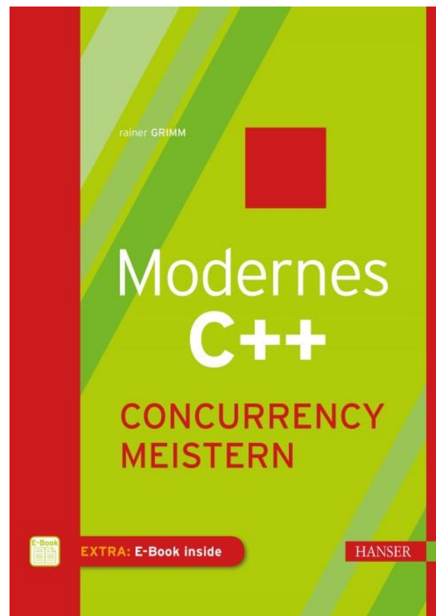
HANSER



Leseprobe

zu

Modernes C++: Concurrency meistern

von Rainer Grimm

ISBN (Buch): 978-3-446-45590-0
ISBN (E-Book): 978-3-446-45665-5

Weitere Informationen und Bestellungen unter
http://www.hanser-fachbuch.de/
sowie im Buchhandel

© Carl Hanser Verlag, München

Inhalt

Einführung XI

Teil I: Der Überblick 1

1 Concurrency mit modernem C++ 3
1.1 C++11 und C++14: Die Grundlagen 3
1.1.1 Das Speichermodell 4
1.1.2 Multithreading 4
1.2 C++17: Die parallelen Algorithmen der Standard Template Library 6
1.2.1 Ausführungsstrategie 7
1.2.2 Neue Algorithmen 7
1.3 Fallstudien 7
1.3.1 Berechnung der Summe eines Vektors 7
1.3.2 Thread-sichere Initialisierung eines Singletons 7
1.3.3 Fortwährende Optimierung mit CppMem 7
1.4 C++20: Die concurrent Zukunft 8
1.4.1 Atomare Smart Pointer 8
1.4.2 Erweiterte Futures 8
1.4.3 Latches und Barriers 9
1.4.4 Coroutinen 9
1.4.5 Transaction Memory 9
1.4.6 Task-Blöcke 9
1.5 Herausforderungen 10
1.6 Best Practice 10
1.7 Zeitbibliothek 10
1.8 Glossar 10

Teil II: Die Details **11**

2 Das Speichermodell **13**
2.1 Der Vertrag 13
2.1.1 Die Grundlagen 14
2.1.2 Die Herausforderungen 15
2.2 Atomare Datentypen 16
2.2.1 Starkes versus schwaches Speichermodell 16
2.2.2 std::atomic_flag 18
2.2.3 Das Klassen-Template std::atomic 23
2.2.4 Benutzerdefinierte atomare Datentypen 30
2.2.5 Die atomaren Operationen 31
2.2.6 Freie atomare Funktionen 31
2.2.7 std::shared_ptr 31
2.3 Synchronisations- und Ordnungsbedingungen 34
2.3.1 Die sechs Variationen des C++-Speichermodells 34
2.3.2 Sequenzielle Konsistenz 36
2.3.3 Acquire-Release-Semantik 38
2.3.4 std::memory_order_consume 46
2.3.5 Relaxed-Semantik 51
2.4 Fences 53
2.4.1 atomic_thread_fence als Speicherbarriere 53
2.4.2 Die drei Fences 54
2.4.3 Acquire Release Fences 56
2.4.4 Synchronisation mit atomaren Variablen oder Fences 57

3 Multithreading **63**
3.1 Threads 64
3.1.1 Erzeugung 64
3.1.2 Lebenszeit 65
3.1.3 Argumente 68
3.1.4 Methoden 71
3.2 Geteilte Daten 73
3.2.1 Mutexe 75
3.2.2 Locks 81
3.2.3 Thread-sichere Initialisierung 90
3.3 Thread-lokale Daten 96
3.4 Bedingungsvariablen 98
3.4.1 Der Arbeitsablauf 100

3.4.2 Lost Wakeup und Spurious Wakeup 102
3.5 Tasks 102
3.5.1 Tasks versus Threads 103
3.5.2 std::async 104
3.5.3 std::packaged_task 110
3.5.4 std::promise und std::future 112
3.5.5 Tasks als sicherer Ersatz für Bedingungsvariablen 119

4 Parallele Algorithmen der Standard Template Library 123
4.1 Ausführungsstrategie 124
4.1.1 Parallel und vektorisierte Ausführungsstrategie 124
4.2 Algorithmen 126
4.3 Die neuen Algorithmen 126
4.3.1 Das funktionale Erbe 130

5 Fallstudien 133
5.1 Berechnen der Summe eines Vektors 134
5.1.1 Single-threaded Summation 134
5.1.2 Multi-threaded Summation mit einer geteilten Variable 140
5.1.3 Thread-lokale Summation 146
5.1.4 Schlussfolgerung 155
5.2 Thread-sichere Initialisierung eines Singletons 156
5.2.1 Double-Checked Locking Pattern 156
5.2.2 Performanzmessung 158
5.2.3 Das Thread-sichere Meyers Singleton 160
5.2.4 std::lock_guard 162
5.2.5 std::call_once mit dem std::once_flag 163
5.2.6 Atomare Variablen 164
5.2.7 Performanzzahlen der verschiedenen Thread-sicheren Implementierungen 168
5.3 Fortwährende Optimierung mit CppMem 168
5.3.1 CppMem – Ein Überblick 170
5.3.2 CppMem: Nicht-atomare Variablen 173
5.3.3 CppMem: Locks 178
5.3.4 CppMem: Atomare Variablen mit sequenzieller Konsistenz 179
5.3.5 CppMem: Atomare Variablen mit Acquire-Release-Semantik 184
5.3.6 CppMem: Atomare Variablen mit nicht-atomaren Variablen 188
5.3.7 CppMem: Atomare Variablen mit Relaxed-Semantik 189
5.4 Schlussfolgerung 191

6 C++20 ... 193
6.1 Atomare Smart Pointer ... 194
6.1.1 Eine Thread-sichere, einfach verkettete Liste ... 194
6.2 Erweiterte Futures ... 196
6.2.1 std::future ... 196
6.2.2 std::async, std::packaged_task und std::promise ... 197
6.2.3 Neue Futures erzeugen ... 198
6.3 Latches und Barriers ... 200
6.3.1 std::latch ... 201
6.3.2 std::barrier ... 202
6.3.3 std::flex_barrier ... 203
6.4 Coroutinen ... 204
6.4.1 Die Generatorfunktion ... 205
6.4.2 Die Details ... 207
6.5 Transactional Memory ... 209
6.5.1 ACI(D) ... 209
6.5.2 Synchronized- und atomic-Blöcke ... 210
6.5.3 Transaction safe versus Transation unsafe Code ... 213
6.6 Task-Blöcke ... 214
6.6.1 Fork und join ... 214
6.6.2 define_task_block vs. define_task_block_restore_thread ... 215
6.6.3 Das Interface ... 216
6.6.4 Der Scheduler ... 217

Teil III: Anhang ... 219

A Herausforderungen ... 221
A.1 ABA ... 221
A.1.1 Eine Analogie ... 221
A.1.2 Ein unkritisches ABA-Szenario ... 222
A.1.3 Eine lock-freie Datenstruktur ... 222
A.1.4 Das ABA-Problem ... 223
A.1.5 Lösung des ABA-Problems ... 223
A.2 Blockieren ... 225
A.3 Verletzung von Programminvarianten ... 226
A.4 Data Race ... 228
A.5 False Sharing ... 229
A.6 Lebenszeitprobleme von Variablen ... 230

A.7 Race Conditions 231
A.8 Threads verschieben 231
A.9 Deadlock 233

B Best Practice 235
B.1 Allgemein 235
B.1.1 Code Reviews 235
B.1.2 Minimiere Sie das Teilen von veränderlichen Variablen 236
B.1.3 Minimieren Sie das Warten 236
B.1.4 Verwenden Sie dynamische Codeanalyse-Werkzeuge 236
B.1.5 Verwenden Sie statische Codeanalyse-Werkzeuge 237
B.1.6 Wenden Sie die passende Abstraktion an 238
B.1.7 Ziehen Sie unveränderliche Daten vor 238
B.2 Multithreading 238
B.2.1 Threads 238
B.2.2 Teilen von Variablen 240
B.2.3 Bedingungsvariablen 243
B.2.4 Promises und Futures 246
B.3 Speichermodell 247
B.3.1 Programmieren Sie nicht Lock-frei 247
B.3.2 Setzen Sie bewährte Muster zum Lock-freien Programmieren ein 247
B.3.3 Verwenden Sie die Zusicherungen des Speichermodells 247
B.3.4 Verwenden Sie volatile nicht zur Synchronisation 248

C Die Zeitbibliothek 249
C.1 Das Zusammenspiel des Zeitpunkts, der Zeitdauer und des Zeitgebers 249
C.2 Zeitpunkt 250
C.2.1 Vom Zeitpunkt zur Kalenderzeit 250
C.2.2 Bruch des gültigen Zeitbereichs 252
C.3 Zeitdauer 253
C.3.1 Berechnungen 255
C.4 Zeitgeber 258
C.4.1 Genauigkeit und Stetigkeit 258
C.4.2 Die Epoche 261
C.5 Schlafen und Warten 263
C.5.1 Konventionen 263
C.5.2 Verschiedene Wartestrategien 264

Glossar 269

Stichwortverzeichnis 271

Einführung

Concurrency mit modernem C++ ist eine Reise durch die aktuellen und zukünftigen Features rund um Concurrency in C++.

- **C++11** und **C++14** besitzen die elementaren Bausteine, um gleichzeitige und parallele Programme zu schreiben.
- Mit **C++17** erhielten wir die parallelen Algorithmen der Standard Template Library (STL). Das heißt, dass die meisten der Algorithmen der STL sequenziell, parallel oder parallel und vektorisierend ausgeführt werden können.
- Die Geschichte zur Gleichzeitigkeit in C++ geht weiter. Mit **C++20** können wir auf erweiterte Features, Coroutinen, Transaktionen und mehr hoffen.

Dieses Buch geht auf die Theorie zur Gleichzeitigkeit in modernem C++ ein und bietet darüber hinaus viele lauffähige Codebeispiele. Damit lässt sich die Theorie gewinnbringend mit der Praxis verknüpfen.

Da sich das Werk intensiv mit der Gleichzeitigkeit beschäftigt, werde ich viele Fallen präsentieren und zeigen, wie sich diese überwinden lassen.

Konventionen

Hier sind die wenigen Konventionen, die ich in meinem Buch einhalte.

Fonts

- *Italic* hebt Ausdrücke leicht hervor.
- **Fett** hebt Ausdrücke stark hervor.
- `Monospace` steht für kleine Codeschnipsel. Dies können Anweisungen oder Schlüsselwörter, aber auch Namen von Typen, Variablen und Klassen sein.

Symbole

- ⇒ steht für Schlussfolgerungen im mathematischen Sinne. Zum Beispiel bedeutet a ⇒ b: Wenn a eintritt, dann auch b.

Kästchen

Kästchen enthalten spezielle Informationen, Tipps und Warnungen.

