

Verteilte Systeme

...für C++ Programmierer

Threads

by

Dr. Günter Kolousek

Threads

- ▶ Grundbausteine paralleler Software
- ▶ aus Sicht des Betriebssystems: kleinste Einheit der Parallelität
- ▶ ein Prozess besteht aus 1 oder mehreren Threads
- ▶ alle Threads innerhalb eines Prozesses können auf alle Ressourcen des Prozesses zugreifen
- ▶ jeder Thread hat separat Stack, Registerinhalte, Schedulingparameter (Priorität, Affinität,...). Außerdem: thread-lokale Daten
- ▶ Ab C++11: `thread`,...!
 - ▶ früher POSIX (`pthreads`), Windows API, Qt (`QThread`), Poco (`Thread`)

Prozess vs. Thread

- ▶ Prozess
 - ▶ Vorteile
 - ▶ Nichtbeeinflussung anderer Prozesse
 - ▶ Rechte, Abrechnung
 - ▶ Nachteile
 - ▶ Anlegen ressourcenintensiv (Zeit, Speicher)
 - ▶ Context Switch zeitintensiv: CPU Kontext (Register, Programmzähler, Stackpointer,...), MMU Register, Swapping, CPU Abrechnungen,...
- ▶ Thread
 - ▶ Vorteile
 - ▶ geringerer Overhead beim Anlegen & Context Switch
 - ▶ Zugriff auf Daten und offene Dateien,...
 - ▶ Nachteile
 - ▶ Beeinflussung durch andere Threads

Einsatz auf Single-Core Systemen

- ▶ Asynchrones Warten (aus globaler Sicht)
 - ▶ Überbrückung der Wartezeit bei Ein- aber auch Ausgaben
- ▶ Responsivität der Benutzeroberfläche
 - ▶ Bedienbarkeit trotz "rechenintensiver" Applikation
- ▶ Trennung der Teilaufgaben
 - ▶ Aufsplittung von unabhängigen Aktivitäten
 - ▶ z.B. Musik, Kommunikation, Darstellung,... in einem Computerspiel

Übersetzen von Threads

- ▶ Es ist eine Threadbibliothek beim Linken anzugeben!
- ▶ Verwende z.B. folgenden Befehl:

```
g++ -o go -std=c++1z  
    -Wsizeof-array-argument -Wall  
    -Wextra -lpthread
```
- ▶ Bzw. füge folgende CMake-Anweisungen hinzu

```
find_package(Threads)  
# replace xxx with name of the executable  
target_link_libraries(xxx -lpthread)
```

Starten von Threads

```
#include <iostream> // thread.cpp
#include <thread>
using namespace std;
void f() {
    for (int i=0; i < 5; ++i) {
        cout << "B";
    }
}

int main() {
    thread t{f}; // thread starts HERE
    for (int i{0}; i < 5; ++i) {
        cout << "A";
    }
}
```

Starten von Threads – 2

- ▶ Mögliche Ausgabe:

`terminate called without an active exception`

`AAAAABBBBBBfish: 'go' terminated by signal SIGABR`

- ▶ Warum?

- ▶ Scheduler!
- ▶ Beenden des Hauptthreads → Destruktur von thread → `terminate`!
- ▶ gepufferte Ausgabe, aber kein "flushen"!

Pufferung von cout

Nach jeder Schleife einfügen:

```
// thread2.cpp  
cout << flush;
```

► Mögliche Ausgabe:

```
AAAAAterminate called without an active exception  
BBBBBBBBBBfish: Job 2, 'go' terminated by signal
```

► Warum?

