

Verteilte Systeme

...für C++ Programmierer

Threadsafe Interfaces

by

Dr. Günter Kolousek

Problem gelöst?

- ▶ Jeder kritische Abschnitt wird thread-safe gemacht
 - ▶ d.h. mittels Mutex geschützt

Problem gelöst?

- ▶ Jeder kritische Abschnitt wird thread-safe gemacht
 - ▶ d.h. mittels Mutex geschützt
- ▶ Keine Pointer oder Referenzen aus den kritischen Abschnitten herausführen:
 - ▶ durch Zurückgeben eines Rückgabewertes
 - ▶ durch Speichern in globaler (oder sonstig zugreifbarer) Variable
 - ▶ durch Weitergeben an Funktionen, die nicht unter unserer Kontrolle stehen (!!!)

Problem gelöst?

- ▶ Jeder kritische Abschnitt wird thread-safe gemacht
 - ▶ d.h. mittels Mutex geschützt
- ▶ Keine Pointer oder Referenzen aus den kritischen Abschnitten herausführen:
 - ▶ durch Zurückgeben eines Rückgabewertes
 - ▶ durch Speichern in globaler (oder sonstig zugreifbarer) Variable
 - ▶ durch Weitergeben an Funktionen, die nicht unter unserer Kontrolle stehen (!!!)
- ▶ Damit können also keine race conditions mehr auftreten!

Problem gelöst?

- ▶ Jeder kritische Abschnitt wird thread-safe gemacht
 - ▶ d.h. mittels Mutex geschützt
- ▶ Keine Pointer oder Referenzen aus den kritischen Abschnitten herausführen:
 - ▶ durch Zurückgeben eines Rückgabewertes
 - ▶ durch Speichern in globaler (oder sonstig zugreifbarer) Variable
 - ▶ durch Weitergeben an Funktionen, die nicht unter unserer Kontrolle stehen (!!!)
- ▶ Damit können also keine race conditions mehr auftreten!
 - ▶ **NEIN!**

Thread-safe Interface?

```
// similar to std::stack
template<typename T>
class Stack {
public:
    int size() const;
    bool empty() const;
    T& top(); // undefined if empty
    T const& top() const;
    void push(T const&);
    void push(T&&);
    void pop(); // undefined if empty
};
```

Probleme?

Thread-safe Interface? – 2

`s.size() == 1`

t1	t2
<pre>if (!s.empty()) { v = s.top() // ... s.pop() }</pre>	<pre>if (!s.empty()) { v = s.top() s.pop() }</pre>

Thread-safe Interface? – 2

`s.size() == 1`

t1	t2
<pre>if (!s.empty()) { v = s.top() // ... s.pop() }</pre>	<pre>if (!s.empty()) { v = s.top() s.pop() }</pre>

→ `s.pop()` wird aufgerufen, obwohl Stack leer ist → undefiniertes Verhalten!

→ `v` verweist auf nicht existentes Objekt

Thread-safe Interface: Lösungen

- ▶ einfachste Lösung: top wirft Exception, wenn kein Element am Stack ansonsten liefert es Kopie zurück.
- ▶ Problem gelöst

Thread-safe Interface: Lösungen

- ▶ einfachste Lösung: top wirft Exception, wenn kein Element am Stack ansonsten liefert es Kopie zurück.
- ▶ Problem gelöst ...ja, aber...
 - ▶ Exception muss abgefangen werden und das macht die Programmierung gegen diese Schnittstelle mühsamer
 - ▶ Aufruf von `empty()` ist jetzt redundant
 - ▶ mehrmalige Verarbeitung eines Elementes

Thread-safe Interface? – 3

`s.size() == 2`

t1	t2
<pre>if (!s.empty()) { v = s.top() // ... s.pop() }</pre>	<pre>if (!s.empty()) { v = s.top() // ... s.pop() }</pre>

Thread-safe Interface? – 3

`s.size() == 2`

t1	t2
<pre>if (!s.empty()) { v = s.top() // ... s.pop() }</pre>	<pre>if (!s.empty()) { v = s.top() // ... s.pop() }</pre>

→ 1 Wert wird 2 Mal verarbeitet und ein Wert wird nicht gelesen (aber gelöscht)!