Die Hintergrundinformation

Beschreibung des Tipps

Beschreibung der Warnung

Deutsche und englische Begriffe

Beim Übersetzen meines Buchs ins Deutsche stand ich häufig vor der Herausforderung: Soll ich einen etablierten englischen Begriff ins Deutsche übersetzen? Für die deutsche Sprache spricht typischerweise die Lesbarkeit des Buchs, für die englische Sprache spricht vor allem die Verständlichkeit, denn häufig verwende ich lang etablierte Begriffe. Im Zweifelsfall habe ich mich für die englischen Fachbegriffe entschieden, zumal es Begriffe im Englischen gibt, zu denen kein deutsches Pendant existiert. So werden zum Beispiel die zwei unterschiedlichen englischen Begriffe Data Race und Race Condition vereinfachend ins Deutsche mit kritischem Wettlauf übersetzt. Englische Begriffe, die ich direkt ins Deutsche übernehme, werde ich in diesem Buch groß schreiben.

Im Kapitel Glossary nehme ich auf die wenigen, leicht etablierten deutschen Begriffe wie Verklemmung oder Nebenläufigkeit Bezug und verweise auf ihre entsprechenden, deutlich stärker etablierten englischen Begriffe. Ohne diese Zuordnung ist das Verwirrungspotenzial beim Wechsel zwischen deutscher und englischer Fachliteratur deutlich zu hoch.

Beispiele

Alle Dossierbeispiele sind lauffähig. Das bedeutet, dass sie mit einem hinreichend aktuellen Compiler übersetzt und ausgeführt werden können. Der Name der Sourcode-Datei befindet sich im Header des Sourcecode-Ausdrucks. Falls es aus Gründen der Übersichtlichkeit notwendig ist, werde ich die Direktive `using namespace std` in den Beispielen verwenden.

Übersetzen und Ausführen

Das Übersetzen und Ausführen der Programme ist relativ einfach, wenn es Beispiele zum C++11- und C++14-Standard betrifft. Jeder moderne Compiler setzt diese beiden Standards bereits vollständig um. Für den GCC (siehe https://gcc.gnu.org/)- und den Clang (siehe https://clang.llvm.org/)-Compiler muss der verwendetet C++-Standard beim Übersetzen angegeben und gegen die threading-Bibliothek gelinkt werden.

Zum Beispiel erzeugt der g++-Compiler als eine ausführbare Datei `thread` mit der folgenden Kommandozeile: `g++ -std=c++14 -pthread thread.cpp -o thread`.

- **`-std=c++14`**: verwende den C++14-Standard
- **`-pthread`**: füge die Multithreading-Unterstützung mit der pthread-Bibliothek hinzu
- **`thread.cpp`**: Name der Sourcecode-Datei
- **`-o thread`**: Name der ausführbaren Datei

Der clang++-Compiler kann mit den gleichen Argument wie der g++-Compiler aufgerufen werden. Der Microsoft Visual Studio 17 C++ Compiler unterstützt ebenfalls den C++14-Standard.

Falls Sie keinen modernen C++-Compiler zur Verfügung haben, können Sie einen der sehr vielen Online-Compiler verwenden. Arne Mertz gibt in seinem Blog-Artikel C++ Online Compiler unter https://arne-mertz.de/2017/05/online-compilers/ eine sehr gute Übersicht zu den verfügbaren C++-Online-Compilern.

Mit dem C++17- und insbesondere dem C++20-Standard wird die Geschichte deutlich kompliziert. Ich habe für die Beispiele das Framework HPX (High Performance ParallelX) (siehe http://stellar.cct.lsu.edu/projects/hpx/) installiert. HPX ist ein universal einsetzbares Laufzeitsystem für parallele und verteilte Applikationen jeder Größe. HPX hat bereits die parallelen Algorithmen der Standard Template Library und viele der neuen Features aus dem C++20-Standard implementiert.

Wie das Buch gelesen werden sollte

Falls Sie mit der Gleichzeitigkeit in C++ nicht vertraut sind, sollten Sie das Buch mit dem Überblickskapitel beginnen.

Sobald Sie sich einen Überblick verschafft haben, können Sie sich genauer mit den Details beschäftigen. Überspringen Sie beim ersten Durchlesen des Buchs dabei das Kapitel Das Speichermodell und starten Sie direkt mit Kapitel Multithreading. Anschließend sollten Sie die parallelen Algorithmen der Standard Template Library studieren. Das Kapitel Fallstudien soll Ihnen insbesondere helfen, die Theorie mit der Praxis zu verknüpfen.

Das Kapitel C++20 ist optional, denn es gibt einen Ausblick auf die nahe C++-Zukunft.

Die drei Kapitel des Anhangs Herausforderungen, Best Practice und Die Zeitbibliothek runden einerseits das Buch ab, bieten aber auch andererseits wertvolle Zusatzinformation.

Persönliche Anmerkungen

Danksagungen

Diese Buch habe ich ursprünglich in Englisch geschrieben und ca. ein halbes Jahr nach seinem Erscheinen ins Deutsche übersetzt. Für die ursprünglich englische Version startete ich einen Aufruf für *Proofreader* in der C++-Community. Die Resonanz dieses Aufrufs auf

meinem englischen Blog http://www.ModernesCpp.com war überwältigend. Mehr als 50 Experten wollten mein Buch Korrektur lesen.

Hier sind die Namen der Korrekturleser in alphabetischer Reihenfolge: Nikos Athanasiou, Robert Badea, Joe Das, Jonas Devlieghere, Juliette Grimm, Marius Grimm, Randy Hormann, Lasse Natvig, Erik Newton, Ian Reeve, Bart Vandewoestyne, Dafydd Walters, Andrzej Warzynski und Enrico Zschemisch.

Über mich

Meine Wurzeln als Softwarearchitekt, Gruppenleiter und Trainer reichen bis ins Jahr 1999 zurück. In meiner Freizeit schreibe ich gerne Artikel und Bücher zu C++ und Python. Seit 2016 verfolge ich nur noch meine Leidenschaft und bin selbständiger Trainer für C++ und Python.

Meine besonderen Umstände

Die englischen Originalversion dieses Buches habe ich in Oberstdorf begonnen, während ich eine neue Hüfte bekam. Während meines Aufenthaltes in der Klinik und dem anschließenden Rehaaufenthalt in Bad Sebastiansweiler habe ich ca. die Hälfte der englischen Originalversion dieses Buches verfasst. Um ehrlich zu sein, hat mir das Schreiben diese Buchs sehr geholfen, über diese schwierige Phase hinwegzukommen.

Der Programmierer wendet in dem Beispiel atomare Variablen an. Er hält damit seinen Teil des Vertrages ein. Das System sichert damit wohldefiniertes Verhalten. Das bedeutet im Wesentlichen, dass kein Data Race besteht. Zusätzlich kann das System die vier Operationen in jeder Reihenfolge ausführen. Falls nun die Relaxed-Semantik zum Einsatz kommt, ändern sich die Grundlagen des Vertrags fundamental. Einerseits wird es von den Programmierer deutlich anspruchsvoller, das Verhalten des Programms zu verstehen; andererseits besitzt das System deutlich mehr Möglichkeiten der Optimierung.

Mit der Relaxed-Semantik, auch schwaches Speichermodell genannt, sind viel mehr Ausführungsreihenfolgen der vier Anweisungen möglich. Selbst das nicht intuitive Verhalten kann auftreten, dass der Thread 1 die Operationen des Threads 2 in einer anderen Reihenfolge wahrnimmt, denn es gibt keinen globalen Zeitgeber. Aus der Perspektive des Threads 1 ist es möglich, dass die Anweisung `res2 = x.load()` die Anweisung `y.store(1)` überholt. Es ist sogar möglich, dass sowohl Thread 1 als auch Thread 2 ihre Anweisungen nicht in der Reihenfolge ausführen, in der sie im Sourcecode stehen. So kann zum Beispiel Thread 2 zuerst `res2 = x.load()` und dann `y.store(1)` ausführen.

Zwischen der sequenziellen Konsistenz und der Relaxed-Semantik sind noch ein paar Abstufungen des Speichermodells. Die Wichtigste ist die Acquire-Release-Semantik. Mit der Acquire-Release-Semantik verwendet der Programmierer schwächere Regeln, sodass das System mehr Optimierungspotenzial besitzt. Die Acquire-Release-Semantik ist Schlüssel für ein tieferes Verständnis der Synchronisation und der partiellen Ordnung in Multithreading-Programmen, denn diese synchronisieren sich mithilfe der Acquire-Release-Semantik an expliziten Synchronisationspunkten. Ohne diese Synchronisationspunkte gäbe es kein wohldefiniertes Verhalten mit Threads, Tasks oder Bedingungsvariablen.

Die sequenzielle Konsistenz ist das Standardverhalten für atomare Operationen. Doch was heißt das? Für jede atomare Operation kann das Speichermodell explizit angegeben werden. Falls kein Speichermodell explizit gesetzt ist, wird implizit das Flag `std::memory_order_seq_cst` verwendet.

Daher ist der Codeschnipsel

```
x.store(1);
res = x.load();
```

äquivalent zu dem folgenden Codeschnipsel:

```
x.store(1, std::memory_order_seq_cst);
res = x.load(std::memory_order_seq_cst);
```

Der Einfachheit halber kommt in diesem Buch die erste Variante zum Einsatz. Nun ist es an der Zeit, einen tieferen Blick in die atomaren Datentypen zu werfen. Los geht es mit dem elementaren Datentyp `std::atomic_flag`.

2.2.2 std::atomic_flag

`std::atomic_flag` besitzt ein sehr einfaches Interface. Seine `clear`-Methode erlaubt es, seinen Wert auf `false` zu setzen. Mit der Methode `test_and_set` lässt sich der Wert auf `true` setzen. Das atomare Flag bietet keine Methode an, um nach seinem Wert zu fragen. Um `std::atomic_flag` zu verwenden, muss es zuerst mit der Konstante

`ATOMIC_FLAG_INIT` auf `false` gesetzt werden. `std::atomic_flag` besitzt zwei herausragende Eigenschaften.

`std::atomic_flag` ist

- der einzige atomare Datentyp, der lock-free ist. Ein nicht-blockierender Algorithmus ist lock-free, falls zugesichert ist, dass ein systemweiter Fortschritt besteht.
- der elementare Baustein für die höheren Thread-Abstraktionen.

Der einzige lock-free atomare Datentyp? Alle verbleibenden atomaren Datentypen können ihre Funktionalität gemäß dem C++-Standard anbieten, indem sie intern Mutexe verwenden. Die verbleibenden atomaren Datentypen besitzen eine Methode `is_lock_free`. Damit lässt sich einfach prüfen, ob der atomare Datentyp mit einem Mutex implementiert ist. Dies ist auf den populären Architekturen äußerst unwahrscheinlich. Trotzdem muss diese Eigenschaft geprüft werden, um sicherzustellen, dass der Code lock-free ausgeführt wird.

Obwohl das Interface des `std::atomic_flag` sehr eingeschränkt, lässt sich damit ein Spinlock implementieren. Mit einem Spinlock lässt sich ein kritischer Bereich entsprechend einem Mutex schützen. Im Gegensatz zum Mutex wird ein Spinlock nicht passiv warten, bis er den Lock des Mutex erhält. Er wird permanent nach dem Lock fragen, um den kritischen Bereich verwenden zu können. Der Spinlock benötigt keinen teueren Kontextwechsel vom Userspace zum Kernelspace. Diesen Vorteil erkauft er sich, indem er die CPU zu 100 % auslastet.

