rainer GRIMM

Modernes C++

CONCURRENCY MEISTERN

HANSER

Grimm

Modernes C++: Concurrency meistern

Rainer Grimm

Modernes C++: Concurrency meistern

HANSER

Der Autor:
Rainer Grimm, Rottenburg
www.grimm-jaud.de

Bibliografische Information der Deutschen Nationalbibliothek:
Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über http://dnb.d-nb.de abrufbar.

Lektorat: Brigitte Bauer-Schiewek
Copy editing: Jürgen Dubau, Freiburg/Elbe
Herstellung: Irene Weilhart
Layout: le-tex publishing services GmbH
Umschlagdesign: Marc Müller-Bremer, www.rebranding.de, München
Umschlagrealisation: Stephan Rönigk
Druck und Bindung: Kösel, Krugzell
Ausstattung patentrechtlich geschützt. Kösel FD 351, Patent-Nr. 0748702
Printed in Germany

Print-ISBN: 978-3-446-45590-0
E-Book-ISBN: 978-3-446-45665-5

Inhalt

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

- $\Rightarrow$ steht für Schlussfolgerungen im mathematischen Sinne. Zum Beispiel bedeutet a $\Rightarrow$ 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.

TEIL I

Der Überblick

1 Concurrency mit modernem C++

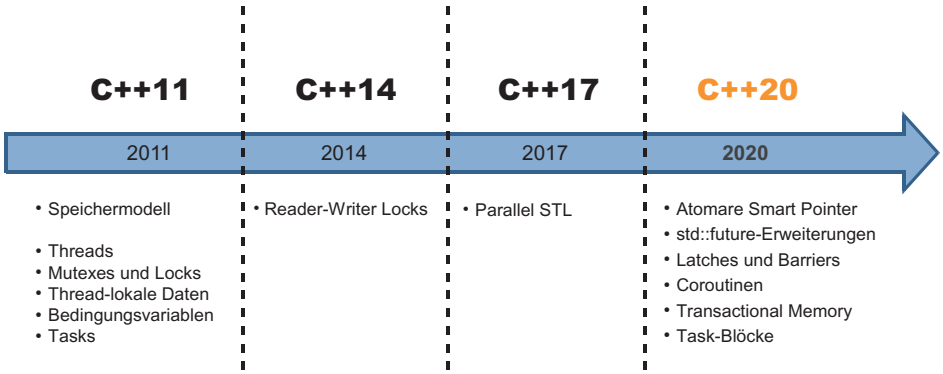

Bild 1.1 Multithreading in C++

Mit der Veröffentlichung des C++11-Standards erhielt C++ eine Multithreading-Bibliothek und ein Speichermodell. Die Bibliothek enthält die elementaren Bausteine wie atomare Variablen, Threads, Locks und Bedingungsvariablen, aber auch mächtigere Bausteine wie Tasks. Dies sind die Fundamente, auf den C++-Standards wie C++17 und C++20 höhere Abstraktionen anbieten können.

Vereinfacht betrachtet, lässt sich die Gleichzeitigkeit in C++ in drei Evolutionsstufen unterteilen.

1.1 C++11 und C++14: Die Grundlagen

Mit C++11 wurde Multithreading in C++ eingeführt. Diese Funktionalität besteht aus zwei Teilen: einem wohldefinierten Speichermodell und einer standardisierten Threading-Schnittstelle. C++14 erweiterte diese Multithreading-Schnittstelle um Reader-Writer Locks.

1.1.1 Das Speichermodell

Die Grundlage für Multithreading ist das wohldefinierte Speichermodell. Dieses Speichermodell muss sich mit den folgenden Punkten auseinandersetzen.

- Atomare Operationen: Operationen, die ohne Unterbrechung ausgeführt werden.
- Partielle Ordnung von Operationen: Sequenz von Operationen, die nicht umsortiert werden dürfen.
- Effekte von Operationen: Garantien, wann Operationen auf geteilten Variablen in anderen Threads sichtbar sind.

