Verteilte Systeme

...für C++ Programmierer

Threads 2

by

Dr. Günter Kolousek

9. Oktober 2018

Threads und ’callable types’

- Thread kann jeden beliebigen ’callable’ Typ bei Initialisierung als Parameter bekommen
- Dies sind:
  - Normale Funktionen (per Funktionsname, per Pointer auf Funktion)
  - Lambda-Ausdrücke
  - `std::function` - Objekte (aus `<functional>`)
  - Klassen mit überladenen Operator `operator()`.

Lambda-Thread

```
#include <iostream>  // lambdathread.cpp
#include <thread>
using namespace std;

int main() {
    thread t{[]() {
                cout << ”lambda thread” << endl; }};
    cout << ”main thread” << endl;
    t.join();
}
```

Thread mit Wertparameter

```
#include <iostream>  // valparthread.cpp
#include <thread>
using namespace std;

int main() {
    auto outfunc = [](string const& msg) {
        cout << ”lambda ” << msg << endl;
    };
    thread t{outfunc,
             ”thread”};
    cout << ”main thread” << endl;
    t.join();
}
```

```
main threadlambda
thread
```

Thread mit Wertparameter - 2

```
#include <iostream>  // valparthread2.cpp
#include <thread>
using namespace std;
void f() {
    char buffer[1024]{"very long string..."};
    auto outfunc{[](string msg) {
        cout << "lambda " << msg << endl; }};
    thread t{outfunc, buffer};
    t.detach();
}
int main() {
    f();   this_thread::sleep_for(10ms);
}
```

unsicher!!!

Thread mit Wertparameter - 2

```
#include <iostream>  // valparthread2.cpp
#include <thread>
using namespace std;
void f() {
    char buffer[1024]{”very long string...”};
    auto outfunc{[](string msg) {
        cout << ”lambda ” << msg << endl; }};
    thread t{outfunc, buffer};
    t.detach();
}
int main() {
    f();   this_thread::sleep_for(10ms);
}
```

unsicher!!! `buffer` wird als `char*` übergeben (per value) und im Kontext des Threads wird ein `string` Objekt erzeugt...

Thread mit Wertparameter - 2

```
#include <iostream>  // valparthread2.cpp
#include <thread>
using namespace std;
void f() {
    char buffer[1024]{"very long string..."};
    auto outfunc{[](string msg) {
        cout << "lambda " << msg << endl; }};
    thread t{outfunc, buffer};
    t.detach();
}
int main() {
    f();   this_thread::sleep_for(10ms);
}
```

unsicher!!! `buffer` wird als `char*` übergeben (per value) und im Kontext des Threads wird ein `string` Objekt erzeugt... besser:
`thread t{outfunc, string{buffer}};`

Thread mit Referenzparameter

```cpp
#include <iostream>  // refparthread.cpp
#include <thread>
#include <functional>  // ref
using namespace std;

int main() {
    int n{};
    auto incrfunc{[](int& n) { ++n; }};
    // use std::ref, otherwise per-value!
    //   generates 'reference_wrapper'
    thread t{incrfunc, ref(n)};
    t.join();
    cout << n << endl; }
```

```
1
```

Aber Achtung, wenn Speicherobjekt nicht mehr vorhanden…

Thread mit Referenzparameter – 2

```
#include <iostream>  // refparthread2.cpp
#include <thread>
#include <functional>  // ref
using namespace std;
using namespace literals;

struct Distance {
    double len{};
    ~Distance() { cout << ”dstor” << endl; }
};
```

Thread mit Referenzparameter – 3

```
int main() {
    Distance* pd;
    {
        Distance d{};
        pd = &d;
        auto incrfunc{[](Distance& d) {
            this_thread::sleep_for(1s); ++d.len; }};
        thread t(incrfunc, ref(d)); // be careful!
        t.detach();
    }
    this_thread::sleep_for(2s);
    cout << "trying to access pd->len..." << endl;
}
```

Thread mit Referenzparameter – 3

```
int main() {
    Distance* pd;
    {
        Distance d{};
        pd = &d;
        auto incrfunc{[](Distance& d) {
            this_thread::sleep_for(1s); ++d.len; }};
        thread t(incrfunc, ref(d)); // be careful!
        t.detach();
    }
    this_thread::sleep_for(2s);
    cout << "trying to access pd->len..." << endl;
}
```

```
dstor
trying to access pd->len...
```