- Scheduler?
- cout thread-safe (keine data races), aber "beliebige" Reihenfolge der Ausgabe
- mehrere Bs auf Grund von terminate

Scheduler & sleep_for

In for nach jeder Ausgabe einfügen:

```
// thread3.cpp
```

```
this_thread::sleep_for(chrono::milliseconds(10));
```

► Mögliche Ausgabe:

```
ABABBABABAterminate called without an active exception
```

```
fish: 'go' terminated by signal SIGABRT (Abbruch)
```

join

Funktion `f()` wie gehabt, `main` wie folgt:

```
int main() {  
    thread t{f};  
    cout << t.joinable() << endl;  
    for (int i{0}; i < 5; ++i) {  
        cout << "A";  
        this_thread::sleep_for(  
            chrono::milliseconds(10));  
    }  
    cout << endl;  
    t.join();  
    cout << t.joinable() << endl;  
}
```

join - 2

- ▶ Mögliche Ausgabe

1

ABABBABABA

0

- ▶ Der Hauptthread wartet auf die Beendigung des gestarteten Thread
- ▶ `terminate` wird *nicht* mehr durch die C++ Runtime aufgerufen, da auf den "joinable" Thread jetzt gewartet wurde. Nach `join` ist der Thread nicht mehr "joinable"!

this_thread::get_id()

Mehrere Threads... Funktion f() wie gehabt, main wie folgt:

```
int main() { // join2.cpp
    vector<thread> threads;

    for(int i = 0; i < 5; ++i){
        threads.push_back(std::thread([](){
            cout << "Hello from thread "
                << this_thread::get_id() << endl;
        }));
    }

    for(auto& thread : threads){
        thread.join();
    }
}
```

this_thread::get_id() - 2

Hello from thread 3073993536

Hello from thread 3038767936

Hello from thread Hello from thread 3047160640
3057208128

Hello from thread 3065600832

join und Exceptions

```
#include <iostream> // join2.cpp
#include <thread>
using namespace std;
void f() {
    throw logic_error{"something failed..."};
}
int main() {
    thread t{f};
    t.join();
}
```

join und Exceptions

```
#include <iostream> // join2.cpp
#include <thread>
using namespace std;
void f() {
    throw logic_error{"something failed..."};
}
int main() {
    thread t{f};
    t.join();
}
```

terminate called after throwing an instance of 'std
what(): something failed...
fish: 'go' terminated by signal SIGABRT (Abbruch)

join und Exceptions

```
#include <iostream> // join2.cpp
#include <thread>
using namespace std;
void f() {
    throw logic_error{"something failed..."};
}
int main() {
    thread t{f};
    t.join();
}
```

terminate called after throwing an instance of 'std::
what(): something failed...
fish: 'go' terminated by signal SIGABRT (Abbruch)

Alle Exceptions innerhalb eines Threads abfangen!

join und Exceptions - 2

```
#include <iostream>    // join3.cpp
#include <thread>
using namespace std;
void g() {
    this_thread::sleep_for(chrono::seconds(1));
}
int inverse(int x) {
    if (x == 0) throw logic_error{"div by zero"};
    else return 1 / x;
}
void f() { // may throw an exception
    thread t{g};
    cout << inverse(0) << endl;
    t.join();
}
int main() { try { f(); } catch (...) { }}
```

join und Exceptions - 3

Bricht mit

`terminate` called without an active exception

ab! Warum?

join und Exceptions - 3

Bricht mit

`terminate` called without an active exception

ab! Warum?

`try` und `catch` um Aufruf von `inverse`, aber Destruktor von `t` wird `terminate` aufrufen, da `join` nicht aufgerufen wurde! Was ist zu tun?

join und Exceptions - 3

Bricht mit

`terminate` called without an active exception

ab! Warum?

`try` und `catch` um Aufruf von `inverse`, aber Destruktor von `t` wird `terminate` aufrufen, da `join` nicht aufgerufen wurde! Was ist zu tun?

Benötigt wird ein Wächter, der sich um den Aufruf von `join` kümmert.

Das kann mittels RAI (Resource Acquisition Is Initialization) erreicht werden.

join-3

```
class thread_guard { // join3.cpp
    thread& t;
public:
    explicit thread_guard(thread& t_)
        : t{t_} {}
    ~thread_guard() {
        if (t.joinable()) {
            t.join();
        }
    }
};

void f() { // may throw an exception
    thread t{g};
    thread_guard tg{t};
    cout << inverse(0) << endl;
}
```

join - 4

Aber auch diese Lösung hat in gewisser Weise Nachteile...

