

# Verteilte Systeme

...für C++ Programmierer

Task-based Programming

by

Dr. Günter Kolousek

# Thread vs. Task

- ▶ thread-based programming
  - ▶ 'low-level'
  - ▶ kein Rückgabewert → Pointer/Referenz-Parameter!
  - ▶ Aufrufer kann keine Exceptions vom Thread abfangen → in Funktion wrappen!
  - ▶ Verwaltung & Synchronisation!
- ▶ task-based programming
  - ▶ 'high-level'
  - ▶ Abstraktion "Task": Arbeit, die erledigt werden soll
    - ▶ Programmierer erstellt Task
    - ▶ Library verwaltet Tasks (starten von Threads nach Bedarf: nicht zu viele, nicht zu wenige, load-balancing der Tasks → Threads → Cores)
  - ▶ C++: *tw.* Unterstützung
    - ▶ `promise`, `future`, `async`, `packaged_task`

# Promise & Future, Task

## ► Promise, Future

- Der Sender verspricht (promise) dem Empfänger in der Zukunft (future) einen

- einen Wert
- eine Exception
- eine Benachrichtigung

zu liefern

- d.h. ein Promise – Future Paar stellt einen Kommunikationskanal dar, um zwischen Threads zu kommunizieren → Entkopplung!

- vgl. ein thread kann keinen Wert mittels return zurückliefern

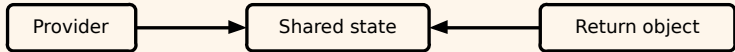
## ► Task

- Wrapper um ein "Callable" → asynchrones Aufrufen

# Promise & Future, Task – 2

- ▶ Provider: `async`, `promise`, `packaged_task`
  - ▶ setzt Wert in shared state
- ▶ Shared state
  - ▶ "ready", wenn Wert gesetzt
- ▶ Return object: `future`, `shared_future`
  - ▶ liest Wert aus shared state
  - ▶ ist `valid()`, wenn mit shared state verbunden
  - ▶ `get()`
    - ▶ liefert Wert von shared state
    - ▶ danach nicht mehr shared state verbunden
  - ▶ `wait()`: warten bis "ready" → Benachrichtigung
- ▶ `future`, `promise`, `packaged_task` können
  - ▶ nicht kopiert werden
  - ▶ verschoben werden

# Promise & Future, Task – 3



- ▶ Empfängerseite
- ▶ wartet auf ein Ergebnis, das in der Zukunft vorliegen wird (Wert, Exception, Benachrichtigung)
- ▶ `future ...` wird irgendwann "ready"
  - ▶ `get()` liefert Ergebnis (blockiert bis "ready")
    - ▶ Exception vom Thread wird wieder geworfen!
  - ▶ `wait()`, `wait_for()`, `wait_until()` wartet auf Ergebnis
    - ▶ d.h. bis "ready"
  - ▶ nicht thread-safe

# future – 2

- ▶ `async` startet eine Funktion asynchron und liefert ein `future` zurück
  - ▶ entweder in einem eigenem Thread oder nicht
    - ▶ `std::launch::async ...` neuer Thread
    - ▶ `std::launch::deferred ...` im aktuellen Thread, wenn `wait`,... oder `get` aufgerufen wird
    - ▶ `std::launch::async` | `std::launch::deferred ...` abhängig von der Implementierung und u.U. den verfügbaren Ressourcen → Defaultargument!
  - ▶ Parameterbehandlung wie bei `thread`
  - ▶ Ein `Promise` wird implizit erzeugt, aber für den Benutzer nicht sichtbar

# future - 3

```
#include <iostream> // future.cpp
#include <future>
using namespace std;
double calc_pi() {
    cout << "calculating 10**30 digits of pi..." <<
    this_thread::sleep_for(3s);
    return 3.1415926; }
int main() {
    future<double> pi{async(calc_pi)};
    cout << "doing something else..." << endl;
    cout << pi.get() << endl; }
```

doing something else...

calculating next 10\*\*30 digits of pi...

