



Jörg Hettel · Manh Tien Tran

Nebenläufige Programmierung mit Java

Konzepte und Programmiermodelle
für Multicore-Systeme

dpunkt.verlag



Jörg Hettel studierte Theoretische Physik und promovierte am Institut für Informationsverarbeitung und Kybernetik an der Universität Tübingen. Nach seiner Promotion war er als Berater bei nationalen und internationalen Unternehmen tätig. Er begleitete zahlreiche Firmen bei der Einführung von objektorientierten Technologien und übernahm als Softwarearchitekt Projektverantwortung. Seit 2003 ist er Professor an der Hochschule Kaiserslautern am Standort Zweibrücken. Seine aktuellen Arbeitsgebiete sind u.a. verteilte internetbasierte Transaktionssysteme und die Multicore-Programmierung.



Manh Tien Tran studierte Informatik an der TU Braunschweig. Von 1987 bis 1995 war er wissenschaftlicher Mitarbeiter am Institut für Mathematik der Universität Hildesheim, wo er 1995 promovierte. Von 1995 bis 1998 war er als Softwareentwickler bei BOSCH Blaupunkt beschäftigt. 1999 wechselte er zu Harman Becker und war dort bis 2000 für Softwarearchitekturen zuständig. Seit 2000 ist er Professor an der Hochschule Kaiserslautern am Standort Zweibrücken. Seine aktuellen Arbeitsgebiete sind Frameworks, Embedded-Systeme und die Multicore-Programmierung.

Jörg Hettel · Manh Tien Tran

Nebenläufige Programmierung mit Java

Konzepte und Programmiermodelle für
Multicore-Systeme



dpunkt.verlag

Prof. Dr. Jörg Hettel
joerg.hettel@hs-kl.de

Prof. Dr. Manh Tien Tran
manhtien.tran@hs-kl.de

Lektorat: Christa Preisendanz
Copy-Editing: Ursula Zimpfer, Herrenberg
Satz: Jörg Hettel, Manh Tien Tran
Herstellung: Frank Heidt
Umschlaggestaltung: Helmut Kraus, www.exclam.de
Druck und Bindung: M.P. Media-Print Informationstechnologie GmbH, 33100 Paderborn

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.

ISBN:
Print 978-3-86490-369-4
PDF 978-3-96088-012-7
ePub 978-3-96088-013-4
mobi 978-3-96088-014-1

1. Auflage 2016
Copyright © 2016 dpunkt.verlag GmbH
Wiebinger Weg 17
69123 Heidelberg

Die vorliegende Publikation ist urheberrechtlich geschützt. Alle Rechte vorbehalten. Die Verwendung der Texte und Abbildungen, auch auszugsweise, ist ohne die schriftliche Zustimmung des Verlags urheberrechtswidrig und daher strafbar. Dies gilt insbesondere für die Vervielfältigung, Übersetzung oder die Verwendung in elektronischen Systemen.

Es wird darauf hingewiesen, dass die im Buch verwendeten Soft- und Hardware-Bezeichnungen sowie Markennamen und Produktbezeichnungen der jeweiligen Firmen im Allgemeinen warenzeichen-, marken- oder patentrechtlichem Schutz unterliegen.

Alle Angaben und Programme in diesem Buch wurden mit größter Sorgfalt kontrolliert. Weder Autor noch Verlag können jedoch für Schäden haftbar gemacht werden, die in Zusammenhang mit der Verwendung dieses Buches stehen.

5 4 3 2 1 0

Vorwort

Warum ist das Schreiben von nebenläufiger Software so schwer? Zeitgleiche Abläufe beherrschen doch unseren Alltag. Wir arbeiten in Teams, koordinieren unsere Termine und übernehmen oder verteilen Aufgaben. In der Regel kommen wir mit dieser Art der Parallelität ganz gut zurecht.

Die uns vertraute Parallelität erweist sich bei der Entwicklung von Softwaresystemen als schwer zugänglich. Das liegt sicherlich mit daran, dass wir dabei immer das Ganze im Blick haben und Abläufe immer wieder über neue Koordinationsregeln steuern müssen. Darüber hinaus haben wir es technikbedingt mit einer anderen Art der Beschreibung von Parallelität zu tun.

Die Abstraktion der nebenläufigen Programmierung ist bei vielen Konzepten der *Thread*, ein Kontrollfluss bzw. -faden, der unabhängig von anderen agiert und durch einen Programmcode gesteuert wird. Diese Beschreibungsweise hat ihren Ursprung in der sequenziellen Programmierung, bei der es genau einen Ablaufstrang gibt. Leider ist es für uns auch im normalen Leben unmöglich, viele gleichzeitige, obwohl sequenzielle Vorgänge zu bewältigen. Diese Parallelitätsabstraktion ist intuitiv nicht leicht zugänglich; wir denken im Alltag nicht in *Threads*.

Der Umgang mit Threads birgt deshalb zahlreiche Fehlerquellen. Viele Multithreaded-Anwendungen enthalten Anomalien, die erst nach Monaten oder Jahren auftreten (siehe z. B. [36]). Nicht reproduzierbare Programmabstürze oder Verklemmungen, die häufig erst spät im Produktivbetrieb auftreten, sind typische Symptome dafür.

Um einfache, sichere Programmiermodelle zu ermöglichen, versucht man auf den in der Sprache vorhandenen primitiven Mechanismen *Abstraktionskonzepte* und *Frameworks* aufzubauen. Auf diesem Gebiet hat sich in den letzten Jahren sehr viel getan. Insbesondere wurde die Programmiersprache Java um viele solcher Konzepte erweitert.

Die nebenläufige Programmierung ist keine neue Domäne und es existiert auch schon viel Literatur hierzu. Einen guten Überblick über diesen komplexen Themenbereich findet man z. B. in dem Buch *Multicore-Software* von Urs Gleim und Tobias Schüle [15]. Im Bereich der Java-Programmierung gilt nach wie vor das Buch von Doug Lea *Concurrent Programming in Java: Design Principles and Patterns* [34] als Standard-

werk. Viele Ideen aus diesem Buch wurden sukzessive in die einzelnen Java-Versionen übernommen. Als Fortsetzung dieses Werks gilt das 2005 erschienene Buch *Java Concurrency in Practice* von Brian Goetz et al. [16], das ausführlich das Java 5 *Concurrency-API* diskutiert. Gute Beiträge zu vielen einzelnen Themen gibt es z. B. von Klaus Krefl und Angelika Langer [30] oder Heinz Kabutz [27].