Thread-safe Interface? – 3

`s.size() == 2`

t1	t2
<pre>if (!s.empty()) { v = s.top() // ... s.pop() }</pre>	<pre>if (!s.empty()) { v = s.top() // ... s.pop() }</pre>

→ 1 Wert wird 2 Mal verarbeitet und ein Wert wird nicht gelesen (aber gelöscht)!

→ Race condition!

Thread-safe Interface: Lösungen – 2

- ▶ einfachste Lösung:
 - ▶ top wirft Exception, wenn kein Element am Stack ansonsten liefert es Kopie zurück und
 - ▶ pop wirft Exception, wenn Stack leer und pop vor Verarbeitung aufrufen.
- ▶ Problem gelöst

Thread-safe Interface: Lösungen – 2

- ▶ einfachste Lösung:
 - ▶ top wirft Exception, wenn kein Element am Stack ansonsten liefert es Kopie zurück und
 - ▶ pop wirft Exception, wenn Stack leer und pop vor Verarbeitung aufrufen.
- ▶ Problem gelöst ...ja, aber...
 - ▶ Exception muss abgefangen werden und das macht die Programmierung gegen diese Schnittstelle mühsamer
 - ▶ Aufruf von `empty()` ist jetzt redundant

Thread-safe Interface: Lösungen – 3

- ▶ Zusammenlegen von top und pop...
 - ▶ pop liefert das oberste Element zurück
 - ▶ pop wirft Exception, wenn kein Element am Stack
- ▶ Problem gelöst

Thread-safe Interface: Lösungen – 3

- ▶ Zusammenlegen von top und pop...
 - ▶ pop liefert das oberste Element zurück
 - ▶ pop wirft Exception, wenn kein Element am Stack
- ▶ Problem gelöst
 - ▶ ja, aber...

```
stack<vector<int>>> s;  
...  
v = s.pop(); // copy constructor
```

vector ist dynamische Datenstruktur → heap!
 - ▶ Wenn der Kopierkonstruktor eine `bad_alloc` Exception wirft, dann sind die Daten verloren (vom Stack weg und nicht in v angekommen)!
 - ▶ Lösung: aufsplitten in top und pop...

Thread-safe Interface: Lösungen – 3

- ▶ Zusammenlegen von top und pop...
 - ▶ pop liefert das oberste Element zurück
 - ▶ pop wirft Exception, wenn kein Element am Stack
- ▶ Problem gelöst
 - ▶ ja, aber...

```
stack<vector<int>>> s;  
...  
v = s.pop(); // copy constructor
```

vector ist dynamische Datenstruktur → heap!
 - ▶ Wenn der Kopierkonstruktor eine `bad_alloc` Exception wirft, dann sind die Daten verloren (vom Stack weg und nicht in v angekommen)!
 - ▶ Lösung: aufsplitten in top und pop... → Race condition!!

Thread-safe Interface: Lösungen – 4

1. Referenzparameter:

```
vector<int> result;  
s.pop(result);
```

► Nachteile

- Instanz muss vorher angelegt werden
- Konstruktor könnte Parameter benötigen, die vorweg nicht verfügbar sind
- gespeicherter Typ muss zuweisbar sein

Thread-safe Interface: Lösungen – 5

2. Kopierkonstruktor oder Verschiebekonstruktor werfen keine Exception

- ▶ Nachteile

- ▶ Tja, das muss erst einmal so sein

3. Rückgabe eines Pointers auf das zurückgegebene Objekt

- ▶ Nachteile

- ▶ im Kontext von Nebenläufigkeit!
 - ▶ manuelle Speicherverwaltung bei rohen Pointern: daher `shared_ptr` sinnvoller
 - ▶ Overhead bei einfachen Typen wie `int`