3.14159



# future - 4

```
#include <iostream> // future2.cpp
#include <future>
using namespace std;
void doit() { cout << "***" << endl;
              this_thread::sleep_for(3s); }
int main() { // serialize main with other thread
             future<void> other{async(launch::async, doit)};
             cout << "doing something else... " << endl;
             cout << "waiting for other thread... " << endl;
             other.wait();
             cout << "done" << endl; }
```

doing something else... \*\*\*

waiting for other thread...

done

# future – 5

```
#include <iostream>    // fire_forget_future.cpp
#include <chrono>
#include <future>
#include <thread>
using namespace std;
int main() {
    async(launch::async, [] {
        this_thread::sleep_for(chrono::seconds(2));
        cout << "first thread" << endl;
    });
    async(launch::async, [] {
        this_thread::sleep_for(chrono::seconds(1));
        cout << "second thread" << endl;
    });
    cout << "main thread" << endl;
}
```

# future – 6

Ausgabe:

```
first thread  
second thread  
main thread
```

Warum?

# future – 6

Ausgabe:

```
first thread  
second thread  
main thread
```

Warum?

- ▶ fire and forget futures
  - ▶ in dieser Form nicht realisierbar

# future – 6

Ausgabe:

```
first thread  
second thread  
main thread
```

Warum?

- ▶ fire and forget futures
  - ▶ in dieser Form nicht realisierbar
- ▶ Destruktor von Future
  - ▶ wartet auf Beendigung der Operation

# future – 6

Ausgabe:

```
first thread  
second thread  
main thread
```

Warum?

- ▶ fire and forget futures
  - ▶ in dieser Form nicht realisierbar
- ▶ Destruktor von Future
  - ▶ wartet auf Beendigung der Operation
- ▶ Rückgabewert von `async` = temporäres Objekt (Future!)
  - ▶ lebt bis Ende des vollständigen Ausdrucks

Lösungen?

# future – 6

Ausgabe:

```
first thread  
second thread  
main thread
```

Warum?

- ▶ fire and forget futures
  - ▶ in dieser Form nicht realisierbar
- ▶ Destruktor von Future
  - ▶ wartet auf Beendigung der Operation
- ▶ Rückgabewert von `async` = temporäres Objekt (Future!)
  - ▶ lebt bis Ende des vollständigen Ausdrucks

Lösungen?

→ Variable definieren ;-): `auto first = async(...);`

# future – 6

Ausgabe:

```
first thread  
second thread  
main thread
```

Warum?

- ▶ fire and forget futures
  - ▶ in dieser Form nicht realisierbar
- ▶ Destruktor von Future
  - ▶ wartet auf Beendigung der Operation
- ▶ Rückgabewert von `async` = temporäres Objekt (Future!)
  - ▶ lebt bis Ende des vollständigen Ausdrucks

Lösungen?

- Variable definieren ;-) : `auto first = async(...);`
- Thread erzeugen und `detach()`



# future - 7

```
#include <iostream>    // future_wait_for.cpp
#include <chrono>
#include <future>
#include <thread>
#include <algorithm>    // accumulate
using namespace std;
int accumulate_block(int* data, size_t count) {
    this_thread::sleep_for(3s);
    return accumulate(data, data + count, 0);
}
```

# future – 8

```
int main(int argc, const char** argv) {  
    vector<int> v{1, 2, 3, 4, 5, 6, 7, 8};  
    future<int> acc = std::async(launch::async,  
        accumulate_block, v.data(), v.size());  
    while (acc.wait_for(chrono::seconds(1)) !=  
        future_status::ready) {  
        cout << "...still not ready\n";  
    }  
    cout << "result: " << acc.get() << "\n";  
}
```

```
...still not ready  
...still not ready  
...still not ready  
result: 36
```

# shared\_future

- ▶ Zugriff von mehreren Threads auf das Ergebnis
  - ▶ man erhält ein `shared_future` mittels
    - ▶ `future<int> f; shared_future<int> sf{f.share()};` oder
    - ▶ `shared_future<int> sf{future<int>{}}`
- (in dieser Form natürlich sinnlos, da `future<int>{}` nicht mit shared state verbunden und daher *nicht* "valid")