Das C++-Speichermodell lehnt sich an seinen Vorgänger, das Java-Speichermodell, an. Im Gegensatz zum Java-Speichermodell erlaubt das C++-Speichermodell den Bruch der sequenziellen Konsistenz. Die Sequenzielle Konsistenz stellt das Default-Verhalten in C++ dar.

Die sequenzielle Konsistenz bietet zwei Garantien an:

- Die Anweisungen eines Programms werden in der Reihenfolge der Sourcecode-Anweisungen ausgeführt.
- Es gibt eine globale Ordnung aller Operationen auf allen Threads.

Die Grundlage des Speichermodells sind atomaren Operationen auf den atomaren Datentypen.

Atomare Datentypen

C++ besitzt die elementaren atomaren Datentypen. Dies sind Wahrheitswerte, Zeichen, Ganzzahlen und Zeiger in vielen Variationen. Dank des Klassen-Templates `std::atomic` lassen sich eigene atomare Datentypen definieren. Atomare Datentypen etablieren Synchronisations- und Ordnungsbedingungen, die auch für nicht-atomare Datentypen gelten.

Die standardisierte Threading-Schnittstelle ist das Herz der Gleichzeitigkeit in C++.

1.1.2 Multithreading

Multithreading besteht in C++ aus den Komponenten Thread, Synchronisationsmechanismen für geteilte Daten, Thread-lokale Daten und Tasks.

Threads

Ein Thread repräsentiert eine unabhängige Ausführungseinheit. Die Ausführungseinheit, die sofort startet, erhält ihr Arbeitspaket in der Form einer aufrufbaren Einheit. Eine aufrufbare Einheit kann eine Funktion, ein Funktionsobjekt oder eine Lambda-Funktion sein.

Der Erzeuger des Threads ist für diesen verantwortlich. Die ausführbare Einheit des neuen Threads endet mit dem Ausführen der aufrufbaren Einheit. Der Erzeuger wartet, bis der erzeugte Thread `t` fertig ist (`t.join()`), oder er trennt sich von seinem erzeugten Thread: `t.detach()`. Ein Thread `[t]` kann noch *joinable* sein, wenn noch keine Operation `t.join()` oder `t.detach()` auf ihn aufgerufen wurde. Ein *joinable* ruft `std::terminate` in seinem Destruktor auf, und das Programm beendet sich.

Ein Thread, der von der Lebenszeit seines Erzeugers getrennt ist, wird gerne Daemon Thread genannt, da er vollkommen im Hintergrund läuft.

Ein `std::thread` ist ein variadic-Template. Das bedeutet, dass er eine beliebige Anzahl von Argumenten annehmen kann. Entweder erhalten die aufrufbare Einheit oder der Thread selbst alle Argumente.

Geteilte Daten

Der Zugriff auf geteilte, veränderliche Daten muss synchronisiert werden, wenn mehr als ein Thread zu einem Zeitpunkt auf die geteilten Daten zugreift. Das gleichzeitige Lesen und Schreiben von geteilten Daten ist ein Data Race und stellt ein undefiniertes Verhalten dar. Mit Mutex und Locks lässt sich koordiniert auf geteilten Daten zugreifen.

Mutex

Mutexe (`mutual exclusion`) sichern den exklusiven Zugriff auf geteilte Daten zu. Der Mutex lockt den kritischen Bereich, zu der die geteilte Variable gehört. C++ bietet fünf verschiedene Mutex an. Diese können rekursiv, versuchsweise, mit und ohne Zeitlimit versuchen, den Mutex zu erhalten. Mutexe können selbst einen Lock zum gleichen Zeitpunkt teilen.

Locks

Ein Mutex sollte in einem Lock gekapselt werden. Ein Lock setzt das RAII-Idiom um. Dabei wird die Lebenszeit des Mutex an die des Locks gebunden. C++ kennt den `std::lock_guard` für den einfachen und std::unique_lock für den anspruchsvollen Workflow wie das explizite Locken und Unlocken seines zugrunde liegenden Mutex.