Das Beispiel spinLock.cpp zeigt einen Spinlock, der mit einem `std::atomic_flag` implementiert ist:

```
1  // spinLock.cpp
2
3  #include <atomic>
4  #include <thread>
5
6  class Spinlock{
7      std::atomic_flag flag;
8      public:
9      Spinlock(): flag(ATOMIC_FLAG_INIT){}
10
11     void lock(){
12         while( flag.test_and_set() );
13     }
14
15     void unlock(){
16         flag.clear();
17     }
18 };
19
20 Spinlock spin;
21
22 void workOnResource(){
23     spin.lock();
24     // shared resource
25     spin.unlock();
26 }
```

```
27
28
29 int main(){
30
31     std::thread t(workOnResource);
32     std::thread t2(workOnResource);
33
34     t.join();
35     t2.join();
36
37 }
```

Beide Threads `t` und `t2` konkurrieren um den kritischen Bereich. Der Einfachheit halber besteht der kritische Bereich in Zeile 24 nur aus einem Kommentar. Wie verhält sich der Spinlock? Die Klasse Spinlock besitzt in Anlehnung an einen Mutex die Methoden `lock` und `unlock`. Zusätzlich initialisiert der Konstruktor von Spinlock die `std::atomic_flag` zu `false` (Zeile 9).

Falls Thread `t` die Funktion `workOnResource` ausführt, kann es zu folgendem Ablauf kommen.

1. Thread `t` erhält den Lock, da sein `lock`-Aufruf erfolgreich war. Der `lock`-Aufruf ist dann erfolgreich, falls der initiale Wert des Flags in der Zeile 12 `false` ist. In diesem Fall setzte der Thread `t` das Flag in einer atomaren Operation auf `true`. Der Wert `true` ist der Wert, den die while-Schleife an den Thread `t2` zurückgibt, falls dieser versucht, das Lock zu erhalten. Daher befindet sich nun Thread `t2` in einer Endlosschleife. Thread `t2` besitzt keine Möglichkeit, den Wert des Flags auf `false` zu setzen. Daher muss er warten, bis Thread `t` seine `unlock`-Methode aufruft und das Flag auf `false` zurücksetzt (Zeilen 15–17).
2. Thread `t` erhält nicht den Lock. Dieser Fall entspricht genau dem Fall 1 mit vertauschten Rollen.

Die Methode `test_and_set` des `std::atomic_flag` hat einen genaueren Blick verdient. Die Methode `test_and_set` besteht aus zwei Operationen: einem Lesen und einem Schreiben. Die entscheidende Beobachtung ist, dass beide Operationen in einem atomaren Schritt ausgeführt werden. Falls diese Zusicherung nicht gelten würde, bestünde die Möglichkeit, dass ein gleichzeitiges Lesen und Schreiben auf der Ressource in Zeile 24 stattfinden würde. Dies ist per Definition ein Data Race und das Programm besitzt damit ein undefiniertes Verhalten.

Es ist sehr interessant, das aktive Warten eines Spinlocks mit dem passiven Warten eines Mutex zu vergleichen.

Spinlock versus Mutex

Was wird mit der CPU-Auslastung meines PCs passieren, falls die Funktion `workOnResource` den Spinlock für 2 Sekunden lockt (Zeilen 23–25 in der Datei spinLockSleep.cpp)?

```
1 // spinLockSleep.cpp
2
3 #include <atomic>
```

```
4  #include <thread>
5
6  class Spinlock{
7      std::atomic_flag flag;
8      public:
9      Spinlock(): flag(ATOMIC_FLAG_INIT){}
10
11     void lock(){
12         while( flag.test_and_set() );
13     }
14
15     void unlock(){
16         flag.clear();
17     }
18 };
19
20 Spinlock spin;
21
22 void workOnResource(){
23     spin.lock();
24     std::this_thread::sleep_for(std::chrono::milliseconds(2000));
25     spin.unlock();
26 }
27
28
29 int main(){
30
31     std::thread t(workOnResource);
32     std::thread t2(workOnResource);
33
34     t.join();
35     t2.join();
36
37 }
```

Falls die Theorie stimmt, muss eine der vier CPUs voll ausgelastet sein. Genau dies findet statt. Das Bild 2.5 bringt es auf den Punkt.

Mit jeder neuen Ausführung des Programms geht die Last einer CPU auf 100 %. Jedes Mal ist eine andere CPU am Zuge.

Im nächsten Beispiel mutex.cpp kommt ein Mutex anstelle des Spinlocks zum Einsatz. Die Frage ist natürlich: Ändert sich die CPU-Auslastung des PCs?

```
1  // mutex.cpp
2
3  #include <mutex>
4  #include <thread>
5
6  std::mutex mut;
7
8  void workOnResource(){
9      mut.lock();
```

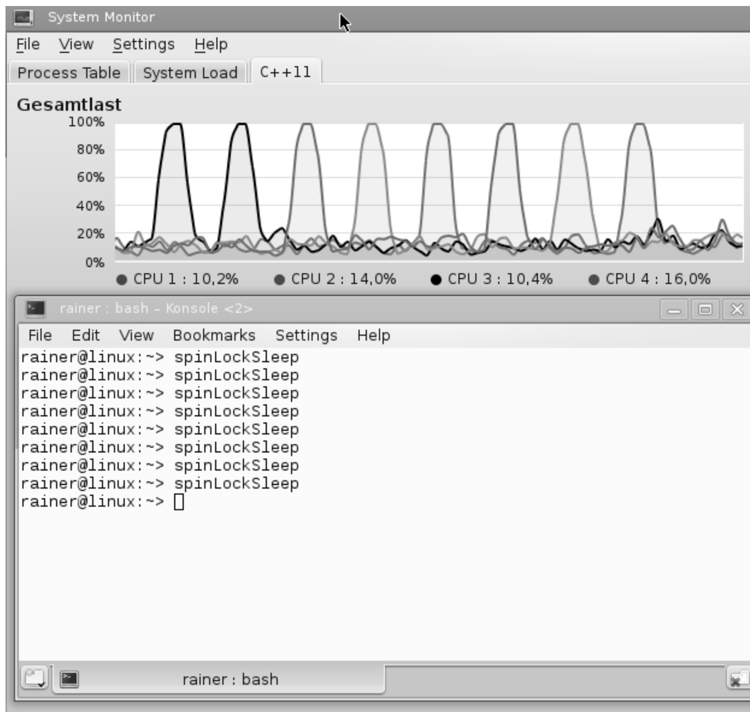


Bild 2.5 Ein Spinlock, der für zwei Sekunden blockiert

```
10      std::this_thread::sleep_for(std::chrono::milliseconds(5000));
11      mut.unlock();
12  }
13
14  int main(){
15
16      std::thread t(workOnResource);
17      std::thread t2(workOnResource);
18
19      t.join();
20      t2.join();
21
22  }
```

Obwohl das Programm mehrmals aufgerufen wird, lässt sich keine signifikante Last auf den CPUs in feststellen.

Jetzt geht es einen Schritt weiter: von den elementaren Bausteine `std::atomic_flag` zu den deutlichen mächtigeren atomaren Datentypen: dem Klassen-Template `std::atomic`. Die verschiedenen teilweisen und vollen Spezialisierungen für Wahrheitswerte, Ganzzahlen und Zeiger bieten ein deutlich mächtigeres Interface als `std::atomic_flag`. Der Nachteil der Varianten von `std::atomic` ist, dass sie intern einen Lock verwenden können.

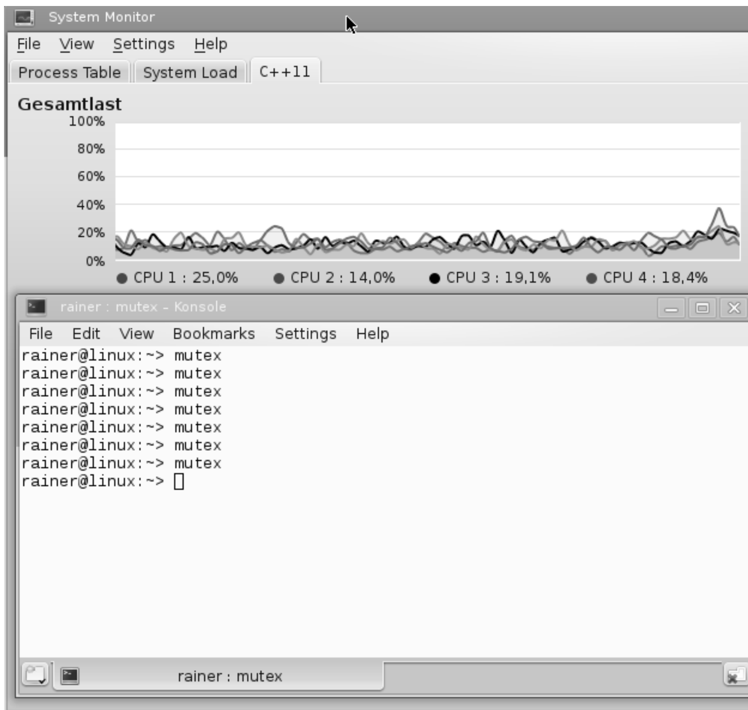


Bild 2.6 Ein Mutex, der für zwei Sekunden blockiert

2.2.3 Das Klassen-Template std::atomic

Die einfachste vollständige Spezialisierung ist `std::atomic<std::bool>`.

Das Klassen-Template `std::atomic<bool>`

`std::atomic<std::bool>` kann im Gegensatz zu `std::atomic_flag` explizit auf `true` oder `false` gesetzt werden.

atomic ist nicht volatile

Was hat das Schlüsselwort `volatile` in C# oder Java gemein mit dem Schlüsselwort `volatile` in C++? Nichts! Das ist der Unterschied zwischen `volatile` und `std::atomic` in C++.

- **`volatile`**: unterbindet optimierte Lese- und Schreiboperationen.
- **`std::atomic`**: definiert eine atomare Variable, die Thread-sicheres Lesen und Schreiben erlaubt.

Eigentlich ist es eine so einfache Regel. Doch genau hier beginnt die Verwirrung. Das Schlüsselwort `volatile` besitzt in C# und Java die Bedeutung von `std::atomic` in C++. Um es nochmals auf den Punkt zu bringen: `volatile` besitzt keine Multithreading-Semantik in C++.

Mithilfe des Speichermodells lässt sich der Arbeitsablauf deutlich formaler beschreiben:

1. `work= "done"` is **sequenced-before** `ready = true`
   ⇒ work= `"done"` **happens-before** `ready = true`
2. `while(!ready.load()){}` is **sequenced-before** `std::cout « work « std::endl`
   ⇒ `while(!ready.load()){}` **happens-before** `std::cout « work « std::endl`
3. `ready= true` **synchronizes-with** `while(!ready.load())`
   ⇒ `ready= true` **inter-thread happens-before** `while (!ready.load()){}`
   ⇒ `ready= true` **happens-before** `while (!ready.load()){}`

Die Schlussfolgerung ist: Da die **happens-before**-Relation transitiv ist, gilt: `work = "done"` **happens-before** `ready= true` **happens-before** `while(!ready.load()){}` **happens-before** `std::cout « work « std::endl`

In der sequenziellen Konsistenz nimmt ein Thread alle Operationen eines anderen Threads und damit von allen anderen Threads in derselben Reihenfolge wahr. Diese zentrale Eigenschaft der sequenziellen Konsistenz gilt nicht mehr, wenn die Acquire-Release-Semantik verwendet wird. Die Acquire-Release-Semantik ist das Modell, dem Java oder auch C# nicht folgen. Gleichzeitig ist es aus dem Bereich, in dem unsere Intuition zu schwinden beginnt.

