapewnić prawidłową synchronizację wątków:
 - + `conditional_variable` współpracuje tylko z typem `std :: mutex`
 - + `conditional_variable_any` współpracuje z dowolnym typem muteksu

Idle waits

```
bool data_ready = false;  
std::mutex mtx_ready;  
std::condition_variable cv_ready;  
void process_data();
```

```
void set_data_ready()  
{  
    prepare_data();  
  
    {  
        std::lock_guard<std::mutex> lk(mtx_ready);  
        data_ready = true;  
    } // release lock  
  
    cv_ready.notify_one();  
}
```

```
void task()  
{  
    std::unique_lock<std::mutex> lk(mtx_ready);  
    while( !data_ready )  
    {  
        cv_ready.wait(lk);  
    }  
    lk.unlock();  
  
    process_data();  
}
```

Idle waits

```
bool data_ready = false;  
std::mutex mtx_ready;  
std::condition_variable cv_ready;  
void process_data();
```

```
void set_data_ready()  
{  
    prepare_data();  
  
    {  
        std::lock_guard<std::mutex> lk(mtx_ready);  
        data_ready = true;  
    } // release lock  
  
    cv_ready.notify_one();  
}
```

```
void task()  
{  
    std::unique_lock<std::mutex> lk(mtx_ready);  
    while(!data_ready)  
    {  
        cv_ready.wait(lk);  
    }  
    lk.unlock();  
  
    process_data();  
}
```

Idle waits

```
bool data_ready = false;  
std::mutex mtx_ready;  
std::condition_variable cv_ready;  
void process_data();
```

```
void set_data_ready()  
{  
    prepare_data();  
  
    {  
        std::lock_guard<std::mutex> lk(mtx_ready);  
        data_ready = true;  
    } // release lock  
  
    cv_ready.notify_one();  
}
```

```
void task()  
{  
    std::unique_lock<std::mutex> lk(mtx_ready);  
    while( !data_ready )  
    {  
        cv_ready.wait(lk);  
    }  
    lk.unlock();  
  
    process_data();  
}
```

Idle waits

```
bool data_ready = false;  
std::mutex mtx_ready;  
std::condition_variable cv_ready;  
void process_data();
```

```
void set_data_ready()  
{  
    prepare_data();  
    {  
        std::lock_guard<std::mutex> lk(mtx_ready);  
        data_ready = true;  
    } // release lock  
  
    cv_ready.notify_one();  
}
```

```
void task()  
{  
    std::unique_lock<std::mutex> lk(mtx_ready);  
    while( !data_ready )  
    {  
        cv_ready.wait(lk);  
    }  
    lk.unlock();  
  
    process_data();  
}
```

Spontaniczne wybudzenia *spurious wake-ups* - dlatego wait() musi być umieszczone w pętli while

Idle waits

```
bool data_ready = false;  
std::mutex mtx_ready;  
std::condition_variable cv_ready;  
void process_data();
```

```
void set_data_ready()  
{  
    prepare_data();  
    {  
        std::lock_guard<std::mutex> lk(mtx_ready);  
        data_ready = true;  
    } // release lock  
    cv_ready.notify_one();  
}
```

```
void task()  
{  
    std::unique_lock<std::mutex> lk(mtx_ready);  
    cv_ready.wait(lk, [] { return data_ready; });  
  
    lk.unlock();  
  
    process_data();  
}
```

Idle waits

```
bool data_ready = false;  
std::mutex mtx_ready;  
std::condition_variable cv_ready;  
void process_data();
```

```
void set_data_ready()  
{  
    prepare_data();  
    {  
        std::lock_guard<std::mutex> lk(mtx_ready);  
        data_ready = true;  
    } // release lock  
    cv_ready.notify_one();  
}
```

```
void task()  
{  
    std::unique_lock<std::mutex> lk(mtx_ready);  
    cv_ready.wait(lk, [] { return data_ready; });  
  
    lk.unlock();  
  
    process_data();  
}
```

Powiadamianie o zdarzeniu

- Powiadomienie o zdarzeniu może zostać zrealizowane przy pomocy dwóch metod:
 - + `void notify_one()` – odblokowuje jeden z wątków znajdujących się w stanie oczekiwania po uprzednim wywołaniu na obiekcie zmiennej warunkowej metody `wait()`
 - + `void notify_all()` – odblokowuje wszystkie wątki znajdujące się w stanie oczekiwania