Mit dem vorliegenden Buch möchten wir an diese Literatur anknüpfen und eine fundierte Einführung in die nebenläufige Programmierung mit Java geben und insbesondere auch die in den letzten Jahren eingeführten Konzepte und Frameworks detailliert beschreiben. Das Buch richtet sich vor allem an erfahrene Softwareentwickler sowie fortgeschrittene Studenten, die nebenläufige Konzepte in Projekten einsetzen möchten.

Wir hoffen, dass Ihnen das Buch *Nebenläufige Programmierung mit Java* gefällt und vor allem, dass es Ihnen ein guter Ratgeber ist.

Aufbau des Buches

Das Buch besteht aus fünf Teilen. Im ersten Teil werden die für die nebenläufige Programmierung grundlegenden Konzepte besprochen. Es wird der Thread-Mechanismus eingeführt und die Koordinierung nebenläufiger Programmflüsse durch rudimentäre *Low-Level-Synchronisationsmechanismen* erläutert. Im Wesentlichen sind dies die Verfahrensweisen, die seit Einführung von Java im Sprachumfang zur Verfügung stehen. Die Basiskonzepte bilden die Grundlage für die weiteren Teile des Buches.

Mit dem Aufkommen von Multicore-Prozessoren und den damit verbundenen Möglichkeiten ist die nebenläufige Programmierung immer mehr in den Fokus der Anwendungsentwicklung gerückt. Da die rudimentären Konzepte sehr leicht zu fehleranfälligen Implementierungen führen, wurde mit Java 5 ein umfangreiches *Concurrency-API* eingeführt, das mit den folgenden Versionen immer wieder ausgebaut wurde und wird.

Im Teil zwei werden verschiedene weiterführende Konzepte, wie *Threadpools*, *Futures*, *Atomic-Variablen* und *Locks*, vorgestellt.

Im dritten Teil werden ergänzende Synchronisationsmechanismen zur Koordinierung mehrerer Threads eingeführt. Neben dem `Exchanger` sind dies die Klassen `CountDownLatch`, `CyclicBarrier` und `Phaser`.

Teil vier bespricht die Parallelisierungsframeworks, mit denen auf einfache Art und Weise nebenläufige Programme erstellt werden können. Die Frameworks übernehmen hier im Wesentlichen die Thread-Koordination und -Synchronisation. Im Einzelnen werden das *ForkJoin-Framework*, die *Parallel Streams* und die Klasse `CompletableFuture` besprochen. Das *ForkJoin-Framework* erlaubt die Parallelisierung von *Divide-and-Conquer-Algorithmen* und parallele Streams die zeitgleiche Verarbeitung von Datensammlungen, wie z. B. `Collections`. Die Klasse `CompletableFuture` ent-

spricht einem Framework zur Erstellung von asynchronen Abläufen und ist eine Erweiterung des *Future*-Mechanismus um sogenannte *push*-Methoden.

Der fünfte Teil widmet sich der Anwendung der vorgestellten Konzepte und Klassen. Hierbei wurden die Beispiele aus verschiedenen Anwendungsgebieten ausgewählt. Des Weiteren gehen wir kurz auf das Thread-Konzept von JavaFX und Android ein. Abschließend stellen wir das Programmiermodell mit Aktoren vor, wobei hier das Akka-Framework benutzt wird, da im Java-Standard selbst (noch) kein solches Framework vorhanden ist.

Im Anhang geben wir der Vollständigkeit halber einen kurzen Ausblick auf Java 9, das bezüglich des *Concurrency*-API kleine Neuerungen bringt.

Vorausgesetzt werden gute Java-Kenntnisse, und erste Erfahrungen im Umgang mit *Lambda*-Ausdrücken wären wünschenswert. Als ergänzende Literatur empfehlen wir das Buch von Michael Inden [25], von dem wir auch einige Praxistipps übernommen haben. Die Streams von Java 8 und die in dem Zusammenhang benötigten funktionalen Interfaces werden in Kapitel 14 eingeführt.

Bemerkungen zu den Codebeispielen

Wir haben versucht, die Beispiele »so einfach wie möglich und so komplex wie notwendig« zu halten. Insbesondere sind die Fallbeispiele im fünften Teil noch nicht »voll praxistauglich«. Der benutzte *Coding Style* ist zum großen Teil der Buchform angepasst, was z. T. herausfordernd ist, da hier die Zeilenbreite sehr beschränkt ist. Das macht insbesondere die Darstellung von `Stream`- und `CompletableFuture`-Operationen oft schwierig. Wenn möglich, haben wir für das Verständnis nicht relevanten Code weggelassen. Insbesondere wird stets auf die `import`-Anweisungen verzichtet. Alle Codebeispiele findet man auch auf unserer Webseite zum Download. Bei den besprochenen APIs haben wir keinen Wert auf Vollständigkeit gelegt, sondern versucht, das Wesentliche zu extrahieren. Mit dem hier erworbenen Verständnis sollte man keine Probleme haben, die API-Dokumentation zu verstehen. Ein Blick in die Dokumentation ist immer zu empfehlen, da mittlerweile auch Verwendungsbeispiele aufgenommen wurden.

Danksagungen

Ein herzliches Dankeschön geht an die Mitarbeiter des dpunkt.verlags und insbesondere an Frau Christa Preisendanz, die die Fertigstellung des Buches professionell begleitet haben. Wir möchten uns auch bei unseren Studenten und den Reviewern, insbesondere Prof. Dr. Schiedermeier und Michael Inden, für die kritische Prüfung und kompetenten Hinweise bedanken. Zu guter Letzt geht auch ein Dank an unsere Familien für die Unterstützung und die Geduld.

Trotz sorgfältiger Prüfung wird das Buch wahrscheinlich leider noch Fehler enthalten, für die ausschließlich die Autoren verantwortlich sind. Falls Sie Fehler finden, lassen Sie es uns bitte wissen.

Jörg Hettel und Manh Tien Tran
Zweibrücken, Juni 2016

<http://www.hs-kl.de/java-concurrency>

Inhaltsverzeichnis

1	Einführung	1
1.1	Dimensionen der Parallelität	1
1.2	Parallelität und Nebenläufigkeit	2
1.2.1	Die Vorteile von Nebenläufigkeit	3
1.2.2	Die Nachteile von Nebenläufigkeit	3
1.2.3	Sicherer Umgang mit Nebenläufigkeit	4
1.3	Maße für die Parallelisierung	4
1.3.1	Die Gesetze von Amdahl und Gustafson	4
1.3.2	Work-Span-Analyse	6
1.4	Parallelitätsmodelle	7
I Grundlegende Konzepte		9
2	Das Thread-Konzept von Java	11
2.1	Der main-Thread	11
2.2	Erzeugung und Starten von Threads	12
2.2.1	Thread-Erzeugung durch Vererbung	13
2.2.2	Thread-Erzeugung mit Runnable-Objekten	16
2.3	Der Lebenszyklus von Threads	19
2.3.1	Beendigung eines Threads	20
2.3.2	Auf das Ende eines Threads warten	21
2.3.3	Aktives Beenden von Threads	21
2.3.4	Unterbrechung mit interrupt	24
2.3.5	Thread-Zustände	26
2.4	Weitere Eigenschaften eines Thread-Objekts	27
2.4.1	Thread-Priorität	27
2.4.2	Daemon-Eigenschaft	28
2.5	Exception-Handler	29
2.6	Zusammenfassung	31