Thread-sichere Initialisierung von Daten

Falls ein Datum nur lesend verwendet wird, ist es vollkommen ausreichend, dieses Datum lediglich thread-sicher zu initialisieren. C++ bietet mehr Wege an: So werden konstante Ausdrücke thread-sicher initialisiert. Das Gleiche trifft für statische Variablen mit Blockgültigkeit oder die Funktionen std::call_once und std::once_flag.

Thread-lokale Daten

Wird eine Variable als thread-local deklariert, erhält jeder Thread seine eigene Copy des Datums. Damit gibt es kein Teilen. Der Lebenszyklus eines thread-lokalen Datums ist an den Lebenszyklus seines Threads gebunden.

Bedingungsvariablen

Bedingungsvariablen erlauben es Threads, sich über Nachrichten zu synchronisieren. Ein Thread fungiert als Sender, der andere Thread als Empfänger der Nachricht. Dabei blockiert der Empfänger der Nachricht, bis er diese vom Sender erhalten hat. Typischerweise werden Bedingungsvariablen für Producer-Consumer-Workflows eingesetzt. Die Bedingungsvariable kann dabei sowohl als Sender als auch als Empfänger der Nachricht

fungieren. Der richtige Einsatz von Bedingungsvariablen ist sehr trickreich; daher sind Tasks meist die einfachere Lösung.

Tasks

Tasks haben sehr viel mit Threads gemein. Während ein Thread explizit erzeugt werden muss, ist ein Task einfach ein Arbeitspaket, das abgearbeitet wird. Die C++-Laufzeit verwaltet automatisch im Falle des einfachen Tasks std::async deren Lebenszyklus

Tasks entsprechen Datenkanälen zwischen zwei Kommunikationsendpunkten. Sie erlauben die Thread-sichere Kommunikation zwischen Threads. Der Promise, als ein Endpunkt, schiebt Daten in den Datenkanal, der Future als der andere Endpunkt holt diese Daten ab. Die Daten können Werte, Ausnahmen oder Benachrichtigungen sein. Zuzüglich zu `std::async` bietet C++ das Klassen-Template std::promise und std::future an. Beide zusammen erlauben mächtige Arbeitsabläufe.

1.2 C++17: Die parallelen Algorithmen der Standard Template Library

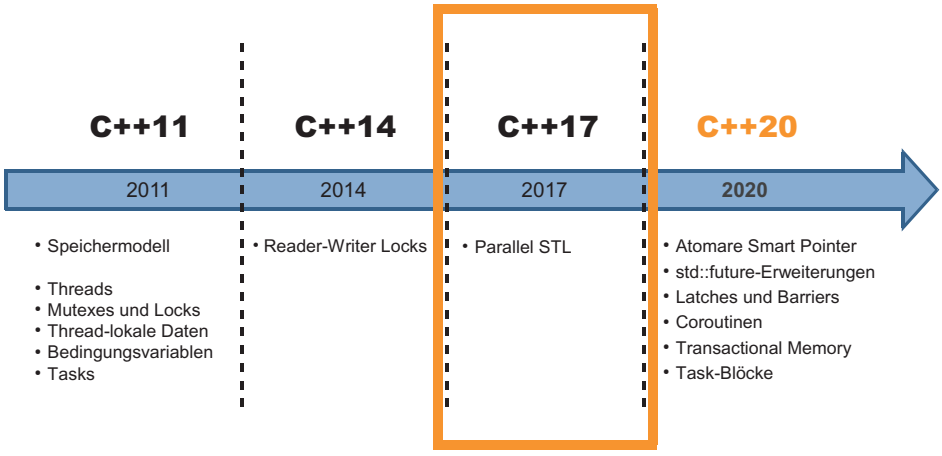

Bild 1.2 Parallele Algorithmen in C++17

Mit C++17 findet ein deutlicher Bruch in der Gleichzeitigkeit statt. Dieser Bruch wird durch die parallelen Algorithmen der Standard Template Library eingeleitet. C++11 und C++14 bot nur die elementaren Bausteine für Gleichzeitigkeit an. Diese Bausteine waren für die Bibliotheks- und Frameworkentwickler konzipiert, besaßen aber nicht die richtige Abstraktion für den Applikationsentwickler. Multithreading in C++11 und C++14 ist die Assemblersprache für Gleichzeitigkeit in C++17!