Lokalna pamięć wątku

Lokalna pamięć wątku

- *Thread Local Storage* - dane, które należą do konkretnego wątku
- Każdy wątek może posiadać własny zestaw zmiennych TLS
- Odpowiednik statycznych (globalnych) zmiennych dla wątków
- Zastosowanie:
 - + `errno`
 - + funkcje reentrant
 - generatory liczb losowych
 - `strtok()`

Modyfikator `thread_local`

C++11 zapewnia dostęp do danych TLS za pomocą modyfikatora `thread_local`

```
namespace Unsafe
{
    double rand()
    {
        static int seed = 665; // shared state

        seed = (seed * IMUL + IADD) & MASK;
        return (seed * SCALE);
    }
}

int main()
{
    std::thread thd1([] {
        for (size_t i = 0; i < 100; ++i)
            std::cout << Unsafe::rand() << std::endl;
    });

    std::thread thd2([] {
        for (size_t i = 0; i < 100; ++i)
            std::cout << Unsafe::rand() << std::endl;
    });

    thd1.join();
    thd2.join();
}
```

```
namespace ThreadSafe
{
    double rand_double()
    {
        std::hash<std::thread::id> hasher;
        thread_local int thd_seed = 665 * hasher(std::this_thread::get_id());

        int result = (thd_seed * IMUL + IADD) & MASK;
        thd_seed = result;
        return (result * SCALE);
    }

    int main()
    {
        std::thread thd1([] {
            for (size_t i = 0; i < 100; ++i)
                std::cout << ThreadSafe::rand_double() << std::endl;
        });

        std::thread thd2([] {
            for (size_t i = 0; i < 100; ++i)
                std::cout << ThreadSafe::rand_double() << std::endl;
        });

        thd1.join();
        thd2.join();
    }
}
```

```
namespace ThreadSafe
{
    double rand_double()
    {
        std::hash<std::thread::id> hasher;
        thread_local int thd_seed = 665 * hasher(std::this_thread::get_id());

        int result = (thd_seed * IMUL + IADD) & MASK;
        thd_seed = result;
        return (result * SCALE);
    }
}

int main()
{
    std::thread thd1([] {
        for (size_t i = 0; i < 100; ++i)
            std::cout << ThreadSafe::rand_double() << std::endl;
    });

    std::thread thd2([] {
        for (size_t i = 0; i < 100; ++i)
            std::cout << ThreadSafe::rand_double() << std::endl;
    });

    thd1.join();
    thd2.join();
}
```


std::future

High-Level API for concurrency

Ilość wątków sprzętowych

- jest zwracana przez statyczną funkcję
`std :: thread :: hardware_concurrency`
- może zwrócić `0`, jeśli w systemie nie udostępnia takiej informacji

```
#include <thread>
#include <iostream>

int main()
{
    size_t threads_count =
        std::max(std::thread::hardware_concurrency(), 1u);

    assert(threads_count >= 0);
}
```

std::future

obsługuje możliwość odczytania wyniku wykonania funkcji (zwracanej wartości typu T lub wyjątku), która została uruchomiona w tym samym lub innym wątku

Stan współdzielony - shared state

obiekt synchronizujący, który przechowuje wynik (prawdopodobnie jeszcze nie obliczony), który może być wartością lub wyjątkiem

Shared state

jest współdzielony między:

- *asynchronous return object* - blokuje wykonanie programu, dopóki wynik nie jest gotowy - std :: future
- *asynchronous provider* - zapisuje wynik w obiekcie shared-state