3	Konkurrierende Zugriffe auf Daten	33
3.1	Ein einleitendes Beispiel	33
3.2	Java-Speichermodell	34
3.2.1	Stacks und Heap	35
3.2.2	Speicher auf der Hardwareebene	37
3.2.3	Probleme mit gemeinsam nutzbaren Daten	38
3.2.4	Sequenzielle Konsistenz	39
3.2.5	Thread-sichere Daten und unveränderliche Objekte	41
3.3	Unveränderbare Objekte	42
3.4	Volatile-Attribute	43
3.5	Final-Attributte	45
3.6	Thread-lokale Daten	46
3.7	Fallstricke	49
3.8	Zusammenfassung	50
4	Elementare Synchronisationsmechanismen	51
4.1	Schlüsselwort synchronized	51
4.1.1	Synchronized-Methoden	51
4.1.2	Synchronized-Blöcke	53
4.1.3	Beispiel: Thread-sicheres Singleton	54
4.1.4	Monitorkonzept bei Java	56
4.2	Fallstricke	57
4.3	Zusammenfassung	62
5	Grundlegende Thread-Steuerung	63
5.1	Bedingungsvariablen und Signalisieren	63
5.2	Regeln zum Umgang mit wait, notify und notifyAll	69
5.3	Zusammenfassung	72

II Weiterführende Konzepte **73**

6	Threadpools	75
6.1	Das Poolkonzept und die Klasse Executors	75
6.1.1	Executors mit eigener ThreadFactory	78
6.1.2	Explizite ThreadPoolExecutor-Erzeugung	78
6.1.3	Benutzerdefinierter ThreadPoolExecutor	79
6.2	Future- und Callable-Schnittstelle	80
6.2.1	Callable, Future und FutureTask	81
6.2.2	Callable, Future und ExecutorService	81
6.3	Callable und ThreadPoolExecutor	84
6.4	Callable und ScheduledThreadPoolExecutor	88
6.5	Callable und ForkJoinPool	88

6.6	Exception-Handling	90
6.7	Tipps für das Arbeiten mit Threadpools	92
6.8	Zusammenfassung	93
7	Atomic-Variablen	95
7.1	Compare-and-Set-Operation	96
7.2	Umgang mit Atomic-Variablen	97
	7.2.1 Atomic-Skalare	97
	7.2.2 Atomic-Referenzen	100
7.3	Accumulator und Adder in Java 8	102
7.4	Zusammenfassung	104
8	Lock-Objekte und Semaphore	105
8.1	Lock-Objekte	106
	8.1.1 Das Lock-Interface	106
	8.1.2 ReentrantLock	109
	8.1.3 Das Condition-Interface	111
	8.1.4 ReadWriteLock	115
	8.1.5 StampedLock	117
8.2	Semaphore	120
8.3	Zusammenfassung	123
9	Thread-sichere Container	125
9.1	Collection-Typen	125
9.2	Thread-sichere Collections	127
	9.2.1 Synchronisierte Collections	127
	9.2.2 Unmodifiable Collections	129
	9.2.3 Concurrent Collections	130
9.3	Zusammenfassung	135

III	Ergänzende Synchronisationsmechanismen	137
------------	---	------------

10	Exchanger und BlockingQueue	139
10.1	Exchanger	139
10.2	Queues	143
10.3	Das Erzeuger-Verbraucher-Muster	146
10.4	Varianten	149
	10.4.1 Pipeline von Erzeugern und Verbrauchern	149
	10.4.2 Erzeuger-Verbraucher-Muster mit Empfangsbestätigung	150
	10.4.3 Erzeuger-Verbraucher-Muster mit Work-Stealing	151
10.5	Zusammenfassung	157

11	CountDownLatch und CyclicBarrier	159
11.1	CountDownLatch	159
11.2	CyclicBarrier	162
11.3	Zusammenfassung	167
12	Phaser	169
12.1	Das Konzept des Phasers	169
	12.1.1 Phaser als CountDownLatch	170
	12.1.2 Phaser als CyclicBarrier	173
12.2	Phaser als variable Barriere	174
12.3	Zusammenspiel mit dem ForkJoin-Threadpool	178
12.4	Zusammenfassung	179
IV	Parallelisierungsframeworks	181
13	Das ForkJoin-Framework	183
13.1	Grundprinzip des ForkJoin-Patterns	183
13.2	Programmiermodell	184
	13.2.1 Einsatz von RecursiveAction	186
	13.2.2 Einsatz von RecursiveTask	189
	13.2.3 Einsatz von CountedCompleter	191
13.3	Work-Stealing-Verfahren	194
13.4	Zusammenfassung	197
14	Parallele Array- und Stream-Verarbeitung	199
14.1	Parallele Array-Verarbeitung	199
	14.1.1 Parallele Transformation	199
	14.1.2 Paralleles Sortieren	200
	14.1.3 Parallele Präfixbildung	201
14.2	Funktionsprinzip der Stream-Verarbeitung	203
	14.2.1 Funktionale Interfaces	204
	14.2.2 Erzeugung von Streams	205
	14.2.3 Transformations- und Manipulationsoperationen	208
	14.2.4 Auswertungen von Streams	211
	14.2.5 Eigenschaften und Operationsoptimierung	216
14.3	Parallele Stream-Verarbeitung: Datenparallelität	217
	14.3.1 Arbeitsweise und korrekte Benutzung	218
	14.3.2 Parallele Reduzierer	220
	14.3.3 Parallele Collectoren	223
	14.3.4 Funktionsweise von Spliteratoren	228
	14.3.5 Benutzerdefinierte Spliteratoren	230
14.4	Zusammenfassung	235

15	CompletableFuture	237
15.1	CompletableFuture als Erweiterung des Future-Patterns	237
15.2	Design von asynchronen APIs	241
	15.2.1 Asynchrone APIs mit Future	242
	15.2.2 Asynchrone APIs mit CompletableFuture	242
15.3	Asynchrone Verarbeitung: Task-Parallelität	244
	15.3.1 Das Starten einer asynchronen Verarbeitung	244
	15.3.2 Definition einer asynchronen Verarbeitungskette	245
15.4	Das Arbeiten mit CompletableFuture	247
	15.4.1 Das Konzept des CompletionStage	248
	15.4.2 Lineare Kompositionsmöglichkeiten	249
	15.4.3 Verzweigen und Vereinen	252
	15.4.4 Synchronisationsbarrieren	256
15.5	Fehlerbehandlung und Abbruch einer Verarbeitung	257
15.6	Zusammenfassung	259