2.3.3 Acquire-Release-Semantik

Es gibt keine globale Synchronisation zwischen Threads bei der Acquire-Release-Semantik. Es gibt nur eine Synchronisation zwischen atomaren Operationen **auf der gleichen atomaren Variable**. Eine Schreiboperation in einem Thread synchronisiert sich mit einer Leseoperation in einem anderen Thread, wenn sie auf der gleichen atomaren Variable stattfindet.

Die Acquire-Release-Semantik basiert auf einer zentralen Idee: Eine Release-Operation synchronisiert mit einer Acquire-Operation auf derselben atomaren Variable und etabliert eine Ordnungsbedingung. Das bedeutet, dass alle Lese- und Schreiboperationen nicht vor eine Acquire-Operationen und alle Lese- und Schreiboperationen nicht nach einer Release-Operation verschoben werden können.

Was ist eine Acquire-Operation? Das Lesen einer atomaren Variablen mit `load` oder `test_and_set` ist eine Acquire-Operation. Es gibt noch deutlich mehr Acquire-Operationen: das Anfordern eines Locks, die Erzeugung eines Threads oder das Warten mit einer Bedingungsvariablen. Natürlich gilt auch das Gegenteil. Das Freigeben eines Locks, der `join`-Aufruf auf einem Thread oder die Benachrichtigung einer Bedingungsvariable sind Release-Operationen. Entsprechend ist eine `store`- oder `clear`-Operation auf einer atomaren Variable eine Release-Operation. Acquire- und Release-Operationen treten typischerweise in Paaren auf.

Es ist sehr aufschlussreich, die letzten Zeilen aus einer anderen Perspektive zu betrachten. Das Locken eines Mutex ist eine Acquire-Operation; das Unlocken eines Mutex eine Release-Operation. Bildlich gesprochen bedeutet dies, dass eine Operation `var += 1` nicht aus einem kritischen Bereich heraus verschoben werden darf. Das bedeutet aber auch, dass eine Variable in einen kritischen Bereich hineingeschoben werden darf. Damit wird die Variable von den nicht-geschützt in einen geschützten Bereich verschoben.

Es ist sehr hilfreich, das folgende Bild 2.11 im Kopf zu behalten.

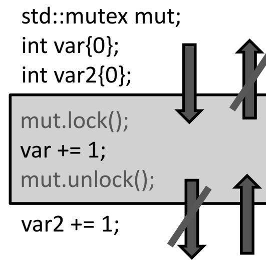


Bild 2.11 Der kritische Bereich

Das Speichermodell ermöglicht ein tieferes Verständnis von Multithreading

Ein wichtiges Aspekt des Speichermodells ist, dass es das Verständnis von Multithreading-Programmen deutlich verbessert. Insbesondere die Acquire-Release-Semantik hilft, die höheren Synchronisationsmechanismen wie Mutexe besser zu verstehen. Die gleiche Argumentation lässt sich aber auch auf das Starten eines Threads und den `join`-Aufruf darauf anwenden. Beides sind Acquire-Release-Operationen. Die Geschichte geht mit dem `wait`- und dem `notify_one`-Aufruf einer Bedingungsvariable weiter. `wait` ist die Acquire- und `notify_one` ist die Release-Operation. Was lässt sich zu `notify_all` sagen? `notify_all` ist natürlich auch eine Release-Operation.

Jetzt ist es natürlich sehr interessant, den Spinlock aus dem Abschnitt std::atomic_flag mithilfe der Acquire-Release-Semantik zu implementieren (spinlockAcquireRelease.cpp):

```
1  // spinlockAcquireRelease.cpp
2
3  #include <atomic>
4  #include <thread>
5
6  class Spinlock{
7      std::atomic_flag flag;
8      public:
9      Spinlock(): flag(ATOMIC_FLAG_INIT) {}
10
11     void lock(){
12         while(flag.test_and_set(std::memory_order_acquire));
13     }
14
15     void unlock(){
16         flag.clear(std::memory_order_release);
17     }
18 };
19
20 Spinlock spin;
21
22 void workOnResource(){
23     spin.lock();
```

```
24      // shared resource
25      spin.unlock();
26  }
27
28
29  int main(){
30
31      std::thread t(workOnResource);
32      std::thread t2(workOnResource);
33
34      t.join();
35      t2.join();
36
37  }
```

Der `flag.clear`-Aufruf in Zeile 16 ist eine Release-Operation, der `flag.test_and_set`-Aufruf in Zeile 12 eine Acquire-Operation. Die Acquire-Operation synchronisiert mit der Release-Operation. In dem Beispiel ist die schwergewichtige Synchronisation der zwei Threads mit der sequenziellen Konsistenz `std::memory_order_seq_cst` durch eine leichtgewichtige und daher performantere Acquire-Release-Semantik ersetzt worden (`std::memory_order_acquire` und `std::memory_order_release`). Das Verhalten des Programms hat sich dadurch nicht verändert.

Obwohl der Aufruf `flag.test_and_set(std::memory_order_acquire)` eine Read-Modify-Write-Operation ist, ist in diesem Fall die Acquire-Semantik ausreichend. Vereinfachend gesagt ist `flag` eine atomare Variable.

Die Acquire-Release-Semantik ist transitiv. Das bedeutet: Liegt eine Acquire-Release-Semantik zwischen zwei Threads (a, b) und eine Acquire-Release-Semantik zwischen zwei Threads (b, c) vor, dann gilt, dass auch eine Acquire-Release-Semantik für die zwei Threads (a, c) gilt.

Transitivität

Eine Release-Operation synchronisiert sich mit einer Acquire-Operation auf derselben atomaren Variable und etabliert zusätzlich eine Ordnungsbedingung. Dies sind die Komponenten, um mehrere Threads miteinander zu synchronisieren, die auf denselben atomaren Variablen agieren. Wie ist dies möglich, wenn die Threads keine gemeinsame atomare Variable verwenden? Auf keinen Fall soll sequenzielle Konsistenz als schwergewichtiges Speichermodell zum Einsatz kommen, wenn die leichtgewichtige Acquire-Release-Semantik verwendet werden kann.

Die Antwort auf die Frage ist naheliegend. Dank der Transitivität der Acquire-Release-Semantik lassen sich Threads synchronisieren, die unabhängig sind.

Im folgenden Beispiel transitivity.cpp ist der Thread `t2` mit seinem Arbeitspaket `deliveryBoy` das verbindende Glied zwischen den zwei unabhängigen Threads `t1` und `t3`.

```
1  // transitivity.cpp
2
3  #include <atomic>
4  #include <iostream>
```

```
 5  #include <thread>
 6  #include <vector>
 7
 8  std::vector<int> mySharedWork;
 9  std::atomic<bool> dataProduced(false);
10  std::atomic<bool> dataConsumed(false);
11
12  void dataProducer(){
13      mySharedWork = {1,0,3};
14      dataProduced.store(true, std::memory_order_release);
15  }
16
17  void deliveryBoy(){
18      while(!dataProduced.load(std::memory_order_acquire));
19      dataConsumed.store(true, std::memory_order_release);
20  }
21
22  void dataConsumer(){
23      while(!dataConsumed.load(std::memory_order_acquire));
24      mySharedWork[1] = 2;
25  }
26
27  int main(){
28
29      std::cout << std::endl;
30
31      std::thread t1(dataConsumer);
32      std::thread t2(deliveryBoy);
33      std::thread t3(dataProducer);
34
35      t1.join();
36      t2.join();
37      t3.join();
38
39      for (auto v: mySharedWork){
40          std::cout << v << " ";
41      }
42
43      std::cout << "\n\n";
44
45  }
```

Die Ausgabe des Programms (siehe Bild 2.12) ist vollkommen deterministisch. `mySharedWork` besitzt den Wert 1, 2 und 3.

Es gibt zwei wichtige Beobachtung zu dem Programm transitivity.cpp:

1. Thread `t2` wartet in Zeile 18, bis Thread `t3` `dataProduced` auf `true` (Zeile 14) gesetzt hat.
2. Thread `t1` wartet in Zeile 23, bis Thread `t2` `dataConsumed` auf `true` (Zeile 119) gesetzt hat.

Die verbleibende Argumentation lässt sich schöner an einer Grafik visualisieren.

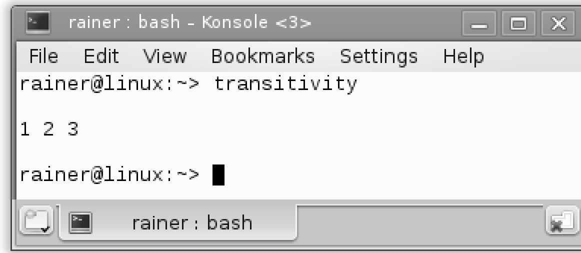


Bild 2.12 Ausgabe des Programms transitivity.cpp

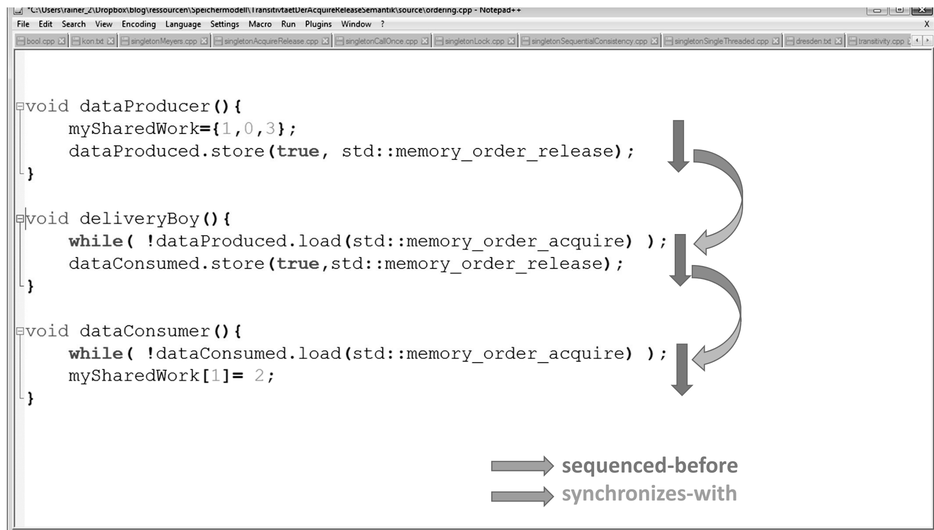


Bild 2.13 Transitivität der Acquire-Release-Semantik

Die entscheidenden Komponenten der Grafik sind die Pfeile.

- Die **blauen** Pfeile stehen für die *sequenced-before*-Relation. Das bedeutet, dass alle Operationen in einem Thread in der Reihenfolge der Sourcecode-Anweisungen stattfinden.
- Die **roten** Pfeile stehen für die *synchronizes-with*-Relation. Der Grund ist die Acquire-Release-Semantik der atomaren Operationen auf den gleichen atomaren Variablen. Damit findet die Synchronisation zwischen den atomaren Variablen und damit auch zwischen den Threads an diesen expliziten Synchronisationspunkten statt.