Shared state

Thread#1

Asynchronous return
object

```
std :: future <int> f = ...  
  
auto result = f.get();
```

shared state

value

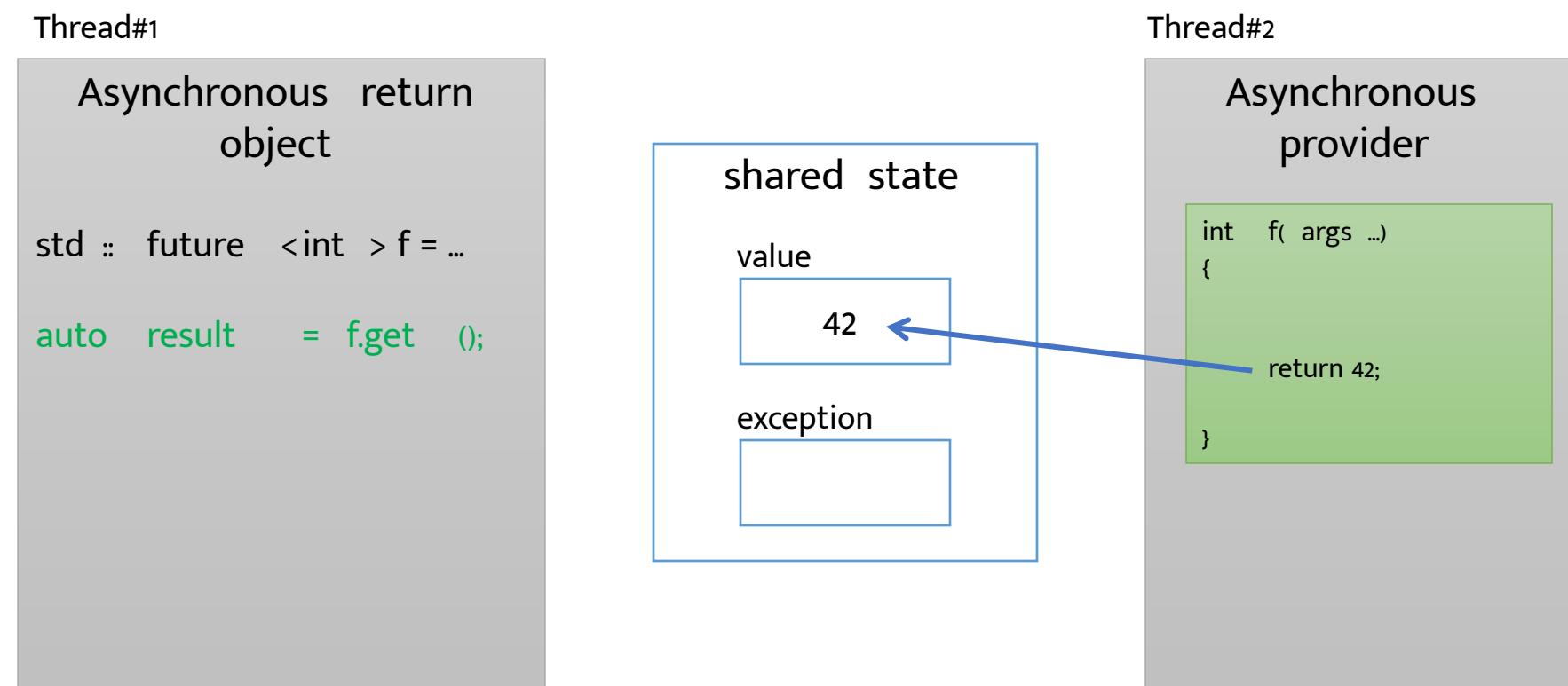
exception

Thread#2

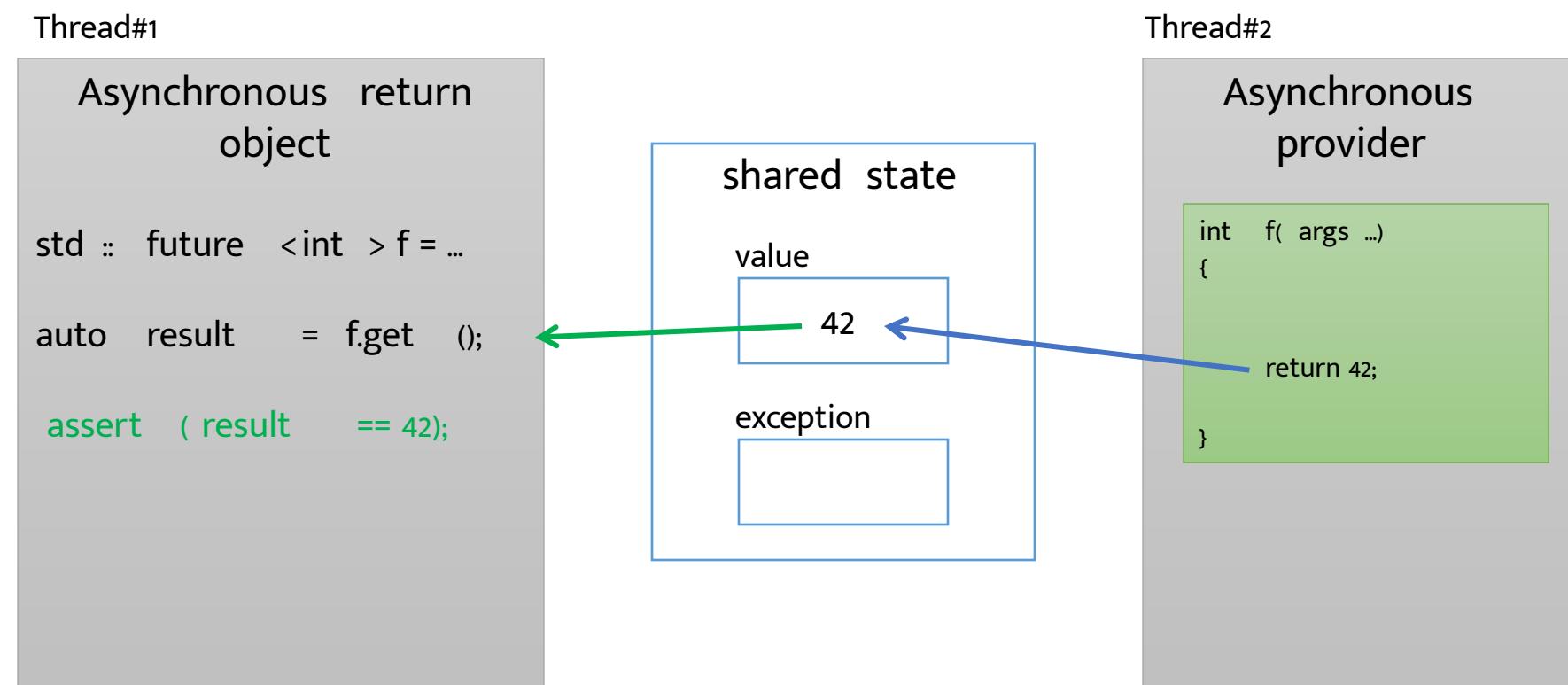
Asynchronous
provider

```
int f( args ... )  
{  
    ...  
}
```

Shared state



Shared state



std::future

- Do obsługi wartości typu **std :: future** wykorzystywane są cztery klasy:
 - + **std :: future<T>**
 - + **std :: shared_future<T>**
 - + **std :: promise<T>**
 - + **std :: packaged_task<T>**
 - + oraz funkcja **std :: async()**

std::future - interfejs klasy

std::future - interfejs klasy

- T get()

std::future - interfejs klasy

- T get()
 - + wstrzymuje bieżący wątek do momentu zakończenia asynchronicznej funkcji i następnie zwraca otrzymaną wartość lub rzuca przechowany w shared state wyjątek

std::future - interfejs klasy

- `T get()`
 - + wstrzymuje bieżący wątek do momentu zakończenia asynchronicznej funkcji i następnie zwraca otrzymaną wartość lub rzuca przechowany w shared state wyjątek
- `void wait() const`

std::future - interfejs klasy

- **T get()**
 - + wstrzymuje bieżący wątek do momentu zakończenia asynchronicznej funkcji i następnie zwraca otrzymaną wartość lub rzuca przechowany w shared state wyjątek
- **void wait() const**
 - + blokuje bieżący wątek dopóki wynik nie zostanie obliczony

std::future - interfejs klasy

- `T get()`
 - + wstrzymuje bieżący wątek do momentu zakończenia asynchronicznej funkcji i następnie zwraca otrzymaną wartość lub rzuca przechowany w shared state wyjątek
- `void wait() const`
 - + blokuje bieżący wątek dopóki wynik nie zostanie obliczony
- `future_status wait_for(const chrono::duration& timeout) const`