4. Kombination von 1 mit 2 oder 3

Thread-safe Interface: Stack

```
struct EmptyStack : public std::exception {};  
  
template<typename T>  
class ThreadsafeStack {  
    public:  
        ThreadsafeStack();  
        ThreadsafeStack(const ThreadsafeStack&);  
        ThreadsafeStack& operator=(  
            const ThreadsafeStack&) = delete;  
        bool empty() const;    // not needed any more  
        void push(T);  
        shared_ptr<T> pop();    // EmptyStack!  
        void pop(T&);    // EmptyStack!  
};
```

Thread-safe Interface: Stack – 2

```
#include <exception> // stack.h
#include <mutex>
#include <stack>
struct EmptyStack : public std::exception {};

template<typename T>
class ThreadsafeStack {
    std::stack<T> data;
    // because of copy-cons and empty()
    mutable std::mutex m;
public:
    ThreadsafeStack() {}
    ThreadsafeStack(const ThreadsafeStack& o) {
        std::lock_guard<std::mutex> lock(o.m);
        // don't do it in member initializer list!
        // don't forget: you need the lock!
        data = o.data; }
}
```

Thread-safe Interface: Stack – 3

```
ThreadsafeStack& operator=(  
    const ThreadsafeStack&) = delete;  
void push(T value) {  
    std::lock_guard<std::mutex> lock(m);  
    data.push(value);  
}  
std::shared_ptr<T> pop() {  
    std::lock_guard<std::mutex> lock(m);  
    if (data.empty()) throw EmptyStack();  
    auto const res{std::make_shared<T>(  
        data.top())};  
    data.pop();  
    return res;  
}
```

- ▶ `shared_ptr`, `weak_ptr`: sind thread-safe, aber **nicht** die Ressource auf die zugegriffen wird!

Thread-safe Interface: Stack – 4

```
void pop(T& value) {  
    std::lock_guard<std::mutex> lock(m);  
    if (data.empty()) throw EmptyStack();  
    value = data.top();  
    data.pop();  
}  
  
// not recommended:  
// if (!s.empty())  
//     s.pop(); // exc. EmptyStack may occur!  
bool empty() const {  
    std::lock_guard<std::mutex> lock(m);  
    return data.empty();  
}  
};
```


Thread-safe Interface: Stack – 5

```
#include <iostream>    // teststack.cpp
#include <thread>
#include "stack.h"
using namespace std;

void reader(ThreadsafeStack<int>& s) {
    int i;
    while (true) {
        this_thread::sleep_for(500ms);
        //s.pop(i);    // per reference
        i = *s.pop(); // using shared pointer
        cout << i << endl;
    }
}
```

Thread-safe Interface: Stack – 6

```
void writer(ThreadsafeStack<int>& s) {  
    int i{};  
    while (true) {  
        s.push(i);  
        ++i;  
        this_thread::sleep_for(500ms);  
    } }
```

```
int main() {  
    ThreadsafeStack<int> s;  
    thread r{reader, ref(s)};  
    thread w{writer, ref(s)};  
    r.join();  
    w.join(); }
```

Granularität beim Locken

- ▶ feingranulares Locking (fine-grained) vs. grobgranulares Locking (coarse-grained)
 - ▶ Wird Lock zu lange gehalten, dann sinkt Performance

Granularität beim Locken

- ▶ feingranulares Locking (fine-grained) vs. grobgranulares Locking (coarse-grained)
 - ▶ Wird Lock zu lange gehalten, dann sinkt Performance
 - ▶ → Lock frühzeitig zurückgeben, z.B. mit `unique_lock`

Granularität beim Locken

- ▶ feingranulares Locking (fine-grained) vs. grobgranulares Locking (coarse-grained)
 - ▶ Wird Lock zu lange gehalten, dann sinkt Performance
 - ▶ → Lock frühzeitig zurückgeben, z.B. mit `unique_lock`
 - ▶ Wird Lock zu kurz gehalten, dann Race Condition

Granularität beim Locken

- ▶ feingranulares Locking (fine-grained) vs. grobgranulares Locking (coarse-grained)
 - ▶ Wird Lock zu lange gehalten, dann sinkt Performance
 - ▶ → Lock frühzeitig zurückgeben, z.B. mit `unique_lock`
 - ▶ Wird Lock zu kurz gehalten, dann Race Condition
- ▶ Ein Lock soll nur die kürzest notwendige Zeit gehalten werden, um die Operation auszuführen.