Sowohl die *sequenced-before-* als auch die *synchronizes-with*-Relation etablieren *happens-before*-Relationen.

Der Rest ist ziemlich einfach. Die `happens-before`-Ordnung von Anweisungen entspricht der Richtung der Pfeile von oben nach unten. Letztendlich besitzen wir die Garantie, dass `mySharedWork[1] == 2` als Letztes ausgeführt wird.

Eine Release-Operation `synchronizes-with` einer Acquire-Operation auf derselben atomaren Variable. Daher können wir einfach Threads synchronisieren, **wenn** … . Genau um dieses **wenn** dreht sich das typische Missverständnis.

Das typische Missverständnis

Warum schreibe ich in diesem Buch über das typische Missverständnis der Acquire-Release-Semantik? Ganz einfach. Viele Teilnehmer meiner Schulungen sind bereits in diese Falle getappt. Zuerst stelle ich den Gut-Fall in dem Programm acquireReleaseWithWaiting.cpp vor.

Warten inklusive

```
1  // acquireReleaseWithWaiting.cpp
2
3  #include <atomic>
4  #include <iostream>
5  #include <thread>
6  #include <vector>
7
8  std::vector<int> mySharedWork;
9  std::atomic<bool> dataProduced(false);
10
11 void dataProducer(){
12     mySharedWork = {1, 0, 3};
13     dataProduced.store(true, std::memory_order_release);
14 }
15
16 void dataConsumer(){
17     while( !dataProduced.load(std::memory_order_acquire) );
18     mySharedWork[1] = 2;
19 }
20
21 int main(){
22
23     std::cout << std::endl;
24
25     std::thread t1(dataConsumer);
26     std::thread t2(dataProducer);
27
28     t1.join();
29     t2.join();
30
31     for (auto v: mySharedWork){
32         std::cout << v << " ";
33     }
34
35     std::cout << "\n\n";
36
37 }
```

Der Consumer-Thread `t1` in Zeile 17 wartet, bis der Consumer-Thread `t2` in Zeile 13 `dataProduced` auf `true` gesetzt hat. `dataProduced` ist der Wächter, der garantiert, dass der Zugriff auf die nicht-atomare Variable `mySharedWork` synchronisiert ist. Das bedeutet, dass der Producer-Thread `t2` zuerst `mySharedWork` initialisiert und dann der Consumer-Thread `t2` den Arbeitsablauf beendet, indem er `mySharedWork[1]` auf 2 setzt. Das Programmverhalten is wohldefiniert.

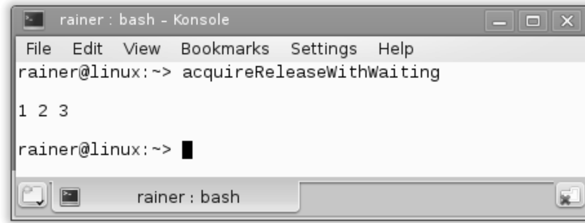


Bild 2.14 Das Programm acquireReleaseWithWaiting in Aktion

Das Bild 2.15 zeigt die *happens-before*-Relation im Thread und die *synchronizes-with*-Relation zwischen den Threads. *synchronizes-with* etabliert eine *happens-before*-Relation. Die weitere Argumentation basiert auf der Transitivität der *happens-before*-Relation.

Damit gilt die Zusicherung: `mySharedWork = {1, 0, 3}` *happens-before* `mySharedWork[1] = 2`.

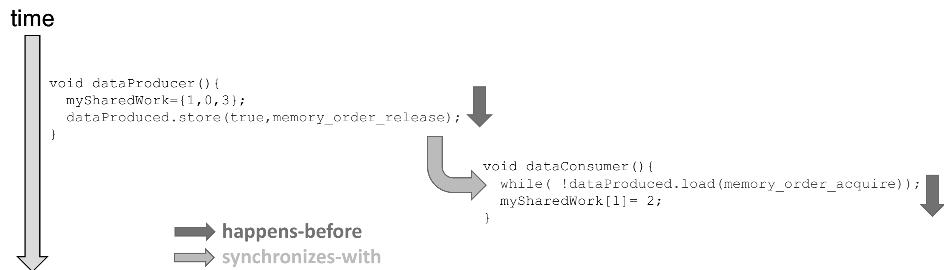


Bild 2.15 Acquire-Release-Semantik mit Warten

Welcher Aspekt wird in der Begründung der Acquire-Release-Semantik häufig übersehen? Das **wenn**.

Wenn ... Was passiert, wenn der Consumer-Thread `t1` in der Zeile 12 nicht auf den Producer-Thread `t2` wartet?

```
 1  // acquireReleaseWithoutWaiting.cpp
 2
 3  #include <atomic>
 4  #include <iostream>
 5  #include <thread>
 6  #include <vector>
 7
 8  std::vector<int> mySharedWork;
 9  std::atomic<bool> dataProduced(false);
10
11  void dataProducer(){
12      mySharedWork = {1, 0, 3};
13      dataProduced.store(true, std::memory_order_release);
14  }
15
```

```
16 void dataConsumer(){
17     dataProduced.load(std::memory_order_acquire);
18     mySharedWork[1] = 2;
19 }
20
21 int main(){
22
23     std::cout << std::endl;
24
25     std::thread t1(dataConsumer);
26     std::thread t2(dataProducer);
27
28     t1.join();
29     t2.join();
30
31     for (auto v: mySharedWork){
32         std::cout << v << " ";
33     }
34
35     std::cout << "\n\n";
36
37 }
```

Das Programm acquireReleaseWithoutWaiting.cpp besitzt undefiniertes Verhalten, denn es gibt ein Data Race um die Variable `mySharedWork`. Das Ausführen des Programms ergibt das folgende (Bild 2.16) nicht-deterministische Verhalten.

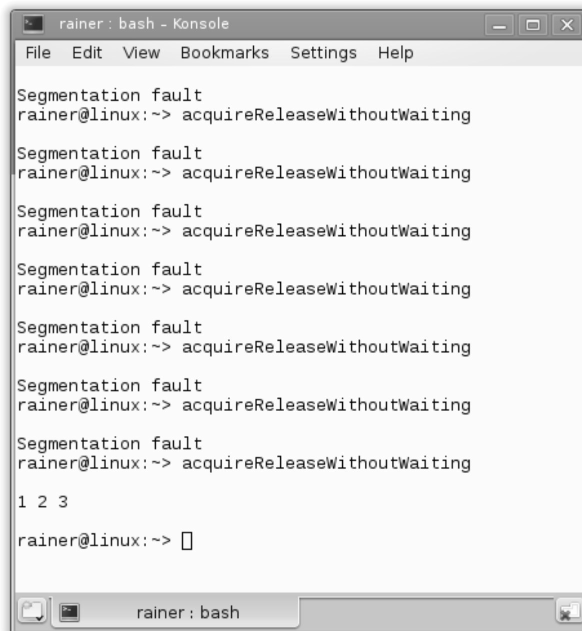


Bild 2.16 Undefiniertes Verhalten mit der Acquire-Release-Semantik

Worin besteht das undefinierte Verhalten? Es gilt doch, dass `dataProduced.store( true, std::memory_order_release)` eine *synchronizes-with*-Relation mit `dataProduced.load (std::memory_order_acquire)` besitzt. Das bedeutet aber nicht, dass die Acquire-Operation auf die Release-Operation wartet. Genau das zeigt das Bild 2.17. In der Grafik wird die `dataProduced.load(std::memory_order_acquire)`-Anweisung vor der `dataProduced.store(true, std::memory_order_release)` ausgeführt. Es gibt daher keine *synchronizes-with*-Relation.

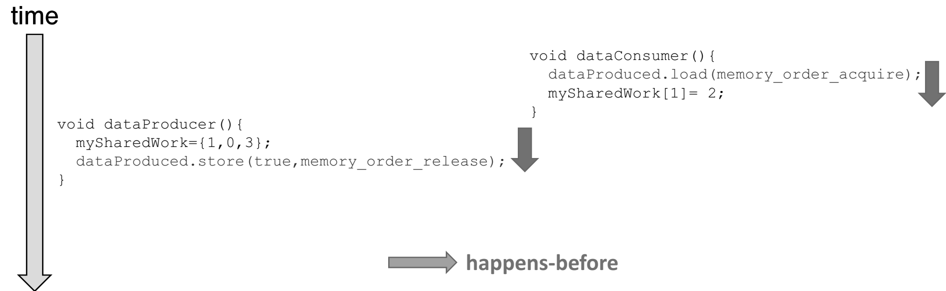


Bild 2.17 Acquire-Release-Semantik ohne Warten

Die Auflösung *synchronizes-with* bedeutet: Wenn `dataProduced.store(true, std::memory_order_release)` vor `dataProduced.load(std::memory_order_acquire)` stattfindet, dann sind alle beobachtbaren Effekte der Operationen vor `dataProduced.store(true, std::memory_order_release)` wahrnehmbar nach `dataProduced.load( std::memory_order_acquire)`. Der entscheidende Punkt ist das Wort **wenn**. Diese Garantie sichert das Prädikat (`while(!dataProduced.load(std::memory_order_acquire)`) in dem Programm acquireReleaseWithWaiting.cpp zu.

Dies lässt sich natürlich deutlich formaler ausdrücken.

Alle Operationen vor `dataProduced.store(true, std::memory_order_release)` *happens-before* allen Operationen nach `dataProduced.load(std::memory_order_acquire)`, wenn die folgende Bedingung gilt: `dataProduced.store(true, std::memory_order_release)` *happens-before* `dataProduced.load(std::memory_order_acquire)`.

Wenn Sie den Abschnitt Die Herausforderungen aufmerksam gelesen haben, werden Sie wohl nur erwarten, dass der Abschnitt zur Relaxed-Semantik als Nächster folgt. Davor möchte ich aber noch genauer auf das Speichermodell `std::memory_order_consume` eingehen, da dies dem Speichermodell `std::memory_order_acquire` sehr ähnlich ist.

2.3.4 std::memory_order_consume

`std::memory_order_consume` ist das legendärste der sechs Speichermodelle, und zwar in zweifacher Hinsicht: Einerseits ist es sehr schwer zu verstehen und daher schwer richtig einzusetzen. Andererseits – und das mag sich in Zukunft ändern – wird es zum jetzigen Zeitpunkt noch von keinem Compiler unterstützt. Mit C++17 hat sich die Lage noch verschärft. Hier ist das offizielle Statement: „The specification of release-consume ordering is being revised, and the use of memory_order_consume is temporarily discouraged.“ (http://en.cppreference.com/w/cpp/atomic/memory_order).

```
25  int main(){
26
27      CriticalData c1;
28      CriticalData c2;
29
30      std::thread t1([&]{deadLock(c1,c2);});
31      std::thread t2([&]{deadLock(c2,c1);});
32
33      t1.join();
34      t2.join();
35
36  }
```

Die Threads `t1` und `t2` rufen die Funktion `deadlock` auf. Die Funktion `deadlock` benötigt die Variablen `CriticalData c1` und `c2` (Zeilen 27 und 28). Da beide Objekte `c1` und `c2` vor gleichzeitigem Zugriff geschützt werden müssen, besitzen sie beide Mutex. Um das Beispiel einfach zu halten, besitzt `CriticalData` abgesehen von dem Mutex keine Methoden oder Attribute.