std::future - interfejs klasy

- `T get()`
 - + wstrzymuje bieżący wątek do momentu zakończenia asynchronicznej funkcji i następnie zwraca otrzymaną wartość lub rzuca przechowany w shared state wyjątek
- `void wait() const`
 - + blokuje bieżący wątek dopóki wynik nie zostanie obliczony
- `future_status wait_for(const chrono::duration& timeout) const`
 - + czeka przez określony interwał czasu aż wynik zostanie obliczony. Zwracany jest status, który może przyjąć jedną z trzech wartości:

std::future - interfejs klasy

- `T get()`
 - + wstrzymuje bieżący wątek do momentu zakończenia asynchronicznej funkcji i następnie zwraca otrzymaną wartość lub rzuca przechowany w shared state wyjątek
- `void wait() const`
 - + blokuje bieżący wątek dopóki wynik nie zostanie obliczony
- `future_status wait_for(const chrono::duration& timeout) const`
 - + czeka przez określony interwał czasu aż wynik zostanie obliczony. Zwracany jest status, który może przyjąć jedną z trzech wartości:
 - `future_status::deferred`

std::future - interfejs klasy

- `T get()`
 - + wstrzymuje bieżący wątek do momentu zakończenia asynchronicznej funkcji i następnie zwraca otrzymaną wartość lub rzuca przechowany w shared state wyjątek
- `void wait() const`
 - + blokuje bieżący wątek dopóki wynik nie zostanie obliczony
- `future_status wait_for(const chrono::duration& timeout) const`
 - + czeka przez określony interwał czasu aż wynik zostanie obliczony. Zwracany jest status, który może przyjąć jedną z trzech wartości:
 - `future_status::deferred`
 - `future_status::timeout`

std::future - interfejs klasy

- `T get()`
 - + wstrzymuje bieżący wątek do momentu zakończenia asynchronicznej funkcji i następnie zwraca otrzymaną wartość lub rzuca przechowany w shared state wyjątek
- `void wait() const`
 - + blokuje bieżący wątek dopóki wynik nie zostanie obliczony
- `future_status wait_for(const chrono::duration& timeout) const`
 - + czeka przez określony interwał czasu aż wynik zostanie obliczony. Zwracany jest status, który może przyjąć jedną z trzech wartości:
 - `future_status::deferred`
 - `future_status::timeout`
 - `future_status::ready`

std::future

Main thread

```
int main ()  
{  
    std::future<int> f1 = std::async( std::launch::async , f, 13);  
    std::future<int> f2 = std::async( std::launch::async , f, 665);  
  
    int r1 = f1.get(); // blocks until outcome is ready  
  
    int final_result = r1 + f2.get();  
}
```

Thread#1

```
int f( int n )  
{  
    return/ throw ...  
}
```

Thread#2

```
int f( int n )  
{  
    return/ throw ...  
}
```

std::future vs. std::shared_future

`std :: future<T>` - wynik przyszłego wywołania funkcji

`std :: shared_future<T>` - implementacja
współdzielonych wartości typu future

Wywołania asynchroniczne funkcji - std::async

Wywołania asynchroniczne funkcji - std::async

- `std::async(std::launch policy, Function&& f, Args&& ... args)`

Wywołania asynchroniczne funkcji - std::async

- `std::async(std::launch policy, Function&& f, Args&& ... args)`
 - + wywołuje funkcję `f` przekazując do niej argumenty `args`

Wywołania asynchroniczne funkcji - std::async

- `std::async(std::launch policy, Function&& f, Args&& ... args)`
 - + wywołuje funkcję `f` przekazując do niej argumenty `args`
 - + zwraca `std::future<std::result_of_t<Function(Args ...)>>`