Thread - moveable Argument

```
#include <iostream>  // movparthread.cpp
#include <thread>
using namespace std;
struct Ressource {
    int value{};
    int id{};
    Ressource(int value_) : value{value_} {}
    Ressource(Ressource&& o) {
        value = o.value;
        id = o.id + 1;
        cout<< ”from ”<< o.id<< ” to ”<< id<< endl;
        o.value = 0;
    }
    ~Ressource() { cout << ”dtor: ” << id <<
                   ” value: ” << value << endl; }
};
```

Thread - moveable Argument – 2

```
int main() {
    thread t1{[](Ressource&& res) { cout <<
      "t1: " << res.value << endl; },
      Ressource{42}};
    t1.join();
}
```

```
from 0 to 1
from 1 to 2
dtor of id 1 value: 0
dtor of id 0 value: 0
t1: 42
dtor of id 2 value: 42
```

Thread ID

```
#include <iostream>  // threadid.cpp
#include <thread>
using namespace std;
using namespace std::literals;
int main() {
    thread t{[]() { this_thread::sleep_for(1s); }};
    // type std::thread::id
    // comparable (<,<=,==,...) & storable in map
    cout << this_thread::get_id() << ' ';
    cout << t.get_id() << endl;
    t.join();
    thread t2;  // no hw thread associated
    cout << t2.get_id() << endl; }

140524781053760 140524781049600
thread::id of a non-executing thread
```

Thread-Affinity (Linux)

```cpp
#include <iostream>  // affinity.cpp
#include <vector>
#include <thread>
#include <mutex>
#include <sched.h>  // sched_getcpu()
using namespace std;
using namespace std::literals;
```

Thread-Affinity (Linux) – 2

```
int main(int argc, const char** argv) {
    constexpr unsigned num_threads = 4;
    mutex mtx;
    vector<thread> threads(num_threads);
    for (unsigned i = 0; i < num_threads; ++i) {
        threads[i] = thread([&mtx, i] {
          while (true) {
              {
                  lock_guard<mutex> iolock(mtx);
                  cout<<"Thread #"<<i<<": on CPU "
                    <<sched_getcpu()<<"\n";
              }
              this_thread::sleep_for(900ms);
          } }); }
    for (auto& t : threads) t.join();
}
```

Thread-Affinity (Linux) – 3

Mögliche Ausgabe des Programmes `affinity`

```
Thread #1: on CPU 2
Thread #0: on CPU 0
Thread #2: on CPU 1
Thread #3: on CPU 0
Thread #1: on CPU 2
Thread #0: on CPU 3
Thread #3: on CPU 1
Thread #2: on CPU 2
...
```

Thread-Affinity (Linux) – 4

Nochmals mit nur den Kernen 2 und 3:

```
$ taskset -c 2,3 affinity

Thread #0: on CPU 2
Thread #1: on CPU 2
Thread #2: on CPU 3
Thread #3: on CPU 2
Thread #0: on CPU 2
Thread #2: on CPU 3
Thread #1: on CPU 3
Thread #3: on CPU 2
...
```

Setzen der Thread-Affinity (Linux)

```cpp
#include <pthread.h>

#include <iostream>  // setaffinity.cpp
#include <vector>
#include <thread>
#include <mutex>
#include <sched.h>  //

using namespace std;
using namespace std::literals;
```

Setzen der Thread-Affinity – 2

```
int main(int argc, const char** argv) {
    constexpr unsigned num_threads = 4;
    mutex mtx;
    vector<thread> threads(num_threads);
    for (unsigned i{}; i < num_threads; ++i) {
        threads[i] = thread([&mtx, i] {
          this_thread::sleep_for(20ms);
          while (true) {
            {
                lock_guard<mutex> iolock(mtx);
                cout << "Thread #" << i
                     << ": on CPU "
                     << sched_getcpu() << "\n";
            }
            this_thread::sleep_for(900ms);
          }});
```

Setzen der Thread-Affinity – 3

```
            // cpu_set_t represents a set of CPUs.
            // Clear it and mark only CPU i as set.
            cpu_set_t cpuset;
            CPU_ZERO(&cpuset);
            CPU_SET(i, &cpuset);
            int rc = pthread_setaffinity_np(
                threads[i].native_handle(),
                sizeof(cpu_set_t), &cpuset);
            if (rc != 0) {
              cerr <<
                "Error calling pthread_setaffinity_np:
                << rc << "\n";
            }
        }
        for (auto& t : threads) t.join();
}
```

Setzen der Thread-Affinity – 4

Mögliche Ausgabe:

```
Thread #0: on CPU 0
Thread #2: on CPU 2
Thread #3: on CPU 3
Thread #1: on CPU 1
Thread #0: on CPU 0
Thread #2: on CPU 2
Thread #1: on CPU 1
Thread #3: on CPU 3
...
```