Ein kurzes Nickerchen von einer Millisekunde in Zeile 16 reicht bereits aus, um diesen Deadlock zu provozieren (Bild 3.11).

Bild 3.11 Zwei Threads verklemmen sich.

Die einzige Möglichkeit, diesen Deadlock zu lösen, besteht darin, das Programm zu beenden.

Locks lösen nicht alle Probleme von Mutexen, doch sie helfen in vielen Fällen.

3.2.2 Locks

Locks verwalten ihre Ressourcen automatisch, indem sie das RAII-Idiom umsetzen. Ein Lock bindet automatisch seinen Mutex im Konstruktor und gibt ihn im Destruktor wieder frei. Das reduziert die Gefahr von Deadlocks deutlich, denn die C++-Laufzeit übernimmt die Verwaltung des Mutex.

Locks gibt es in drei verschiedenen Variationen: `std::lock_guard` für den einfachen und `std::unique_lock` für den anspruchsvollen Anwendungsfall. `std::shared_lock` steht seit C++14 zur Verfügung und erlaubt es, Reader-Writer-Locks zu implementieren.

Zuerst zum einfachen Anwendungsfall.

std::lock_guard

```
std::mutex m;
m.lock();
sharedVariable = getVar();
m.unlock();
```

Der Mutex `m` stellt sicher, dass der Zugriff auf den kritischen Bereich `sharedVariable = getVar()` sequentiell stattfindet. Sequentiell bedeutet in diesem Fall, dass ein Thread nach dem anderen Zugriff auf den kritischen Bereich erhält. Damit stellt das System eine totale Ordnung auf dem kritischen Bereich sicher. Der Code ist einfach, aber empfänglich für Deadlocks. Ein Deadlock tritt dann auf, wenn der kritische Bereich eine Ausnahme auslöst oder der Programmierer schlicht vergessen hat, den Mutex wieder freizugeben. Dies lässt sich mit `std::lock_guard` deutlich eleganter lösen:

```
{
    std::mutex m,
    std::lock_guard<std::mutex> lockGuard(m);
    sharedVariable = getVar();
}
```

Das war einfach. Doch was steht hinter den öffnenden und schließenden geschweiften Klammern? Durch diesen künstlichen Bereich wird die Lebenszeit des `std::lock_guard` geregelt. Das bedeutet, die Lebenszeit des Locks endet genau dann, wenn der Kontrollfluss der geschweiften Klammern endet. Genau zu diesem Zeitpunkt wird der Destruktor des `std::lock_guard` aufgerufen, um den Mutex freizugeben. Das passiert alles automatisch – auch dann, wenn die Funktion `getVar)` in `sharedVariable = getVar()` eine Ausnahme wirft. Auch der Bereich einer Funktion, einer Klasse oder einer Schleife schränken die Lebenszeit eines Objekts ein.

```
{
std::mutex m,
std::lock_guard<std::mutex> lockGuard(m);
sharedVariable = getVar();
}
```

std::scoped_lock mit C++17

Mit C++17 erhielten wir einen neuen Lock: `std::scoped_lock`. Dieser ist dem `std::lock_guard` sehr ähnlich. Im Gegensatz zu `std::lock_guard` kann ein `std::scoped_lock` eine beliebige Anzahl von Mutexen atomar locken.

Zwei Punkt gilt es, zum `std::scoped_lock` im Kopf zu behalten.

1. Falls einer der Threads bereits den Mutex besitzt und der Mutex nicht rekursiv ist, ist das erneute Locken des Mutex ein undefiniertes Verhalten.
2. Sie können nur einfach den Mutex besitzen, ohne ihn zu locken. In diesem Fall muss der `std::scoped_lock` mit dem Flag `std::adopt_lock_t` aufgerufen werden: `std::scoped_lock(std::adopt_lock_t, MutexTypes& ... m.`
3. Sie können mit dem `std::scoped_lock` das Deadlock aus dem Beispiel deadlock.cpp lösen. Im folgenden Abschnitt werde ich auf die Lösung eingehen.

■

Ein `std::unique_lock` ist mächtiger, aber auch teurer als sein kleiner Bruder `std::lock_guard`.

std::unique_lock

Der `std::unique_lock` bietet ein deutlich erweitertes Interface zu dem `std::lock_guard` an.

Ein `std::unique_lock` kann

- ohne einen assoziierten Mutex erzeugt werden,
- mit einem Mutex erzeugt werden, der nicht gelockt ist,
- explizit den assoziierten Mutex locken und freigeben,
- verschoben werden,
- versuchsweise versuchen, den Mutex zu locken,
- verzögert den assoziierten Mutex locken.

Die folgende Tabelle 3.3 stellt das Interface des `std::unique_lock lk` vor.

Tabelle 3.3 Methoden eines std::unique_lock

Methode	Beschreibung
`lk.lock`	Lockt den assoziierten Mutex.
`std::lock(lk1, lk2, ... )`	Lockt automatisch die beliebige Anzahl von assoziierten Mutexen.

`lk.try_lock`
`lk.try_lock_for(relTime)`
`lk.try_lock_until(absTime)`
	Versucht, die assoziierten Mutexe zu locken.

`lk.release()`	Gibt den Mutex frei. Der Mutex bleibt gegebenenfalls gelockt.
`lk.swap(lk2)` `std::swap(lk, lk2)`	Tauscht die Locks.
`lk.mutex()`	Gibt einen Zeiger auf den assoziierten Mutex zurück
`lk.own_lock`	Prüft, ob der Lock `lk` einen Mutex besitzt.

`lk.try_lock_for(relTime)` benötigt eine relative Zeitdauer, `lk.try_lock_until(absTime)` hingegen eine absoluten Zeitpunkt.

`lk.try_lock` versucht, den Mutex zu erhalten, und blockiert nicht. Im Erfolgsfall gibt die Methode `true` zurück, sonst `false`. Im Gegensatz dazu blockieren die Methoden `lk.try_lock_for` und `lk.try_lock_until`, bis die spezifizierte Zeit vergangen ist oder sie den Mutex locken konnten. Sie sollten einen stetigen Zeitgeber für Ihre Zeitpunkte oder Zeitdauern verwenden. Ein stetiger Zeitgeber kann nicht neu gestellt werden.

Die Methode `lk.release` gibt den Mutex zurück. Daher muss dies gegebenenfalls manuell freigegeben werden.

Dank `std::unique_lock` ist es relativ einfach, eine beliebige Anzahl von Mutexen in einem atomaren Schritt zu locken. Damit lassen sich Deadlocks elegant auflösen, die durch das Locken von Mutexen in verschiedener Reihenfolge verursacht wurden.

Zur Erinnerung ist hier nochmals das Programm deadlock.cpp, das im Abschnitt *Mutexe in verschiedener Reihenfolge locken* zu einem Deadlock führte.

```
1  // deadlock.cpp
2
3  #include <iostream>
4  #include <chrono>
5  #include <mutex>
6  #include <thread>
7
8  struct CriticalData{
9      std::mutex mut;
10 };
11
12 void deadLock(CriticalData& a, CriticalData& b){
13
14     a.mut.lock();
15     std::cout << "get the first mutex" << std::endl;
16     std::this_thread::sleep_for(std::chrono::milliseconds(1));
17     b.mut.lock();
18     std::cout << "get the second mutex" << std::endl;
19     // do something with a and b
20     a.mut.unlock();
21     b.mut.unlock();
22
23 }
24
25 int main(){
26
27     CriticalData c1;
28     CriticalData c2;
29
30     std::thread t1([&]{deadLock(c1,c2);});
31     std::thread t2([&]{deadLock(c2,c1);});
32
33     t1.join();
34     t2.join();
35
36 }
```

Mit dem `std::unique_lock` lässt sich das Deadlock lösen. Es ist lediglich notwendig, dass die Funktion `deadlock` ihren Mutex in einem atomaren Schritt lockt. Genau das passiert im Beispiel deadlockResolved.cpp.

```
1  // deadlockResolved.cpp
2
3  #include <iostream>
4  #include <chrono>
5  #include <mutex>
6  #include <thread>
7
8  using namespace std;
```

```
 9
10 struct CriticalData{
11     mutex mut;
12 };
13
14 void deadLock(CriticalData& a, CriticalData& b){
15
16     unique_lock<mutex> guard1(a.mut,defer_lock);
17     cout << "Thread: " << this_thread::get_id() << " first mutex"
18          << endl;
19
20     this_thread::sleep_for(chrono::milliseconds(1));
21
22     unique_lock<mutex> guard2(b.mut,defer_lock);
23     cout << " Thread: " << this_thread::get_id() << " second mutex"
24          <<  endl;
25
26     cout << "   Thread: " << this_thread::get_id() << " get both mutex"
27          << endl;
28     lock(guard1,guard2);
29     // do something with a and b
30 }
31
32 int main(){
33
34     cout << endl;
35
36     CriticalData c1;
37     CriticalData c2;
38
39     thread t1([&]{deadLock(c1,c2);});
40     thread t2([&]{deadLock(c2,c1);});
41
42     t1.join();
43     t2.join();
44
45     cout << endl;
46
47 }
```

Wenn der Konstruktor von `std::unique_lock` mit `std::defer_lock` aufgerufen wird, wird der zugrunde liegende Mutex nicht automatisch gelockt. Zu diesem Zeitpunkt (Zeilen 16 und 22) ist der `std::unique_lock` nur der Besitzer der Mutexe. Dank dem Variadic Template `std::lock` wird das Locken der Mutexe in einem atomaren Schritt ausgeführt (Zeile 28). Ein Variadic Template ist ein Template, das eine beliebige Anzahl an Argumenten annehmen kann. `std::lock` versucht, alle zugrunde liegenden Mutexe in einem atomaren Schritt zu locken. Dabei lockt `std::lock` entweder alle oder keinen Mutex. Im Fehlerfall setzt sich dieser Prozess fort, bis `std::lock` erfolgreich war.

Im Beispiel deadlockResolved.cpp verwaltet zuerst `std::unique_lock` die Lebenszeit seiner Ressource, und dann lockt `std::lock` die assoziierten Mutexe. Der Arbeitsablauf lässt sich aber auch genau anders herum implementieren. Zuerst wird der Mutex gelockt, und

dann verwaltet `std::unique_lock` die Lebenszeit der Ressource. Der Codeschnipsel zeigt die skizzierte, alternative Variante.

```
std::lock(a.mut, b.mut);
std::lock_guard<std::mutex> guard1(a.mut, std::adopt_lock);
std::lock_guard<std::mutex> guard2(b.mut, std::adopt_lock);
```

Beide Varianten lösen das Deadlock in Bild 3.12 auf.