Wywołania asynchroniczne funkcji - std::async

- Jako zadanie do funkcji `std :: async` można przekazać:
 - + wskaźnik do funkcji
 - + obiekt funkcyjny
 - + obiekt domknięcia
 - + obiekt + wskaźnik do metody

```
int fib(int n);
Gadget g{42, "ipad"};

std::future<int> f1 = std::async(std::launch::async, fib, 10);
std::future<int> f2 = std::async(std::launch::async, [] { return fib(20); });
std::future<int> f3 = std::async(std::launch::async, [g] { g.use(); });
std::future<int> f4 = std::async(std::launch::async, &Gadget::use, g);
```

std::async - argumenty funkcji

- Argumenty funkcji, które są
 - + *lvalue* - są kopiowane (również wtedy, gdy są to referencje)
 - + *rvalue* - są przenoszone
- Aby przekazać parametry przez referencję należy:
 - + `std :: ref` - przez referencję
 - + `std :: cref` - przez referencję do stałej
- Argumenty mogą też zostać przechwycone do obiektu domknięcia

std::async - argumenty funkcji

```
void work(int id, const std::string& path, std::atomic<int>& counter);  
  
int id = 1;  
const std::string path = "/dev/home/";  
std::atomic<int> counter{0};  
  
{  
    using namespace std;  
  
    auto f1 = async(launch::async,  
                    &work, id, cref(path), ref(counter));  
  
    auto f2 = async(launch::async,  
                    [id, &path, &counter] { work(id, path, counter); });  
}
```

Wyliczenie std::launch::async

specyfikuje sposób uruchomienia funkcji przekazanej do
std :: async()

```
enum class std::launch
{
    async,
    deferred
};
```

Wyliczenie std::launch::async

Wyliczenie std::launch::async

- std::launch::async

Wyliczenie std::launch::async

- std :: launch :: async
 - + funkcja jest uruchamiana w osobnym wątku (asynchronicznie)

Wyliczenie std::launch::async

- std :: launch :: async
 - + funkcja jest uruchamiana w osobnym wątku (asynchronicznie)
- std :: launch :: deferred

Wyliczenie std::launch::async

- std :: launch :: async
 - + funkcja jest uruchamiana w osobnym wątku (asynchronicznie)
- std :: launch :: deferred
 - + std::launch::deferred - nie tworzy osobnego wątku. Leniwie wykonuje funkcję f (wywołanie następuje w momencie pierwszego wywołania na obiekcie future get() lub wait())

std::async - destruktory obiektów std::future

jeśli obiekt typu `std :: future` został utworzony przez wywołanie `std :: async`, to jego destruktor **blokuje** wykonanie wątku, do czasu zakończenia wykonania funkcji asynchronicznej!

std::async - destruktory - WTF!!!

```
void save_to_file(const std::string& path, const std::string& msg);  
  
{  
    std::async(std::launch::async, &save_to_file, "a.log", "Do not do");  
    std::async(std::launch::async, &save_to_file, "b.log", "like THIS");  
}
```

std::async - destruktory - poprawna wersja

```
void save_to_file(const std::string& path, const std::string& msg);  
  
{  
    auto f1 = std::async(std::launch::async, &save_to_file, "a.log", "Do LIKE");  
    auto f2 = std::async(std::launch::async, &save_to_file, "b.log", "THIS");  
}
```

std::async - destruktory - poprawna wersja

```
void save_to_file(const std::string& path, const std::string& msg);  
  
{  
    auto f1 = std::async(std::launch::async, &save_to_file, "a.log", "Do LIKE");  
    auto f2 = std::async(std::launch::async, &save_to_file, "b.log", "THIS");  
}
```

Przypisanie do nazwanych zmiennych powoduje, że funkcje są rzeczywiście wykonywane asynchronicznie

std::packaged_task

jest klasą dostarczającą wrapper umożliwiający wywołanie funkcji lub funkторa i zapisujący wynik w *shared state*, który może być odczytany przez obiekt `std :: future`

`std::packaged_task`

`std::packaged_task`

- pozwala oddzielić:

`std::packaged_task`

- pozwala oddzielić:
 - + definicję zadania, które ma być wykonane

std::packaged_task

- pozwala oddzielić:
 - + definicję zadania, które ma być wykonane
 - + wywołanie zadania, które może zostać wykonane w osobnym wątku

std::packaged_task

- pozwala oddzielić:
 - + definicję zadania, które ma być wykonane
 - + wywołanie zadania, które może zostać wykonane w osobnym wątku
 - + przetworzenie wyników zadania

std::packaged_task

```
int calculate(int arg); // may throw std::runtime_error

std::packaged_task<int()> ptask1([] { return calculate(665); });

std::future<int> f1 = ptask1.get_future();

std::thread thd1(std::move(ptask1));
thd1.detach();

try
{
    auto result = f1.get();
}
catch(const std::runtime_error& e)
{
    // handle the exception
}
```

std::packaged_task

```
int calculate(int arg); // may throw std::runtime_error

std::packaged_task<int()> ptask1([] { return calculate(665); });

std::future<int> f1 = ptask1.get_future();

std::thread thd1(std::move(ptask1));
thd1.detach();

try
{
    auto result = f1.get();
}
catch(const std::runtime_error& e)
{
    // handle the exception
}
```

Przekazanie zadania do obiektu `std :: packaged_task`

std::packaged_task

```
int calculate(int arg); // may throw std::runtime_error

std::packaged_task<int()> ptask1([] { return calculate(665); });

std::future<int> f1 = ptask1.get_future();

std::thread thd1(std::move(ptask1));
thd1.detach();

try
{
    auto result = f1.get();
}
catch(const std::runtime_error& e)
{
    // handle the exception
}
```

Pozyskanie obiektu `std :: future`, który pozwoli odczytać wynik

std::packaged_task

```
int calculate(int arg); // may throw std::runtime_error

std::packaged_task<int()> ptask1([] { return calculate(665); });

std::future<int> f1 = ptask1.get_future();

std::thread thd1(std::move(ptask1));
thd1.detach();

try
{
    auto result = f1.get();
}
catch(const std::runtime_error& e)
{
    // handle the exception
}
```

Transfer std::packaged_task do wątku. Obiekty std::packaged_task są tylko przenaszalne

std::packaged_task

```
int calculate(int arg); // may throw std::runtime_error

std::packaged_task<int()> ptask1([] { return calculate(665); });

std::future<int> f1 = ptask1.get_future();

std::thread thd1(std::move(ptask1));
thd1.detach();

try
{
    auto result = f1.get();
}
catch(const std::runtime_error& e)
{
    // handle the exception
}
```

Blokujące oczekiwanie na wynik/wyjątek

std::promise

- asynchronous provider
- implementuje niskopoziomowy mechanizm przekazania asynchronicznego wyniku do obiektu `std :: future`, wykorzystując *shared state*
- wykorzystywany w sytuacji, gdy kod obliczający wartość nie może być zapakowany do `std :: packaged_task` lub przekazany do `std :: async`

std::promise

- konstruktor - tworzy pusty obiekt *shared state*
- `set_value(T)` - zapisuje wartość do stanu współdzielonego
- `set_exception(T)` - zapisuje wyjątek do stanu współdzielonego
- destruktor
 - + jeśli stan współdzielony jest gotowy, zwalnia go
 - + jeśli stan współdzielony nie jest gotowy, ustawia jako wyjątek instancję typu `std :: future_error` z kodem błędu `std :: future_error :: broken_promise` i zwalnia go

std::promise - kod

```
class DataSource {
    std::vector<std::promise<Data>> channels_;

public:
    DataSource(size_t channels_count) : channels_(channels_count)
    {}

    std::future<Data> get_future_channel(size_t channel_no) {
        return channels_.at(channel_no).get_future();
    }

    void read_data() {
        for(size_t i = 0; i < channels_.size(); ++i) {
            std::this_thread::sleep_for(250ms);
            try {
                auto result = calculate(i);
                channels_[i].set_value(result);
            }
            catch( ... ) {
                std::exception_ptr eptr = std::current_exception();
                channels_[i].set_exception(eptr);
            }
        }
    }
}
```

std::promise - kod

```
class DataSource {
    std::vector<std::promise<Data>> channels_;

public:
    DataSource(size_t channels_count) : channels_(channels_count)
    {}

    std::future<Data> get_future_channel(size_t channel_no) {
        return channels_.at(channel_no).get_future();
    }

    void read_data() {
        for(size_t i = 0; i < channels_.size(); ++i) {
            std::this_thread::sleep_for(250ms);
            try {
                auto result = calculate(i);
                channels_[i].set_value(result);
            }
            catch( ... ) {
                std::exception_ptr eptr = std::current_exception();
                channels_[i].set_exception(eptr);
            }
        }
    }
}
```