Lazy initialization

- ▶ Wann muss nicht gelockt werden?
 - ▶ bei ausschließlichen read-only Zugriff
- ▶ Was ist, wenn Daten nur erzeugt, aber dann nicht mehr verändert werden...
- ▶ Zwei Möglichkeiten
 - ▶ globale Daten initialisiert zur Übersetzungszeit oder beim Starten des Programmes (vor Lesezugriff)
 - ▶ kein Lock notwendig
 - ▶ aber Speicher wird auch verbraucht, wenn diese Daten u.U. überhaupt nicht gelesen werden

Lazy initialization

- ▶ Wann muss nicht gelockt werden?
 - ▶ bei ausschließlichen read-only Zugriff
- ▶ Was ist, wenn Daten nur erzeugt, aber dann nicht mehr verändert werden...
- ▶ Zwei Möglichkeiten
 - ▶ globale Daten initialisiert zur Übersetzungszeit oder beim Starten des Programmes (vor Lesezugriff)
 - ▶ kein Lock notwendig
 - ▶ aber Speicher wird auch verbraucht, wenn diese Daten u.U. überhaupt nicht gelesen werden
 - ▶ Daten werden initialisiert, wenn diese benötigt werden (lazy initialization) → Lock nur bei der Initialisierung notwendig (da mehrfaches Initialisieren!)

Lazy initialization – 2

```
shared_ptr<Element> ptr;  
void use_ptr() {  
    if (!ptr) {  
        ptr.reset(new Element);  
    }  
    ptr->do_something();  
}
```

→ nicht threadsafe!

Lazy initialization – 3

```
shared_ptr<Element> ptr;  
mutex m;  
void use_ptr() {  
    lock_guard<mutex> lock{m};  
    if (!ptr) {  
        ptr.reset(new Element);  
    }  
    ptr->do_something();  
}
```

→ threadsafe

Lazy initialization – 3

```
shared_ptr<Element> ptr;  
mutex m;  
void use_ptr() {  
    lock_guard<mutex> lock{m};  
    if (!ptr) {  
        ptr.reset(new Element);  
    }  
    ptr->do_something();  
}
```

→ threadsafe,

- ▶ aber Lock auch beim Lesen notwendig
- ▶ fehleranfällig
- ▶ Flaschenhals des Locks: Serialisierung!

Lazy initialization – 4

Double checked locking:

```
shared_ptr<Element> ptr;  
mutex m;  
void use_ptr() {  
    if (!ptr) { // <--  
        lock_guard<mutex> lock{m};  
        if (!ptr) {  
            ptr.reset(new Element);  
        }  
    }  
    ptr->do_something();  
}
```

Lazy initialization – 4

Double checked locking:

```
shared_ptr<Element> ptr;  
mutex m;  
void use_ptr() {  
    if (!ptr) { // <--  
        lock_guard<mutex> lock{m};  
        if (!ptr) {  
            ptr.reset(new Element);  
        }  
    }  
    ptr->do_something();  
}
```

→ Race condition möglich!

Lazy initialization – 4

Double checked locking:

```
shared_ptr<Element> ptr;  
mutex m;  
void use_ptr() {  
    if (!ptr) { // <--  
        lock_guard<mutex> lock{m};  
        if (!ptr) {  
            ptr.reset(new Element);  
        }  
    }  
    ptr->do_something();  
}
```

→ Race condition möglich!

Abfrage des Pointers ist **nicht** synchronisiert mit Setzen (in reset)!