V	Fallbeispiele	261
----------	----------------------	------------

16	Asynchrones Logging	263
16.1	Lösung mit Thread-lokalen Daten	264
16.2	Verbesserte Version (Exchanger)	266
17	Datenstrukturen in Multithreaded-Umgebungen	271
17.1	Liste als sortierte Menge	271
17.2	Blockierende Lösungen (Locks)	276
	17.2.1 Grobgranulare Synchronisierung	276
	17.2.2 Feingranulare Synchronisierung	276
	17.2.3 Optimistische Synchronisierung	279
17.3	Lockfreie Lösung (AtomicMarkableReference)	280
18	The Dining Philosophers Problem	287
18.1	Basialgorithmus	288
18.2	Lösungsvarianten (Semaphore und Lock)	288
	18.2.1 Lösung mit einem Semaphor	289
	18.2.2 Lösung mit asymmetrischer Lock-Anforderung	290
	18.2.3 Lösung mithilfe eines Koordinators	291
	18.2.4 Lösung mit asymmetrischer Wait-Release-Strategie	293
19	Minimal aufspannende Bäume	295
19.1	Graphen und Spannbäume	295
19.2	Der Prim-Algorithmus	297
	19.2.1 Funktionsweise des Algorithmus	297

19.2.2	Implementierung des Algorithmus	299
19.3	Parallelisierung (Phaser)	301
20	Mergesort	305
20.1	Funktionsprinzip des Algorithmus	305
20.2	Parallelisierung (ForkJoin-Framework)	307
21	Der k-Mean-Clusteralgorithmus	309
21.1	Der k-Mean-Algorithmus	309
21.2	Parallelisierung (Parallel Streams)	311
21.2.1	Datenmodell	311
21.2.2	Hilfsmethoden	311
21.2.3	Implementierung	312
21.2.4	Variante mit benutzerdefiniertem Collector	316
22	RSA-Schlüsselerzeugung	321
22.1	Verfahren für die Schlüsselerzeugung	321
22.2	Parallelisierung (CompletableFuture)	323
23	Threads bei JavaFX	327
23.1	Ein einfaches Beispiel	327
23.2	JavaFX-Concurrent-API	329
24	Handler-Konzept bei Android	335
24.1	UI-Thread und nebenläufige Aktivitäten	335
24.2	Messages, Message-Queue, Looper	336
24.3	Handler	338
25	Aktoren	341
25.1	Aktorenmodell	341
25.2	Beispielimplementierung mit Akka	342
25.2.1	Nachrichten	343
25.2.2	Beteiligte Aktoren	345
25.2.3	Starten der Anwendung	347
VI	Anhang	349
A	Ausblick auf Java 9	351
A.1	Die Flow-Interfaces	351
	Literaturverzeichnis	357
	Index	361

1 Einführung

Die meisten Computer können heute verschiedene Anweisungen parallel abarbeiten. Um diese zur Verfügung stehende Ressource auszunutzen, müssen wir sie bei der Softwareentwicklung entsprechend berücksichtigen. Die nebenläufige Programmierung wird deshalb häufiger eingesetzt. Der Umgang und die Koordinierung von *Threads* gehören heute zum Grundhandwerk eines guten Entwicklers.

1.1 Dimensionen der Parallelität

Bei Softwaresystemen gibt es verschiedene Ebenen, auf denen Parallelisierung eingesetzt werden kann bzw. bereits eingesetzt wird. Grundsätzlich kann zwischen Parallelität auf der Prozessorebene und der Systemebene unterschieden werden [26, 15]. Auf der Prozessorebene lassen sich die drei Bereiche *Pipelining* (Fließbandverarbeitung), superskalare Ausführung und Vektorisierung für die Parallelisierung identifizieren.

Auf der Systemebene können je nach Prozessoranordnung und Zugriffsart auf gemeinsam benutzte Daten folgende Varianten unterschieden werden:

- Bei *Multinode-Systemen* wird die Aufgabe über verschiedene Rechner hinweg verteilt. Jeder einzelne Knoten (in der Regel ein eigenständiger Rechner) hat seinen eigenen Speicher und Prozessor. Man spricht in diesem Zusammenhang von verteilten Anwendungen.
- Bei *Multiprocessor-Systemen* ist die Anwendung auf verschiedene Prozessoren verteilt, die sich in der Regel alle auf demselben Rechner (Mainboard) befinden und die alle auf denselben Hauptspeicher zugreifen, wobei die Zugriffszeiten nicht einheitlich sind. Jeder Prozessor hat darüber hinaus auch noch verschiedene Cache-Levels. Solche Systeme besitzen häufig eine sogenannte NUMA-Architektur (*Non-Uniform Memory Access*).
- Bei *Multicore-Systemen* befinden sich verschiedene Rechenkerne in einem Prozessor, die sich den Hauptspeicher und zum Teil auch Caches teilen. Der Zugriff auf den Hauptspeicher ist von allen Kernen

gleich schnell. Man spricht in diesem Zusammenhang von einer UMA-Architektur (*Uniform Memory Access*).

Neben den hier aufgeführten allgemeinen Unterscheidungsmerkmalen gibt es noch weitere, herstellerspezifische Erweiterungsebenen. Genannt sei hier z. B. das von Intel eingeführte Hyper-Threading. Dabei werden Lücken in der Fließbandverarbeitung mit Befehlen von anderen Prozessen möglichst aufgefüllt.

Hinweis

In dem vorliegenden Buch werden wir uns ausschließlich mit den Konzepten und Programmiermodellen für Multicore- bzw. Multiprocessor-Systeme mit Zugriff auf einen gemeinsam benutzten Hauptspeicher befassen, wobei wir auf die Besonderheiten der NUMA-Architektur nicht eingehen. Bei Java hat man außer der Verwendung der beiden VM-Flags `-XX:+UseNUMA` und `-XX:+UseParallelGC` kaum Einfluss auf das Speichermanagement.

1.2 Parallelität und Nebenläufigkeit

Zwei oder mehrere Aktivitäten (*Tasks*) heißen *nebenläufig*, wenn sie zeitgleich bearbeitet werden können. Dabei ist es unwichtig, ob zuerst der eine und dann der andere ausgeführt wird, ob sie in umgekehrter Reihenfolge oder gleichzeitig erledigt werden. Sie haben keine kausale Abhängigkeit, d.h., das Ergebnis einer Aktivität hat keine Wirkung auf das Ergebnis einer anderen und umgekehrt. Das Abstraktionskonzept für Nebenläufigkeit ist bei Java der *Thread*, der einem eigenständigen Kontrollfluss entspricht.