# shared\_future - 2

```
#include <iostream>    // shared_future.cpp
#include <future>
using namespace std;
int calc_it() { this_thread::sleep_for(1s);
    return 42; }
void use_it(shared_future<int> f) {
    cout << f.get() << endl; }
int main() {    // serialize main with other thread
    auto other{shared_future<int>{async(
        launch::async, calc_it)}};
    // also: other = async(...).share()
    thread use_it_trd{use_it, other};    // -> 42
    thread use_it2_trd{use_it, other};    // -> 42
    cout << "doing something else..." << endl;
    use_it_trd.join();
    use_it2_trd.join(); }
```

# promise

- ▶ Senderseite
- ▶ setzt Ergebnis (Wert, Exception, Benachrichtigung)
- ▶ `promise`
  - ▶ `promise<T> ...` legt ein `promise` mit dem angegebenen Typ `T` für das Ergebnis an
  - ▶ `get_future()` ... liefert `future`
  - ▶ `set_value(VALUE)`
    - ▶ `set_value()`, wenn `promise<void>` → `get_future().wait()`
  - ▶ `set_exception(std::exception_ptr)`
  - ▶ `set_exception_at_thread_exit(std::exception_ptr)`
    - shared-ownership smart pointer
  - ▶ `set_value_at_thread_exit(VALUE), set_value_at_thread_exit()`
    - ▶ Wert erst am Threadende verfügbar

# promise - 2

```
#include <iostream>    // promise.cpp
#include <thread>
#include <future>
#include <random>
using namespace std;
int main() {
    random_device rd;
    mt19937 gen{rd()};
    uniform_int_distribution<> dis{0, 100};
    promise<int> result;
```

# promise – 3

```
thread calc{[&]() {  
    this_thread::sleep_for(1s);  
    if (dis(gen) > 50)  
        result.set_exception(  
            make_exception_ptr(logic_error("x")))  
    else  
        result.set_value(42);  
}};  
cout << result.get_future().get() << endl;  
calc.join();  
}
```

# promise – 4

Entweder

terminate called after throwing an instance of 'std  
    what(): x

oder einfach

42



# promise – 4

Entweder

`terminate` called after throwing an instance of 'std  
`what(): x`

oder einfach

42

Beachte: nur einmalige Verwendung eines Promise-Future Paares!

# promise – 5

Benachrichtigung mittels `promise<void>`:

```
#include <iostream>    // notification.cpp
#include <thread>
#include <future>
using namespace std;
int main() {
    promise<void> go;
    auto go_future = go.get_future();
    thread worker{[&go]() {
        this_thread::sleep_for(1s);
        go.set_value();
    }};
    go_future.wait();
    cout << "finished working" << endl;
    worker.join(); }
```

# Promise & Future

- ▶ Promise kann *nicht* zurückgesetzt werden!
  - ▶ d.h. setzen des Promise und auslesen des Wertes bedeutet, dass dieses Paar nicht mehr weiterverwendet werden kann.
- ▶ → condition variable

# Task – packaged\_task

```
#include <iostream>    // task.cpp
#include <thread>
#include <future>    // packaged_task
#include <deque>
#include <vector>
using namespace std;
int main() {
    deque<packaged_task<int(int, int)>> tasks{};
    for (int i{}; i < 10; ++i)
        tasks.push_back(
            packaged_task<int(int,int)>(
                [](int i, int j) { return i + j; }));

    vector<future<int>> results;
```

# Task – packaged\_task – 2

```
while (not tasks.empty()) {  
    auto t = move(tasks.front());  
    tasks.pop_front();  
    results.push_back(t.get_future());  
    thread thd{move(t), 1, 2};  
    thd.detach();  
}
```

```
int res{};  
for (int i{}; i < 10; ++i) {  
    res += results[i].get();  
}  
cout << res << endl;    // -> 30  
}
```