1.2.1 Ausführungsstrategie

Mit C++17 stehen die meisten der STL-Algorithmen in einer parallelen Implementierung zur Verfügung. Dadurch wird es möglich, einen Algorithmus mit einer sogenannten Ausführungsstrategie aufzurufen. Die Ausführungsstrategie gibt vor, ob der Algorithmus sequenziell (`std::seq`), parallel (`std::par`) oder parallel und vektorisierend (`std::par_unseq`) ausgeführt wird.

1.2.2 Neue Algorithmen

Zuzüglich zu den 69 Algorithmen, die mit C++17 in paralleler oder paralleler und vektorisierter Version zur Verfügung stehen, enthält C++17 acht neue Algorithmen. Die neuen Algorithmen sind für die parallele Reduktion oder Transformation konzipiert.

1.3 Fallstudien

Auf die Theorie zum Speichermodell und zur Multithreading-Schnittstelle folgt die Praxis. Dies wird in Form von drei Fallstudien sein.

1.3.1 Berechnung der Summe eines Vektors

Das Berechnen der Summe eines Vektors ist in vielen Variationen möglich. Die Summe lässt sich sequentiell oder parallel berechnen. Darüber hinaus stellt es einen großen Performanzunterschied dar, ob die Summation auf einer gemeinsamen Variable oder in jedem Thread separat vollzogen wird.

1.3.2 Thread-sichere Initialisierung eines Singletons

Die Thread-sichere Initialisierung eines Singletons ist der klassische Anwendungsfall für eine geteilte Variable, die nur Thread-sicher initialisiert werden muss. C++11 bietet für die Thread-sichere Initialisierung einer Variable mehrere Möglichkeiten an. Die Performanzcharakteristiken unterscheiden sich aber deutlich.

1.3.3 Fortwährende Optimierung mit CppMem

Die fortwährende Optimierung mit CppMem startet mit einem kleinen Programm. Bei jeder Verbesserung wird uns CppMem (siehe http://svr-pes20-cppmem.cl.cam.ac.uk/cppmem/) wertvolle Dienste erweisen und die Frage beantworten: Besitzt das Programm undefiniertes Verhalten?

1.4 C++20: Die concurrent Zukunft

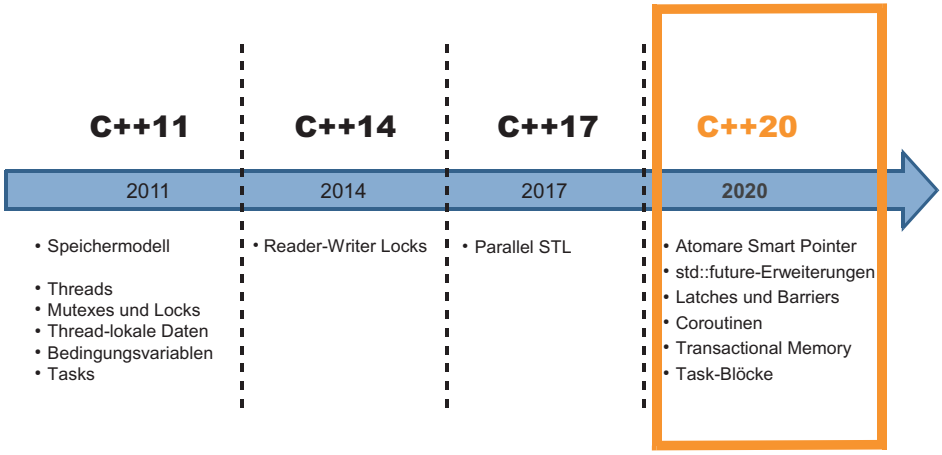

Bild 1.3 C++20

It is difficult to make predictions, especially about the future. (Nils Bohr)