Besitzt ein Rechner mehr als eine CPU bzw. mehrere Rechenkerne, kann die Nebenläufigkeit parallel auf Hardwareebene realisiert werden. Dadurch besteht die Möglichkeit, die Abarbeitung eines Programms zu beschleunigen, wenn der zugehörige Kontrollfluss nebenläufige Tasks (Aktivitäten) beinhaltet. Dabei können moderne Hardware und Übersetzer nur bis zu einem gewissen Grad automatisch ermitteln, ob Anweisungen sequenziell oder parallel (gleichzeitig) ausgeführt werden können. Damit Programme die Möglichkeiten der Multicore-Prozessoren voll ausnutzen können, müssen wir die Parallelität explizit im Code berücksichtigen.

Die nebenläufige bzw. parallele Programmierung beschäftigt sich zum einen mit Techniken, wie ein Programm in einzelne, nebenläufige Abschnitte/Teilaktivitäten zerlegt werden kann, zum anderen mit den verschiedenen Mechanismen, mit denen nebenläufige Abläufe synchronisiert und gesteu-

ert werden können. So schlagen z. B. Mattson et al. in [37] ein »pattern-basiertes« Vorgehen für das Design paralleler Anwendungen vor. Ähnliche Wege werden auch in [7] oder [38] aufgezeigt. Spezielle Design-Patterns für die nebenläufige Programmierung findet man in [15, 38, 42, 45].

1.2.1 Die Vorteile von Nebenläufigkeit

Der Einsatz von Nebenläufigkeit ermöglicht die Anwendung verschiedener neuer Programmierkonzepte. Der offensichtlichste Vorteil ist die Steigerung der Performance. Auf Maschinen mit mehreren CPUs kann zum Beispiel das Sortieren eines großen Arrays auf mehrere Threads verteilt werden. Dadurch kann die zur Verfügung stehende Rechenleistung voll ausgenutzt und somit die Leistungsfähigkeit der Anwendung verbessert werden. Ein weiterer Aspekt ist, dass Threads ihre Aktivitäten unterbrechen und wiederaufnehmen können. Durch Auslagerung der blockierenden Tätigkeiten in separate Threads kann die CPU in der Zwischenzeit andere Aufgaben erledigen. Hierdurch ist es möglich, asynchrone Schnittstellen zu implementieren und somit die Anwendung reaktiv zu halten. Dieser Gesichtspunkt gewinnt immer mehr an Bedeutung.

1.2.2 Die Nachteile von Nebenläufigkeit

Der Einsatz von Nebenläufigkeit hat aber nicht nur Vorteile. Er kann unter Umständen sogar mehr Probleme verursachen, als damit gelöst werden. Programmcode mit Multithreading-Konzepten ist nämlich oft schwer zu verstehen und mit hohem Aufwand zu warten. Insbesondere wird das Debugging erschwert, da die CPU-Zuteilung an die Threads nicht deterministisch ist und ein Programm somit jedes Mal verschieden verzahnt abläuft.

Parallel ablaufende Threads müssen koordiniert werden, sodass man immer mehrere Programmflüsse im Auge haben muss, insbesondere wenn sie auf gemeinsame Daten zugreifen. Wenn eine Variable von einem Thread geschrieben wird, während der andere sie liest, kann das dazu führen, dass das System in einen falschen Zustand gerät. Für gemeinsam verwendete Objekte müssen gesondert Synchronisationsmechanismen eingesetzt werden, um konsistente Zustände sicherzustellen. Des Weiteren kommen auch Cache-Effekte hinzu. Laufen zwei Threads auf verschiedenen Kernen, so besitzt jeder seine eigene Sicht auf die Variablenwerte. Man muss nun dafür Sorge tragen, dass gemeinsam benutzte Daten, die aus Performance-Gründen in den Caches gehalten werden, immer synchron bleiben. Weiter ist es möglich, dass sich Threads gegenseitig in ihrem Fortkommen behindern oder sogar verklemmen.

1.2.3 Sicherer Umgang mit Nebenläufigkeit

Den verschiedenen Nachteilen versucht man durch die Einführung von Parallelisierungs- und Synchronisationskonzepten auf höherer Ebene entgegenzuwirken. Ziel ist es, dass Entwickler möglichst wenig mit *Low-Level*-Synchronisation und Thread-Koordination in Berührung kommen. Hierzu gibt es verschiedene Vorgehensweisen. So wird z. B. bei C/C++ mit OpenMP¹ die Steuerung der Parallelität deklarativ über `#pragma` im Code verankert. Der Compiler erzeugt aufgrund dieser Angaben parallel ablaufenden Code. Die Sprache Cilk erweitert C/C++ um neue Schlüsselworte, wie z. B. `cilk_for`².

Java geht hier den Weg über die Bereitstellung einer »Concurrency-Bibliothek«, die mit Java 5 eingeführt wurde und sukzessive erweitert wird. Nachdem zuerst Abstraktions- und Synchronisationskonzepte wie *Thread-pools*, *Locks*, *Semaphore* und *Barrieren* angeboten wurden, sind mit Java 7 und Java 8 auch Parallelisierungsframeworks hinzugekommen. Nicht vergessen werden darf hier auch die Einführung Thread-sicherer Datenstrukturen, die unverzichtbar bei der Implementierung von Multithreaded-Anwendungen sind. Der Umgang mit diesen *High-Level*-Abstraktionen ist bequem und einfach. Nichtsdestotrotz gibt es auch hier Fallen, die man nur dann erkennt, wenn man die zugrunde liegenden *Low-Level*-Konzepte beherrscht. Deshalb werden im ersten Teil des Buches die Basiskonzepte ausführlich erklärt, auch wenn diese im direkten Praxiseinsatz immer mehr an Bedeutung verlieren.

1.3 Maße für die Parallelisierung

Neben der Schwierigkeit, korrekte nebenläufige Programme zu entwickeln, gibt es auch inhärente Grenzen für die Beschleunigung durch Parallelisierung. Eine wichtige Maßzahl für den Performance-Gewinn ist der *Speedup* (Beschleunigung bzw. Leistungssteigerung), der wie folgt definiert ist:

$$S = \frac{T_{seq}}{T_{par}}$$

Hierbei ist T_{seq} die Laufzeit mit einem Kern und T_{par} die Laufzeit mit mehreren.

1.3.1 Die Gesetze von Amdahl und Gustafson

Eine erste Näherung für den *Speedup* liefert das Gesetz von Amdahl [2]. Hier fasst man die Programmteile zusammen, die parallel ablaufen können.