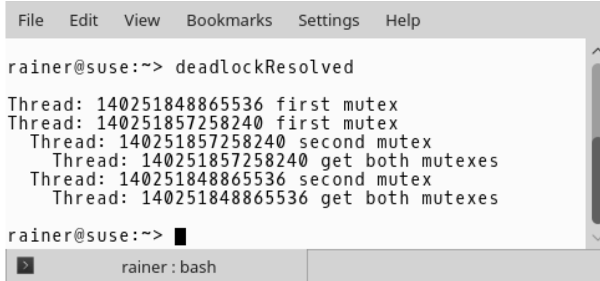


Bild 3.12 Deadlock dank std::lock aufgelöst

Einen Deadlock mit std::scoped_lock auflösen

Mit C++17 wird das Auflösen eines Deadlocks noch einfacher. Ein `std::scoped_lock` kann automatisch eine beliebige Anzahl von Mutexen locken. Dazu muss lediglich ein `std::lock_guard` anstelle eines `std::lock` eingesetzt werden. Hier erscheint die modifizierte Funktion `deadlock` aus deadlockResolved.cpp nochmals.

```
// deadlockResolvedScopedLock.cpp

...
void deadLock(CriticalData& a, CriticalData& b){

  cout << "Thread: " << this_thread::get_id() << " first mutex"
   << endl;
  this_thread::sleep_for(chrono::milliseconds(1));
  cout << "  Thread: " << this_thread::get_id() << " second mutex"
   <<  endl;
  cout << "    Thread: " << this_thread::get_id() << " get both mutex"
   << endl;

  std::scoped_lock(a.mut, b.mut);
  // do something with a and b
}
...
```

C++14 wurde um den `std::shared_lock` erweitert.

std::shared_lock

Ein `std::shared_lock` besitzt dasselbe Interface wie ein `std::unique_lock`, verhält sich aber anders, wenn er mit einem `std::shared_timed_mutex` verwendet. Viele Threads können einen `std::shared_timed_mutex` gemeinsam verwenden und damit ein Reader-Writer-Lock umsetzen. Die Idee des Reader-Writer-Locks ist einfach und sehr praktisch. Eine beliebige Anzahl von Threads kann zu einem Zeitpunkt lesend auf einen kritischen Bereich zugreifen, aber nur ein Thread darf schreibend darauf zugreifen.

Reader-Writer-Locks lösen nicht das fundamentale Problem, dass mehrere Threads um den Zugang zu einem kritischen Bereich in Konkurrenz stehen, aber sie helfen, den Engpass deutlich zu mildern.

Ein Telefonbuch ist ein typischer Anwendungsfall für ein Reader-Writer-Lock. Viele Anwender wollen im Telefonbuch eine Telefonnummer nachschlagen, aber nur wenige wollen eine neue Telefonnummer hinzufügen. Das Beispiel readerWriterLock.cpp setzte die Idee in Sourcecode um.

```
1  // readerWriterLock.cpp
2
3  #include <iostream>
4  #include <map>
5  #include <shared_mutex>
6  #include <string>
7  #include <thread>
8
9  std::map<std::string,int> teleBook{{"Dijkstra", 1972}, {"Scott", 1976},
10     {"Ritchie", 1983}};
11
12 std::shared_timed_mutex teleBookMutex;
13
14 void addToTeleBook(const std::string& na, int tele){
15     std::lock_guard<std::shared_timed_mutex> writerLock(teleBookMutex);
16     std::cout << "\nSTARTING UPDATE " << na;
17     std::this_thread::sleep_for(std::chrono::milliseconds(500));
18     teleBook[na]= tele;
19     std::cout << " ... ENDING UPDATE " << na << std::endl;
20 }
21
22 void printNumber(const std::string& na){
23    std::shared_lock<std::shared_timed_mutex> readerLock(teleBookMutex);
24    std::cout << na << ": " << teleBook[na];
25 }
26
27 int main(){
28
29     std::cout << std::endl;
30
31     std::thread reader1([]{ printNumber("Scott"); });
32     std::thread reader2([]{ printNumber("Ritchie"); });
33     std::thread w1([]{ addToTeleBook("Scott",1968); });
34     std::thread reader3([]{ printNumber("Dijkstra"); });
35     std::thread reader4([]{ printNumber("Scott"); });
```

```
36      std::thread w2([]{ addToTeleBook("Bjarne",1965); });
37      std::thread reader5([]{ printNumber("Scott"); });
38      std::thread reader6([]{ printNumber("Ritchie"); });
39      std::thread reader7([]{ printNumber("Scott"); });
40      std::thread reader8([]{ printNumber("Bjarne"); });
41
42      reader1.join();
43      reader2.join();
44      reader3.join();
45      reader4.join();
46      reader5.join();
47      reader6.join();
48      reader7.join();
49      reader8.join();
50      w1.join();
51      w2.join();
52
53      std::cout << std::endl;
54
55      std::cout << "\nThe new telephone book" << std::endl;
56      for (auto teleIt: teleBook){
57          std::cout << teleIt.first << ": " << teleIt.second
58            << std::endl;
59      }
60
61      std::cout << std::endl;
62
63  }
```

Das Telefonbuch in Zeile 9 ist die geteilte Variable, die es zu schützen gilt. Ein Eintrag des Telefonbuchs besteht aus dem Schlüssel Vorname und dem Wert Telefonnummer. Mithilfe des Schlüssels wird auf den Wert zugegriffen. Acht Akteure wollen auf das Telefonbuch lesend, zwei schreibend zugreifen (Zeilen 31–40). Um das gleichzeitige Lesen zu unterstützen, kommt ein `std::shared_lock<std::shared_timed_mutex` in Zeile 23 zum Einsatz. Dies steht im Kontrast zum schreibenden Zugriff auf die geteilte Variable. In diesem Fall ist ein exklusiver Lock notwendig: `std::lock_guard<std::shared_timed_mutex` in Zeile 15. Zum Abschluss stellt das Programm das veränderte Telefonbuch in den Zeilen 55–59 dar: Bild 3.13.

Bild 3.13 zeigt schön, wie sich die lesenden Threads überlagern, während die schreibenden Threads einer nach dem anderen ausgeführt werden. Das bedeutet natürlich, dass die lesenden Operationen gleichzeitig ausgeführt wurden.

Das war einfach. Zu einfach! Das Programm readerWriterLock.cpp besitzt undefiniertes Verhalten.

Undefiniertes Verhalten Das Programm besitzt undefiniertes Verhalten. Um noch genauer zu sein: Es besitzt ein Data Race. Warum? Bevor Sie weiterlesen, halten Sie für ein paar Sekunden inne und versuchen Sie, das Data Race zu identifizieren. Nebenbei gesagt: Der gleichzeitige Zugriff auf `std::cout` ist nicht das Problem.

Die Charakteristik eines Data Races ist, dass zumindest zwei Threads gleichzeitig auf die geteilte Variable zugreifen und zumindest ein Thread versucht, diese zu verändern. Genau

```
rainer : bash - Konsole <5>
File Edit View Bookmarks Settings Help
rainer@linux:~> readerWriterLock

Scott: 1976Dijkstra: 1972
STARTING UPDATE Scott ... ENDING UPDATE Scott
Ritchie: 1983Scott: 1968
STARTING UPDATE Bjarne ... ENDING UPDATE Bjarne
Ritchie: 1983Scott: 1968Scott: 1968Bjarne: 1965

The new telephone book
Bjarne: 1965
Dijkstra: 1972
Ritchie: 1983
Scott: 1968

rainer@linux:~>
rainer : bash
```

Bild 3.13 Reader-Writer-Lock beim Lesen und Schreiben des Telefonbuchs

dies kann während der Programmausführung stattfinden. Ein Features eines geordneten assoziativen Containers in C++ ist, dass eine Leseoperation diesen verändern kann. Das passiert genau dann, wenn das gesuchte Element nicht im Container ist. Falls `"Bjarne"` nicht im Telefonbuch steht, wird zum Telefonbuch ein Paar (`"Bjarne"`, 0) hinzugefügt. Dieses Data Race lässt sich erzwingen, indem die Ausgabe von `"Bjarne"` in Zeile 40 vor alle anderen Threads geschoben wird (Zeile 31–40).

Bild 3.14 macht das Data Race sichtbar. `"Bjarne"` besitzt die Telefonnummer 0.

Die naheliegende Art, das Data Race zu beseitigen, ist, nur lesend Operationen in der Funktion `printNumber` zuzulassen. Dies zeigt das Programm readerWriterLockResolved.cpp

```
// readerWriterLocksResolved.cpp

...

void printNumber(const std::string& na){
    std::shared_lock<std::shared_timed_mutex>
    readerLock(teleBookMutex);
    auto searchEntry = teleBook.find(na);
    if(searchEntry != teleBook.end()){
        std::cout << searchEntry->first << ": " << searchEntry->second
          << std::endl;
    }
    else {
        std::cout << na << " not found!" << std::endl;
    }
}

...

}
```

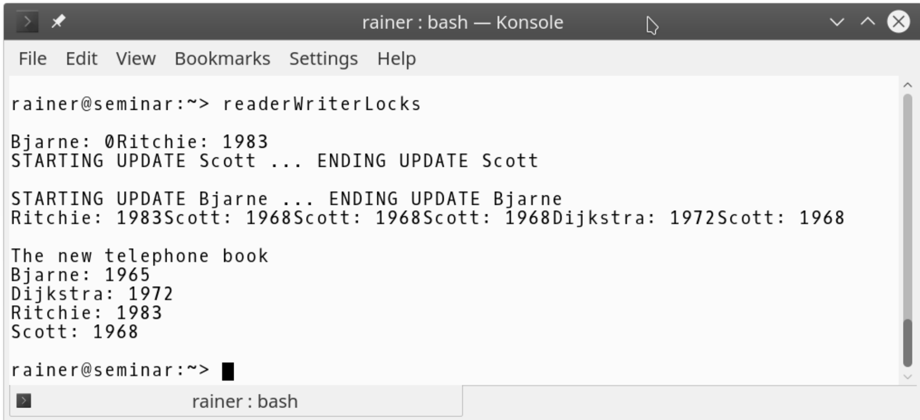


Bild 3.14 Das Programm besitzt ein Data Race.

Falls der Schlüssel nicht im Telefonbuch ist, schreibt das Programm den Schlüssel zusammen mit dem Text `"not found!"` auf die Konsole.

In der zweiten Ausführung des Programms schreibt dieses `"Bjarne not found!"` auf die Konsole. Das gilt nicht für die erste Ausführung. In diesem Fall wird die Funktion `addToTeleBook` zuerst ausgeführt, und damit ist `"Bjarne"` auch im Telefonbuch, wenn nach ihm gesucht wird.

3.2.3 Thread-sichere Initialisierung

Falls eine Variable nur lesend verwendet wird, gibt es keinen Grund, dies aufwendig durch einen Lock oder mit einer atomaren Variable zu synchronisieren. Es muss lediglich sichergestellt werden, dass die Variable Thread-sicher initialisiert wird.

C++ bietet drei Arten an, Variablen Thread-sicher zu initialisieren.

- konstante Ausdrücke
- die Funktion `std::call_once` in Kombination mit dem Flag `std::once_flag`
- statische Variablen mit Blockgültigkeit

Thread-sichere Initialisierung im Haupt-Thread

Der einfachste und vierte Weg, eine Variable Thread-sicher zu initialisieren, wird oft übersehen. Initialisieren Sie die Variable im Haupt-Thread, bevor Sie einen Kind-Thread erzeugt haben.