1.4.1 Atomare Smart Pointer

Die Smart Pointer `std::shared_ptr` und `std::weak_ptr` besitzen ein konzeptionelles Problem in Programmen mit mehreren Threads. Sie teilen per Definition veränderliche Daten. Daher sind sie sehr anfällig für Data Races und damit für undefiniertes Verhalten. `std::shared_ptr` und `std::weak_ptr` sichern zwar zu, dass das In- und Dekrementieren des Referenzzähler eine atomare Operation ist und dass die zugrunde liegende Ressource genau einmal gelöscht wird. Beide sichern aber nicht zu, dass der Zugriff auf die zugrunde liegende Ressource Thread-sicher ist. Die neuen atomaren Smart Pointer `std::atomic_shared_ptr` und `std::atomic_weak_ptr` garantieren auch den atomaren Zugriff auf die zugrunde liegende Ressource.

1.4.2 Erweiterte Futures

Tasks, die in Promise und Futures genannt werden, sind in C++11 eingeführt worden und haben viel anzubieten. Sie besitzen aber einen großen Nachteil: Tasks können nicht komponiert werden, um mächtige Arbeitsabläufe darzustellen. Diese Einschränkung gilt nicht mehr für die erweiterten Futures in C++20. Ein erweiterter Future ist dann bereit, wenn sein Vorgänger (`then`) bereit ist oder wenn einer seiner Vorgänger (`when_any`) oder alle seine Vorgänger (`when_all`) bereit sind.

1.4.3 Latches und Barriers

C++14 kennt keine Semaphoren. Semaphoren geben kontrollierten Zugriff auf eine eingeschränkte Anzahl von Ressourcen. Mit C++20 wird dies mit Latches und Barriers möglich sein. Latches und Barriers können eingesetzt werden, um Threads an einem Synchronisationspunkt warten zu lassen, bis der Zähler den Wert 0 erreicht. Der Unterschied zwischen Latches und Barriers ist, dass ein `std::latch` nur einmal verwendet werden kann. Dies trifft nicht auf einen `std::barrier` oder `std::flex_barrier` zu. Im Gegensatz zum einem `std::barrier` kann ein `std::flex_barrier` seinen Zähler nach jeder Iteration anpassen.

1.4.4 Coroutinen

Coroutinen sind Funktionen, die Ausführungen anhalten und wieder fortsetzen können. Dabei behält eine Coroutine ihren Zustand. Coroutinen werden gerne dazu verwendet, kooperativ Multitaskingsysteme, Event Loops, unendliche Listen oder Pipelines zu implementieren.

1.4.5 Transaction Memory

Transactional Memory greift die Idee der Transaktion der Datenbanktheorie auf. Eine Transaktion ist eine Aktion, die die drei ersten Eigenschaften des ACID-Idioms umsetzt: **A**tomicity, **C**onsistency und **I**solation. Die Eigenschaft **D**urability lässt sich nicht auf Transaktionen in Software anwenden. Der neue Standard wird zwei Formen von Transactional Memory anbieten: synchronized-Blöcke und atomic-Blöcke. Beide werden eine Transactional Memory bilden und sich verhalten, als ob sie durch einen globalen Lock koordiniert werden. Im Gegensatz zu synchronized-Blöcken können atomic-Blöcke keinen *transaction-unsave* -Sourcecode ausführen.

1.4.6 Task-Blöcke

Task-Blöcke setzen das Fork-Join-Paradigma um. Das Bild 1.4 illustriert die zentrale Idee von Task-Blöcken. In der Fork-Phase werden Tasks gestartet, die in der Join-Phase wieder synchronisiert werden.