¹Siehe <http://www.openmp.org>.

²Siehe <http://www.cilkplus.org>.

Wenn P der prozentuale, parallelisierbare Anteil ist, dann entspricht $(1 - P)$ dem sequenziellen, nicht parallelisierbaren. Hat man nun N Prozessoren bzw. Rechenkerne zur Verfügung, so ergibt sich der maximale *Speedup*

$$S(N) = \frac{\text{Sequenzielle Laufzeit}}{\text{Parallele Laufzeit}} = \frac{1}{\frac{P}{N} + (1 - P)},$$

wobei hier implizit davon ausgegangen wird, dass die Parallelisierung einen konstanten, vernachlässigbaren, internen Verwaltungsaufwand verursacht. Durch Grenzwertbildung $N \rightarrow \infty$ ergibt sich dann der theoretisch maximal erreichbare *Speedup* beim Einsatz von unendlich vielen Kernen bzw. Prozessoren zu

$$\lim_{N \rightarrow \infty} S(N) = \lim_{N \rightarrow \infty} \frac{1}{\frac{P}{N} + (1 - P)} = \frac{1}{(1 - P)}.$$

An der Formel sieht man, dass der nicht parallelisierbare Anteil den *Speedup* begrenzt. Beträgt der parallelisierbare Anteil z.B. nur 50%, so kann nach dem Amdahl'schen Gesetz maximal nur eine Verdopplung der Ausführungsgeschwindigkeit erreicht werden (vgl. Abb. 1-1).

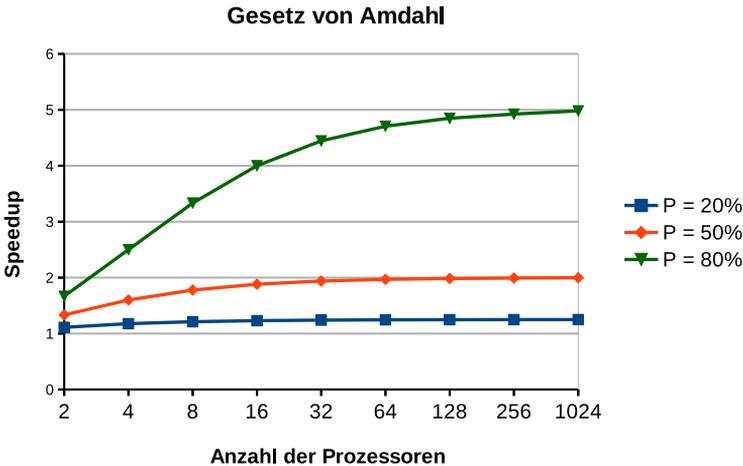


Abbildung 1-1: *Speedup* in Abhängigkeit von P und N

Man kann die Parallelisierung aber auch unter einem anderen Gesichtspunkt betrachten. Amdahl geht von einem fest vorgegebenen Programm bzw. einer fixen Problemgröße aus. Gustafson betrachtet dagegen eine variable Problemgröße in einem festen Zeitfenster [18]. Er macht die Annahme, dass sich die Vergrößerung des zu berechnenden Problems im Wesentlichen üblicherweise nur auf den parallelisierbaren Programmteil P auswirkt (man sagt, die Anwendung ist skalierbar). Unter diesem Aspekt er-

gibt sich ein *Speedup* von

$$S(N) = (1 - P) + N \cdot P$$

d.h., der Zuwachs ist hier proportional zu N .

Die unterschiedlichen Sichtweisen zwischen Amdahl und Gustafson sind in der Abbildung 1-2 verdeutlicht.

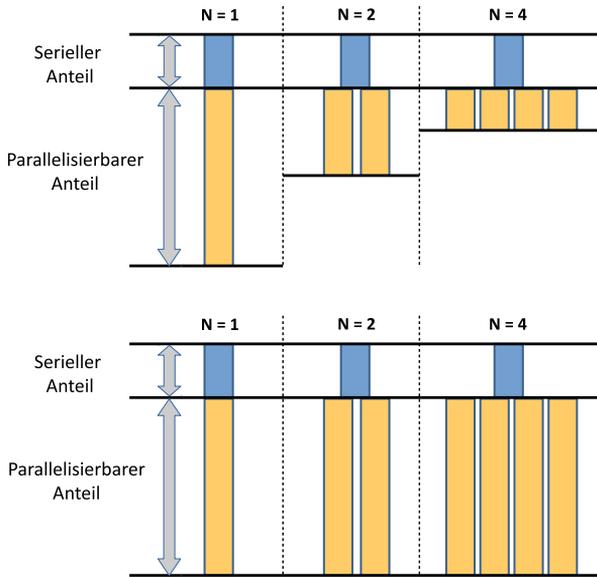


Abbildung 1-2: Amdahl (oben) versus Gustafson (unten)

1.3.2 Work-Span-Analyse

Eine weitere Methode, den Grad einer Parallelisierung zu beschreiben, ist die *Work-Span-Analyse* [10]. In dem zugrunde liegenden Modell werden die Abhängigkeiten der auszuführenden Aktivitäten in einem azyklischen Graphen dargestellt (vgl. Abb. 1-3). Eine Aktivität kann hier erst dann ausgeführt werden, wenn alle »Vorgänger« abgeschlossen sind.

Die von dem Algorithmus zu leistende Gesamtarbeit ist die Summe der auszuführenden Aktivitäten. Man bezeichnet die benötigte Zeit (*work*) hierfür mit T_1 . Der sogenannte *span*, der mit T_∞ bezeichnet wird, entspricht dem kritischen Pfad, also dem längsten Weg von Aktivitäten, die nacheinander ausgeführt werden müssen³.

³In der Literatur wird der *span* auch manchmal als *step complexity* oder *depth* bezeichnet.

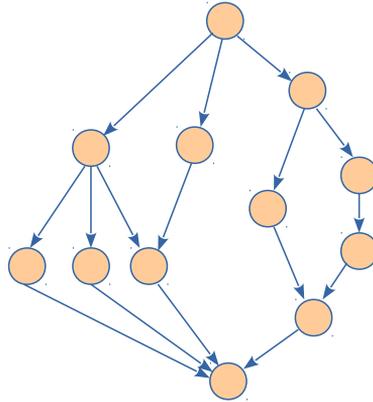


Abbildung 1-3: Azyklischer Aktivitätsgraph

Wenn wir uns den Aktivitätsgraphen in Abbildung 1-3 anschauen und annehmen, dass jede Aktivität eine Zeiteinheit dauert, so erhalten wir für den *work* $T_1 = 12$ und den *span* $T_\infty = 6$. Sei N wieder die Anzahl der Rechenkerne bzw. Prozessoren, dann erhält man als Speedup:

$$S(N) = \frac{T_1}{T_N} \leq N.$$

Der Speedup wächst linear mit der Anzahl der Prozessoren, vorausgesetzt dass die CPU immer voll ausgelastet ist (*greedy scheduling*). Der Speedup ist allerdings durch den *span* begrenzt, da der kritische Pfad sequenziell abgearbeitet werden muss:

$$S(N) = \frac{T_1}{T_N} \leq \frac{T_1}{T_\infty} = \frac{\text{work}}{\text{span}}.$$

In unserem Beispiel beträgt der maximal erreichbare Speedup $T_1/T_\infty = 2$.