- ▶ Thread wird per Referenz an `thread_guard` übergeben. Damit besteht wieder die Möglichkeit, dass das `thread_guard` Objekt den Thread "überlebt" (d.h. das Thread-Objekt davor zerstört wird).
- ▶ Es könnte an anderer Stelle auf den Thread gewartet werden (mittels `join`) oder dieser in den Hintergrund geschickt werden (mittels `detach`) (siehe später), obwohl die Idee ist, dass das `thread_guard` Objekt die Eigentümerrolle übernommen hat.

Lösung wäre ein Thread, der wie innerhalb eines Gültigkeitsbereiches (scope) existiert...

scoped thread

```
#include <iostream>    // scoped.cpp
#include <thread>
using namespace std;
class scoped_thread {
    thread t;
public:
    explicit scoped_thread(thread t_)
        : t{move(t_)} {
        if (!t.joinable())
            throw logic_error("Not joinable");
    }
    ~scoped_thread() { t.join(); }
    scoped_thread(scoped_thread const&) = delete;
    scoped_thread& operator=(
        scoped_thread const&) = delete;
};
```

scoped thread – 2

```
void g() {  
    this_thread::sleep_for(chrono::seconds(1));  
}  
  
int inverse(int x) {  
    if (x == 0)  
        throw logic_error("div by zero");  
    else  
        return 1 / x; }  
  
void f() { // may throw an exception  
    // g() and inverse(int) like before  
    scoped_thread t{thread{g}};  
    cout << inverse(0) << endl;  
}  
// main() like before
```


scoped thread – 3

Auf das eigentliche Thread-Objekt kann nicht mehr gewartet werden...

```
void f() { // may throw an exception  
    thread t{g};  
    scoped_thread st{move(t)};//dont do it this way  
    cout << t.joinable() << endl; // -> 0  
    cout << inverse(0) << endl;  
}  
// main() like before
```

detach

```
#include <iostream> // detach.cpp
#include <thread>
using namespace std;
using namespace std::literals;
void f() {
    for (int i{0}; i < 5; ++i) {
        cout << "B";
        this_thread::sleep_for(10ms);
    }
}
```

detach - 2

```
int main() {  
    {  
        thread t{f};  
        // daemon: term by programmers at MIT  
        // supernatural being working in the background  
        t.detach(); // ... Disk And Execution MONitor  
        cout << t.joinable() << endl;  
    } // attn: thread proceeds!  
    for (int i{0}; i < 5; ++i) {  
        cout << "A";  
        this_thread::sleep_for(10ms);  
    }  
    cout << endl;  
}  
  
0  
ABBAABBABA
```

detach – 3

```
#include <iostream> // detach2.cpp
#include <thread>
using namespace std;
using namespace std::literals;
void f() {
    for (int i{0}; i < 5; ++i) {
        cout << "B";
        this_thread::sleep_for(10ms);
    }
}
int main() {
    thread t{f};
    t.detach();
    this_thread::sleep_for(20ms);
}
```

detach – 3

```
#include <iostream> // detach2.cpp
#include <thread>
using namespace std;
using namespace std::literals;
void f() {
    for (int i{0}; i < 5; ++i) {
        cout << "B";
        this_thread::sleep_for(10ms);
    }
}
int main() {
    thread t{f};
    t.detach();
    this_thread::sleep_for(20ms);
}
```

→ Ausgabe: BB!

detach – 3

```
#include <iostream>    // detach2.cpp
#include <thread>
using namespace std;
using namespace std::literals;
void f() {
    for (int i{0}; i < 5; ++i) {
        cout << "B";
        this_thread::sleep_for(10ms);
    }
}
int main() {
    thread t{f};
    t.detach();
    this_thread::sleep_for(20ms);
}
```

→ Ausgabe: BB! ...da daemon-Threads beendet werden!