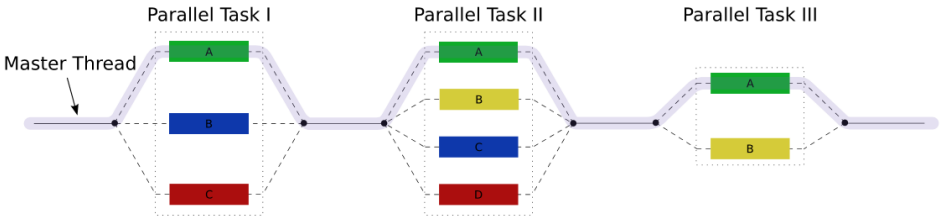

Bild 1.4 Task-Blöcke (Copyright by Wikipedia user A1 – w:en:File:Fork_join.svg, CC BY 3.0, https://commons.wikimedia.org/w/index.php?curid=32004077)

1.5 Herausforderungen

Programme, die mehrere Threads verwenden, besitzen eine inhärent große Komplexität. Dies Aussage trifft verstärkt zu, wenn das Programm mit C++11- und C++14- Features implementiert ist. Daher geht dieses Kapitel explizit auf die Herausforderungen dieser besonderen Domäne ein und widmet sich vor allem deren Fallen und Fallstricken. Diese besitzen Namen wie Data Race, Race Conditions und Deadlock.

1.6 Best Practice

Programme, die mehrere Threads verwenden, besitzen eine inhärent große Komplexität. Daher helfen Best Practices wie minimieren Sie das Teilen von veränderlichen Daten, verwenden Sie statische Codeanalyse Werkzeuge oder ziehen Sie unveränderliche Daten vor, diese Komplexität zu meistern.

1.7 Zeitbibliothek

Die Zeitbibliothek ist eine zentrale Komponente der Features rund um Gleichzeitigkeit in C++. Oft wird ein Thread für eine bestimmte Zeitdauer oder bis zu einem Zeitpunkt schlafen gelegt. Die Zeitbibliothek in C++11 besteht aus den Komponenten Zeitdauer, Zeitpunkt und Zeitgeber.

1.8 Glossar

Das Kapitel Glossar enthält eine nicht vollständige Erläuterung der wichtigsten Begriffe dieses Buchs.

TEIL II

Die Details

2 Das Speichermodell

Die Grundlage von Multithreading in C++ ist das wohldefinierte Speichermodell. Aus Perspektive des Lesers besteht es aus zwei Aspekten. Einerseits besitzt es eine sehr hohe Komplexität, die oft unserer Intuition widerspricht. Andererseits hilft es, deutlich tiefere Einsichten in die Multithreading-Herausforderungen zu bekommen. In der ersten Annäherung definiert das Speichermodell einen Vertrag.

2.1 Der Vertrag

Der Vertrag besteht zwischen dem Programmierer und dem System. Das System besteht aus dem Compiler, der den Maschinencode erzeugt, und dem Prozessor, der den Maschinencode ausführt. Dazu kommen noch die verschiedenen Caches, die den Zustand des Programms speichern. Das Ergebnis ist – wenn alles gut läuft – eine wohldefinierte ausführbare Datei, die optimal an die Plattform angepasst ist. Um genau zu sein, gibt nicht nur einen Vertrag, sondern viele feine Abstufungen des Vertrags. Oder anders ausgedrückt: Je schwächer die Regeln sind, denen der Programmierer zu folgen hat, desto mehr Potenzial besitzt das System, eine hoch optimierte, ausführbare Datei zu erzeugen.

Es gibt eine einfache Faustregel: Je strenger der Vertrag ausgelegt wird, desto weniger Freiheit besitzt das System, eine optimierte ausführbare Datei zu erzeugen. Traurig, aber die konträre Aussage gilt leider nicht. Falls der Programmierer einen sehr schwachen Vertrag oder auch ein Speichermodell anwendet, besteht sehr viel Potenzial, das System zu optimieren. Die Konsequenz ist aber, dass das Programm nur noch durch ein paar weltweit bekannte Experten zu beherrschen ist … Experten, zu denen vermutlich weder die Leser noch der Schreiber dieser Zeilen gehört.