1.4 Parallelitätsmodelle

In der Literatur wird zwischen verschiedenen Modellen für die Parallelisierung unterschieden. Java unterstützt jedes dieser Modelle durch das Bereitstellen verschiedener Konzepte und APIs.

Zur Parallelisierung von Anwendungen gibt es grundsätzlich zwei Ansätze: Daten- und Task-Parallelität⁴. Bei der *Datenparallelität* wird ein Datenbestand geteilt und die Bearbeitung der Teilbereiche verschiedenen Threads zugeordnet. Hierbei führt jeder Thread dieselben Operationen aus.

⁴Die beiden Parallelisierungskonzepte werden ausführlich in [15] diskutiert.

Diese Art der Parallelisierung wird durch das Gesetz von Gustafson beschrieben und ist in der Regel gut skalierbar [53]. Mit dem ForkJoin-Framework und dem `Stream-API` stehen bei Java hierfür zwei leistungsfähige Möglichkeiten zur Verfügung (siehe Kapitel 13 und 14). Falls man diese Frameworks nicht einsetzen möchte, kann für eine explizite Umsetzung auf zahlreiche Synchronisationskonzepte zurückgegriffen werden (siehe Kapitel 11 und 12).

Bei der *Task-Parallelität*⁵ wird die Anwendung in Funktionseinheiten zerlegt, die dann bezüglich ihrer Abhängigkeiten ausgeführt werden. Diese Art der Parallelisierung wird durch die *Work-Span-Analyse* beschrieben und kann bei Java mithilfe der `CompletableFuture`-Klasse oder je nachdem auch mit dem ForkJoin-Framework realisiert werden (siehe Kapitel 13 und 15).

Neben diesen beiden grundsätzlichen Ansätzen wird auch oft noch zwischen dem *Master-Slave*-, dem *Work-Pool*- und dem *Erzeuger-Verbraucher*- bzw. *Pipeline*-Programmiermuster unterschieden [32]. Das Unterscheidungsmerkmal ist hierbei die Art und Weise, wie die beteiligten Komponenten miteinander kommunizieren. Beim *Master-Slave*-Modell gibt es einen dedizierten Thread, der Aufgaben an andere verteilt und dann die Ergebnisse einsammelt. Bei Java kann dieses Modell mit dem `Future`-Konzept umgesetzt werden (siehe Abschnitt 6.2). Das *Work-Pool*-Modell entspricht dem `ExecutorService`, dem man Aufgaben zur Ausführung delegieren kann (siehe Abschnitt 6.1). Das bewährte *Erzeuger-Verbraucher*-Modell wird typischerweise durch `BlockingQueue`-Datenstrukturen realisiert und existiert in verschiedenen Varianten (siehe Abschnitt 10.3). In der Praxis findet man häufig Kombinationen der verschiedenen Modelle bzw. Muster.

⁵Genauer müsste man eigentlich »funktionale Dekomposition« (*functional decomposition*) sagen, da der Begriff Task-Parallelität oft auf alles Mögliche angewendet wird.

Teil I

Grundlegende Konzepte

2 Das Thread-Konzept von Java

Die Unterstützung der Thread-Programmierung ist ein zentraler Bestandteil der Java-Sprachdefinition. Man erkennt dies sowohl an der Klasse `Thread`, die im Paket `java.lang` zu finden ist, als auch an Schlüsselwörtern, wie z.B. `synchronized` und `volatile`. Durch diese wichtige Sprachverankerung können portable Multithreaded-Anwendungen implementiert werden¹.

Da es mit Java sehr einfach ist, Threads zu erzeugen und zu starten, werden sie auch gerne eingesetzt und mitunter ohne wirklichen Nutzen. Insbesondere möchte man ja die Ressourcen eines Multicore-Rechners ausschöpfen. Dabei machen sich viele Entwickler wenig Gedanken darüber, dass man mit dem Einsatz von Threads den Programmfluss aufspaltet, asynchrone Programmfäden (Nebenflüsse) startet und damit unter Umständen parallel auf gemeinsam genutzte Daten zugreift.

In diesem Kapitel stellen wir das grundlegende Thread-API von Java vor. Es sind nur wenige Konstrukte und Klassen, die speziell für die Unterstützung der nebenläufigen Programmierung entworfen wurden. Dabei spielt die Klasse `java.lang.Thread` eine zentrale Rolle.

2.1 Der main-Thread

Eine Java-Anwendung wird in einer *Java Virtual Machine* (JVM) ausgeführt. Die JVM selbst entspricht hierbei einem Prozess des Betriebssystems. Zur Ausführung des Programms startet die JVM unter anderem den sogenannten `main`-Thread², der die Befehlszeilen Schritt für Schritt abarbeitet.

¹In anderen Programmiersprachen wie C/C++ war die Thread-Unterstützung lange compiler- und plattformabhängig. Dadurch war die Entwicklung portierbarer Multithreaded-Anwendungen mit C/C++ alles andere als einfach. Erst mit dem C++11-Standard wurde eine portable Bibliothek definiert.

²Die JVM startet auch noch weitere Threads. So gibt es z.B. einen für das Garbage-Collecting und einen, der für die Aufräumarbeit am Ende des Lebenszyklus eines Objekts zuständig ist.

Codebeispiel 2.1 zeigt ein einfaches Programm, das neben der Anzahl der zur Verfügung stehenden Rechenkerne (Hardware-Threads) einige Eigenschaften des `main`-Threads ausgibt. Dabei werden Kerne mit Hyperthread-Unterstützung doppelt gezählt, da diese zwei Hardware-Threads bereitstellen.

```
public class MainThreadEigenschaft
{
    public static void main(String[] args)
    {
        // Anzahl der Prozessoren abfragen
        int nr = Runtime.getRuntime().availableProcessors();
        System.out.println("Anzahl der Prozessoren " + nr);

        // Eigenschaften des main-Threads
        Thread self = Thread.currentThread();
        System.out.println("Name      : " + self.getName());
        System.out.println("Priorität : " + self.getPriority());
        System.out.println("ID       : " + self.getId());
    }
}
```

Codebeispiel 2.1: Ausgabe verschiedener Attribute des `main`-Threads

Zugriff auf den ausführenden Thread erhält man über die Klassenmethode `Thread.currentThread`. Im Codebeispiel 2.1 werden der Name, die Priorität und die Kennung des Threads, die ihm von der JVM zugewiesen wurde, auf die Konsole ausgegeben.