std::promise - kod

```
class DataSource {
    std::vector<std::promise<Data>> channels_;

public:
    DataSource(size_t channels_count) : channels_(channels_count)
    {}

    std::future<Data> get_future_channel(size_t channel_no) {
        return channels_.at(channel_no).get_future();
    }

    void read_data() {
        for(size_t i = 0; i < channels_.size(); ++i) {
            std::this_thread::sleep_for(250ms);
            try {
                auto result = calculate(i);
                channels_[i].set_value(result);
            }
            catch( ... ) {
                std::exception_ptr eptr = std::current_exception();
                channels_[i].set_exception(eptr);
            }
        }
    }
}
```

std::promise - kod

```
public:  
    DataSource(size_t channels_count) : channels_(channels_count)  
    {}  
  
    std::future<Data> get_future_channel(size_t channel_no) {  
        return channels_.at(channel_no).get_future();  
    }  
  
    void read_data() {  
        for(size_t i = 0; i < channels_.size(); ++i) {  
            std::this_thread::sleep_for(250ms);  
            try {  
                auto result = calculate(i);  
                channels_[i].set_value(result);  
            }  
            catch( ... ) {  
                std::exception_ptr eptr = std::current_exception();  
                channels_[i].set_exception(eptr);  
            }  
        }  
    }  
};
```

std::promise - kod

```
public:  
    DataSource(size_t channels_count) : channels_(channels_count)  
    {}  
  
    std::future<Data> get_future_channel(size_t channel_no) {  
        return channels_.at(channel_no).get_future();  
    }  
  
    void read_data() {  
        for(size_t i = 0; i < channels_.size(); ++i) {  
            std::this_thread::sleep_for(250ms);  
            try {  
                auto result = calculate(i);  
                channels_[i].set_value(result);  
            }  
            catch( ... ) {  
                std::exception_ptr eptr = std::current_exception();  
                channels_[i].set_exception(eptr);  
            }  
        }  
    }  
};
```

std::promise - kod

```
const size_t channel_count = 128;
DataSource data_source(channel_count);

std::future<int> future_channel_4 = data_source.get_future_channel(4);
std::future<int> future_channel_13 = data_source.get_future_channel(13);

std::thread thd{ds = std::move(data_source) mutable { ds.read_data(); }};
thd.detach();

assert(future_channel_4.get() == 16);
assert_throws(future_channel_13.get());
```

std::promise - kod

```
const size_t channel_count = 128;
DataSource data_source(channel_count);

std::future<int> future_channel_4 = data_source.get_future_channel(4);
std::future<int> future_channel_13 = data_source.get_future_channel(13);

std::thread thd{ds = std::move(data_source) mutable { ds.read_data(); }};
thd.detach();

assert(future_channel_4.get() == 16);
assert_throws(future_channel_13.get());
```

std::promise - kod

```
const size_t channel_count = 128;
DataSource data_source(channel_count);

std::future<int> future_channel_4 = data_source.get_future_channel(4);
std::future<int> future_channel_13 = data_source.get_future_channel(13);

std::thread thd{ds = std::move(data_source) mutable { ds.read_data(); }};
thd.detach();

assert(future_channel_4.get() == 16);
assert_throws(future_channel_13.get());
```

std::promise - kod

```
const size_t channel_count = 128;
DataSource data_source(channel_count);

std::future<int> future_channel_4 = data_source.get_future_channel(4);
std::future<int> future_channel_13 = data_source.get_future_channel(13);

std::thread thd{ds = std::move(data_source) mutable { ds.read_data(); }};
thd.detach();

assert(future_channel_4.get() == 16);
assert_throws(future_channel_13.get());
```

std::promise - kod

```
const size_t channel_count = 128;
DataSource data_source(channel_count);

std::future<int> future_channel_4 = data_source.get_future_channel(4);
std::future<int> future_channel_13 = data_source.get_future_channel(13);

std::thread thd{ds = std::move(data_source) mutable { ds.read_data(); }};
thd.detach();

assert(future_channel_4.get() == 16);
assert_throws(future_channel_13.get());
```