Grob geschrieben, besteht der Vertrag in C++11 aus drei Abstufungen (Bild 2.1).

Vor C++11 gab es nur einen Vertrag. Die Spezifikation von C++ enthielt weder Multithreading noch atomare Datentypen. Das System bestand nur aus einem Kontrollfluss und besaß daher nur sehr eingeschränkte Möglichkeiten, eine optimierte ausführbare Datei zu erzeugen. Der zentrale Punkt des System war die Garantie, dass das beobachtbare Verhalten des Programms der Sequenz der Sourcecode-Anweisungen entsprach. Natürlich bedeutet dies, dass es kein Speichermodell gab. Statt dessen gab es das Konzept eines Sequenzpunkts. Ein Sequenzpunkt ist eine Stelle im Sourcecode, an dem die Effekte aller vorheri-

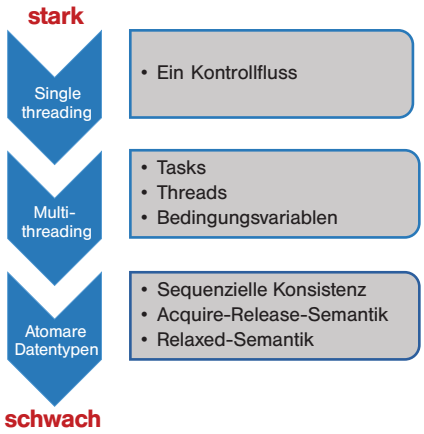

Bild 2.1 Drei Abstufungen des Vertrags

gen Instruktionen beobachtbar sein müssen. Beobachtbar meint in diesem Fall, dass alle Instruktionen vor dem Sequenzpunkt bereits ausgeführt wurden. Zum Beispiel ist der Beginn oder das Ende einer Funktion ein Sequenzpunkt. Wenn eine Funktion mit zwei Argumenten aufgerufen wird, gibt der C++-Standard keine Garantie, welches Argument als Erstes ausgewertet wird. Das Verhalten ist daher nicht spezifiziert. Der Grund ist naheliegend: Der Komma-Operator ist kein Sequenzpunkt in C++.

Mit C++11 hat sich alles verändert. C++11 ist der erste C++-Standard, der sich mehrerer Threads bewusst ist. Der Grund für das wohldefinierte Verhalten von Threads ist das C++-Speichermodell, das sehr stark vom Java-Speichermodell beeinflusst wurde. Es ist aber typisch für C++, dass das C++-Speichermodell ein paar Schritte weiter geht als sein Java-Pendant. Der Programmierer hat ein paar Regeln mit geteilten Daten einzuhalten, um ein wohldefiniertes Programm zu erhalten. Das Programm besitzt undefiniertes Verhalten, falls zumindest ein Data Race existiert. Die Gefahr von Data Races ist in Multithreading-Programmen stets gegenwärtig, falls geteilte, veränderliche Daten verwendet werden. Tasks sind deutlich einfacher zu verwenden als Threads oder Bedingungsvariablen.

Mit atomaren Datentypen beginnt die Domäne von Experten. Diese Aussage wird umso offensichtlicher, je mehr das Speichermodell abgeschwächt wird. Kommen atomare Datentypen zum Einsatz, werden diese Sourcecode-Abschnitte lock-free genannt. Dieser Abschnitt handelt von den starken und schwachen Regeln des Vertrags. Tatsächlich wird die Sequenzielle Konsistenz als `strong memory model` und die Relaxed-Semantik als `weak memory model` bezeichnet.

2.1.1 Die Grundlagen

Das C++-Speichermodell hat sich mit drei Aspekten auseinanderzusetzen.

- Atomare Operationen: Operationen, die ohne Unterbrechung ausgeführt werden.
- Partielle Ordnung von Operationen: Sequenz von Operationen, die nicht umsortiert werden dürfen.