Konstante Ausdrücke

Konstante Ausdrücke sind solche, die der Compiler zur Compile-Zeit auswerten kann. Damit sind sie implizit Thread-sicher. Durch die Verwendung des Schlüsselworts

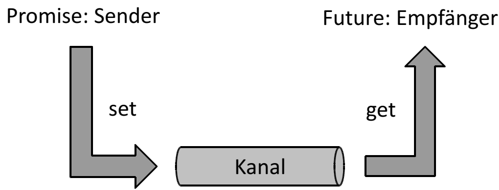


Bild 3.21 Tasks als Datenkanal zwischen Kommunikationsendpunkten

3.5.1 Tasks versus Threads

Tasks unterscheiden sich deutlich von Threads.

```
 1  // asyncVersusThread.cpp
 2
 3  #include <future>
 4  #include <thread>
 5  #include <iostream>
 6
 7  int main(){
 8
 9      std::cout << std::endl;
10
11      int res;
12      std::thread t([&]{ res = 2000 + 11; });
13      t.join();
14      std::cout << "res: " << res << std::endl;
15
16      auto fut= std::async([]{ return 2000 + 11; });
17      std::cout << "fut.get(): " << fut.get() << std::endl;
18
19      std::cout << std::endl;
20
21  }
```

Der Kind-Thread `t` und der asynchrone Funktionsaufruf `std::async` berechnen beide die Summe von `2000` und `11`. Der Erzeuger-Thread erhält das Ergebnis der Berechnung seines Kind-Threads mithilfe der geteilten Variable `res` und stellt das Ergebnis in Zeile 14 dar. Der Aufruf `std::async` in Zeile 16 erzeugt einen Datenkanal zwischen dem Sender (Promise) und dem Empfänger (Future). Der Future fragt das Ergebnis des Datenkanals in Zeile 17 mittels `fut.get()` nach dem Ergebnis der Berechnung. Der `fut.get`-Aufruf ist blockierend.

Bild 3.22 zeigt die Ausgabe des Programms.

Basierend auf dem Programm asyncVersusThread.cpp, stellt die Tabelle 3.5 die Unterschiede von Tasks und Threads gegenüber.

Threads benötigen die Header-Datei `thread`, Tasks die Header-Datei `future`.

Die Kommunikation zwischen dem Erzeuger- und dem Kind-Thread erfordert eine geteilte Variable. Der Task kommuniziert mithilfe des Datenkanals, der automatisch geschützt ist. Daher benötigt der Task keinen Schutzmechanismus wie Mutexe.

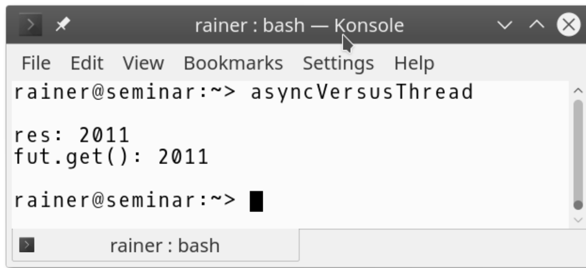


Bild 3.22 Tasks versus Threads

Tabelle 3.5 Threads versus Tasks

Kriterium	Thread	Task
Teilnehmer	Erzeuger- und Kind-Thread	Promise und Future
Kommunikation	geteilte Variable	Datenkanal
Thread-Erzeugung	verbindlich	optional
Synchronisation	der `join`-Aufruf wartet	der `get`-Aufruf blockiert
Ausnahme im Kind-Thread	Erzeuger- und Kind-Thread beenden sich	Rückgabewert des Promise
Arten der Kommunikation	Werte	Werte, Benachrichtigungen und Ausnahmen

Während Sie eine globale Variable *missbrauchen*, um zwischen dem Kind und seinem Erzeuger zu kommunizieren, ist die Kommunikation bei einem Tasks deutlich expliziter. Der Future kann nur einmal das Ergebnis des Promise durch `fut.get()` anfordern. Mehrmaliges Anfordern des Ergebnisses ist undefiniertes Verhalten. Dies gilt nicht für den std::shared_future. Dieser kann das Ergebnis mehr als einmal anfordern.

Der Erzeuger-Thread wartet mithilfe des `join`-Aufruf auf sein Kind-Thread. Der Future `fut` hingegen benutzt den `fut.get()`-Aufruf, der solange blockiert, bis das Ergebnis zur Verfügung steht.

Falls eine Ausnahme im Kind-Thread geworfen wird, werden der Kind- und der Erzeuger-Thread beendet. Im Gegensatz dazu kann der Promise eine Ausnahme an den Future übermitteln, der diese dann managen muss.

Ein Promise kann einen oder mehrere Futures bedienen. Er kann einen Wert, eine Ausnahme oder einfach nur eine Benachrichtigung schicken. Damit ist ein Task ein sicherer Ersatz von Bedingungsvariablen.

3.5.2 std::async

`std::async` ist der einfachste Weg, einen Future zu erzeugen. Er verhält sich wie ein asynchroner Funktionsaufruf. Der Funktionsaufruf nimmt eine aufrufbare Einheit zusammen mit ihren Argumenten an. `std::async` ist ein Variadic Template und kann daher beliebig viele Argumente annehmen. Der Aufruf von `std::async` gibt einen Future `fut` zurück. Das ist ihr Bezug auf das Ergebnis mittels `fut.get()`.

Stichwortverzeichnis

A

ABA 221
Acquire Fence 54
Acquire-Release-Semantik 38
adopt_lock_t 82
Algorithmen 126
arrive_and_drop 202
arrive_and_wait 202
asctime 250
async 104, 105
Atomare Datentypen 16
– atomic 23
– atomic<bool> 23
– atomic<integral type> 29
– atomic<T*> 28
– atomic_flag 18
– compare_exchange_strong 27
– compare_exchange_weak 27
– Schwaches Speichermodell 17
– Starkes Speichermodell 16
– Starkes versus schwaches Speichermodell 16
Atomare Smart Pointer 194
atomic 23
Atomic Block 210
atomic<integral type> 29
atomic<std::bool> 23
atomic<T*> 28
atomic_cancel 213
atomic_commit 213
atomic_compare_exchange_strong 33
atomic_compare_exchange_strong_explicit 33
atomic_compare_exchange_weak 33
atomic_compare_exchange_weak_explicit 33
atomic_exchange_explicit 33
atomic_exhange 33
atomic_flag 18
atomic_is_lock_free 33
Atomicity 209
atomic_load 33
atomic_load_explicit 33
atomic_noexcept 213
atomic_shared_ptr 194
atomic_store 33
atomic_store_explicit 33
atomic_thread_fence 53
atomic_weak_ptr 194
Ausführungsstrategie 124
await_ready 208
await_resume 208
await_suspend 208

B

bad_alloc 213
bad_array_length 213
bad_array_new_length 213
bad_cast 213
bad_exception 213
bad_typeid 213
barrier 202
barriers 53
Bedingungsvariablen 98
– Lost Wakeup 102
– Spurious Wakeup 102
– Workflow 100
Benutzerdefinierte atomare Datentypen 30
Bösartige Race Condition 231

C

Cache Line 229
– False Sharing 229
– hardware_constructive_interference_size 230
– hardware_destructive_interference_size 230
– True Sharing 230
carries-a-dependency-to 50
Child Stealing 217
chrono
– time_point 250

clear 31
compare_exchange_strong 27
compare_exchange_weak 27
Completion Phase 203
Consistency 209
Coroutinen 204
count_down 201
count_down_and_wait 201

D
default 94
deferred 105
define_task_block 215
define_task_block_restore_thread 215
delete 94
dependency-ordered-before 50
detach 66, 71
Double-Checked Locking Pattern 156
Durability 209

E
Eager Evaluation 105
epoch 250
Erweiterte Futures 196
exception 213
exchange 31
exclusive_scan 126

F
Fallstudien 133
False Sharing 229
Fences 53
fetch_add 31
fetch_and 31
fetch_or 31
fetch_sub 31
fetch_xor 31
Fire und Forget Futures 106
flex_barrier 203
foldl 130
foldl1 130
for_each 126
for_each_n 126
Freie atomare Funktionen 31
Full Fence 54
Future
– wait 115
– wait_for 115
– wait_until 115
future 115

G
Garbage Collection 224
get 115
Geteilte Daten 73
– Locks 81
– Mutexe 75
get_future 114
get_id 71
gmtime 250

H
hardware_concurrency 71
hardware_constructive_interference_size 230
hardware_destructive_interference_size 230
Hazard Pointer 224
High Performance ParalleX 215
high_resolution_clock 258
hours 253
HPX 215

I
inclusive_scan 126
is_lock_free 31
Isolation 209
is_ready 201

J
join 66, 71
joinable 71

K
Kalenderzeit 250

L
latch 201
launch
– async 105
– deferred 105
Lazy Evaluation 105
Livelock 226
load 31
LoadLoad 53
LoadStore 53
lock_guard 82
Locks 81

M
make_exceptional_future 198
make_ready_future 198
map 130
Markierte Zustandsreferenz 224
max 250
Memory Barriers 53
Memory Fences 53
memory_order_acq_rel 34
memory_order_acquire 34

memory_order_consume 34, 46
memory_order_relaxed 34
memory_order_release 34
memory_order_seq_cst 34
Meyers Singleton 159
microseconds 253
milliseconds 253
min 250
minutes 253
Modification Order Consistency 51
Multithreading 63
mutex 78
Mutexe 75
– Gefahren 79

N
nanoseconds 253
native_handle 71
notify_all 98
notify_one 98
now 250

P
packaged_task 110
Parallele Algorithmen der STL 123
Parent Stealing 217
promise 114

R
ratio 259
ratio_multiply 259
RCU 224
ready 196
recursive_mutex 78
recursive_timed_mutex 78
reduce 126
Relaxed-Block 210
Relaxed-Semantik 51
Release Fence 54
Release-Acquire-Ordnung 46
Release-Consume-Ordnung 46

S
scanl 130
scanl1 130
scoped_lock 82
scoped_thread 67
seconds 253
Sequenzielle Konsistenz 36
set_exception 114
set_exception_at_thread_exit 114
set_value 114
set_value_at_thread_exit 114
share 115
shared_future 115
shared_lock 87
shared_ptr 31
shared_timed_mutex 78
Singleton 93
sleep_for 71
sleep_until 71
Speicherbarrieren 53
Speichermodell 13
– Grundlagen 14
– Herausforderungen 15
Spinlock 19
steady_clock 258
store 31
StoreLoad 53
StoreStore 53
swap 71
Synchronized-Block 210
system_clock 258

T
Task-Blöcke
– Child Stealing 217
– Interface 216
– Parent Stealing 217
– Scheduler 217
Tasks 102
– async 104
– future 115
– packaged_task 110
– promise 114
– shared_future 115
– Threads versus Tasks 103
Tasks als sicherer Ersatz für Bedingungsvariablen 119
test_and_set 31
then 197
this_thread::get_id 71
this_thread::sleep_for 71
this_thread::sleep_until 71
this_thread::yield 71
Thread 64
– Argumente 68
– detach 71
– Erzeugung 64
– get_id 71
– join 71
– joinable 71
– Lebenszeit 65

– Methoden 71
– swap 71
thread::hardware_concurrency 71
thread_local 96
Thread-lokale Daten 96
Thread-sichere Initialisierung 90
– call_once 92
– Konstante Ausdrücke 90
– once_flag 92
– Statische Variablen 95
timed_mutex 78
time_point 250
time_since_epoch 261
time_t 250
to_time_t 250
Transactional Memory 209
transaction_safe 213
transaction-safe 213
transaction_unsafe 213
transaction-unsafe 213
transform_exclusive_scan 126
transform_inclusive_scan 126
transform_reduce 126
True Sharing 230

U

unique_lock 83

V

valid 115, 196
Vertrag 13

W

wait 98, 201
wait_for 98
wait_until 98
when_all 198
when_any 198
when_any_result 200

Y

yield 71

Z

Zeitpunkt 250