Lazy initialization – 5

```
#include <iostream>    // once.cpp
#include <thread>
#include <mutex>
```

```
using namespace std;
shared_ptr<int> ptr;
once_flag resource_flag;
void init_resource() {
    ptr.reset(new int{42});
    cout << "ptr reset to " << *ptr << endl;
}

void use_ptr() {
    call_once(resource_flag, init_resource);
    cout << *ptr << endl;
}
```

Lazy initialization – 6

```
int main() {  
    thread t1{use_ptr};  
    thread t2{use_ptr};  
    t1.join(); t2.join();  
}
```

ptr reset to 42

42

42

Rekursives Locken

```
#include <iostream>    // nonrecursive_locking.cpp
#include <thread>
#include <mutex>
```

```
using namespace std;
```

```
class Counter {
    int data;
    mutex m;
public:
    bool zero() {
        lock_guard<mutex> lock{m};
        return data == 0;
    }
}
```

Rekursives Locken – 2

```
void incr() {  
    lock_guard<mutex> lock{m};  
    ++data;  
}  
void decr() {  
    lock_guard<mutex> lock{m};  
    if (!zero()) { --data; }  
}  
};  
int main() {  
    Counter cnt;  
    cnt.decr();  
}
```

Rekursives Locken – 2

```
void incr() {  
    lock_guard<mutex> lock{m};  
    ++data;  
}  
void decr() {  
    lock_guard<mutex> lock{m};  
    if (!zero()) { --data; }  
}  
};  
int main() {  
    Counter cnt;  
    cnt.decr();  
}
```

terminiert auf meinem System nicht!

Rekursives Locken – 3

- ▶ Nicht korrekt!

Rekursives Locken – 3

- ▶ Nicht korrekt!
- ▶ da mehrfaches Locken in einem Thread ein nicht definiertes Verhalten aufweist!
- ▶ `mutex` muss gegen `recursive_mutex` ausgetauscht werden!
 - ▶ Ressourcen-intensiver: Zähler + Thread-ID müssen verwaltet und gespeichert werden

Rekursives Locken – 3

- ▶ Nicht korrekt!
- ▶ da mehrfaches Locken in einem Thread ein nicht definiertes Verhalten aufweist!
- ▶ mutex muss gegen `recursive_mutex` ausgetauscht werden!
 - ▶ Ressourcen-intensiver: Zähler + Thread-ID müssen verwaltet und gespeichert werden
- ▶ oder: Aufteilung in private und öffentliche Methoden!
 - ▶ öffentliche Methoden thread-sicher
 - ▶ rufen private Methoden auf
 - ▶ private Methoden nicht thread-sicher

Rekursives Locken – 4

```
#include <iostream>    // recursive_locking.cpp
#include <thread>
#include <mutex>
using namespace std;
class Counter {
    int data;
    recursive_mutex m;
public:
    bool zero() {
        lock_guard<recursive_mutex> lock{m};
        return data == 0;
    }
    void incr() {
        lock_guard<recursive_mutex> lock{m};
        ++data;
    }
}
```

Rekursives Locken – 5

```
void decr() {  
    lock_guard<recursive_mutex> lock{m};  
    if (!zero()) {  
        --data;  
    }  
}  
};  
  
int main() {  
    Counter cnt;  
    cnt.decr();  
}
```


private/öffentliche Methoden

```
#include <iostream>    // private_public.cpp
#include <thread>
#include <mutex>
using namespace std;
class Counter {
    int data;
    mutex m;
    bool zero_() {
        return data == 0;
    }
    void incr_() {
        ++data;
    }
}
```

private/öffentliche Methoden – 2

```
void decr_() {  
    if (!zero_()) {  
        --data;  
    }  
}  
  
public:  
    bool zero() {  
        lock_guard<mutex> lock{m};  
        return zero_();  
    }  
    void incr() {  
        lock_guard<mutex> lock{m};  
        incr_();  
    }
```

private/öffentliche Methoden – 3

```
void decr() {  
    lock_guard<mutex> lock{m};  
    decr_();  
}  
};  
  
int main() {  
    Counter cnt;  
    cnt.decr();  
}
```