Der von der JVM erzeugte Thread, ein sogenannter Java-Thread, ist lediglich ein Abstraktionskonzept. Falls das zugrunde liegende Betriebssystem selbst Threads unterstützt (Betriebssystem- bzw. OS-Threads), kann die JVM die Java-Threads auf sie abbilden. Die Zuordnung der OS- auf die Hardware-Threads übernimmt der Scheduler des Betriebssystems (vgl. Abb. 2-1).

Da moderne Betriebssysteme Threads unterstützen und zeitgemäße Hardware auch mehrere Rechenkerne besitzen, werden wir im Folgenden oft implizit davon ausgehen, dass ein Java-Thread einem Hardware-Thread zugeordnet ist. In vielen Fällen sprechen wir, falls die Unterscheidung unwesentlich ist, deshalb nur noch von Threads, meinen aber streng genommen immer Java-Threads.

2.2 Erzeugung und Starten von Threads

Innerhalb eines Java-Programms können mithilfe der Klasse `Thread` zusätzliche Java-Threads gestartet werden. Der von dem erzeugten Thread

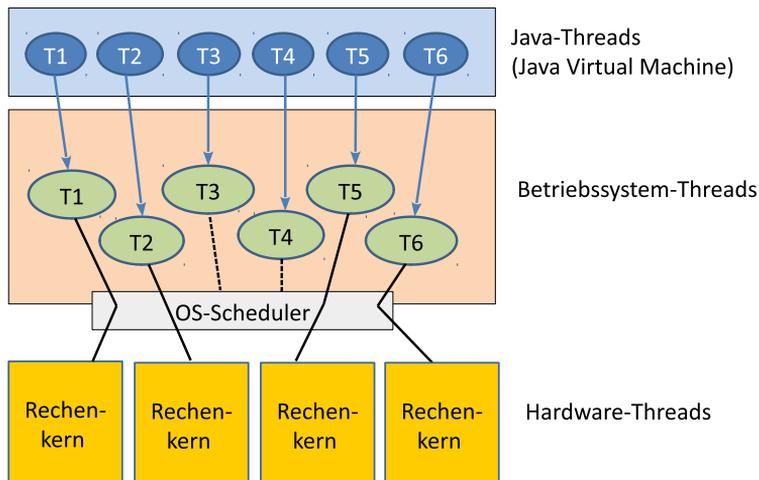


Abbildung 2-1: Zuordnung der Java-Threads zu einzelnen Kernen

auszuführende Code kann hierbei auf zwei Arten zur Verfügung gestellt werden:

1. Man leitet direkt von der Klasse `Thread` ab und überschreibt die `run`-Methode.
2. Man stellt eine Klasse bereit, die das `Runnable`-Interface implementiert. Ein Objekt dieser Klasse wird auch oft als *Task* bezeichnet. Es wird dann einem `Thread` zur Ausführung übergeben.

In der Praxis sollte man die zweite Möglichkeit bevorzugen, da hier konzeptuell klar zwischen dem Programmfluss (Thread) und der nebenläufig durchzuführenden Aufgabe (Task) unterschieden wird.

2.2.1 Thread-Erzeugung durch Vererbung

Eine einfache Art, einen nebenläufigen Programmfluss zu definieren, ist die Implementierung einer Unterklasse von `Thread`, bei der die `run`-Methode mit dem auszuführenden Code überschrieben wird. Das eigentliche Starten des Threads erfolgt durch den Aufruf der `start`-Methode.

Abbildung 2-2 zeigt den schematischen Ablauf im Sequenzdiagramm. Nachdem ein `MyThread`-Objekt erzeugt wurde, wird `start` aufgerufen. Dadurch wird der JVM mitgeteilt, dass vom Betriebssystem ein OS-Thread angefordert wird, der den in der `run`-Methode hinterlegten Code abarbeitet. Auf den exakten Startpunkt der Ausführung von `run` hat man keinen Einfluss. Sobald die `run`-Methode ausgeführt wird und der `main`-Thread noch aktiv ist, laufen in der Anwendung zwei nebenläufige Programmfäden

(Programmflüsse) ab. Wenn der Thread mit der `run`-Methode fertig ist, terminiert er. Ein häufig gemachter Anfängerfehler ist der direkte Aufruf von `run`. In dem Fall wird sie nicht parallel in einem neuen Thread, sondern in dem des Aufrufers ausgeführt.

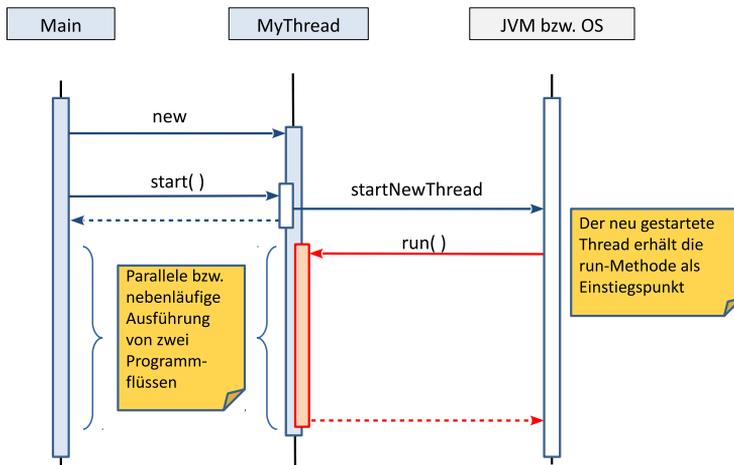


Abbildung 2-2: Sequenzdiagramm für das Starten eines neuen Threads

Codebeispiel 2.2 zeigt ein Programm, in dem drei Threads erzeugt und gestartet werden. Danach gibt jeder zwei Meldungen auf die Konsole aus.

```

class MyWorker extends Thread ❶
{
    public MyWorker(String name) ❷
    {
        super(name);
    }

    @Override
    public void run() ❸
    {
        Thread self = Thread.currentThread(); ❹
        System.out.println("Hallo Welt von " + self.getName());
        System.out.println("Die ID von " + self.getName()
            + " ist " + self.getId());
    }
}

public class ThreadDurchVererbung
{
    public static void main(String[] args)
    {
        for (int i = 0; i < 3; i++)
        {
            MyWorker t = new MyWorker("Worker " + i); ❺
        }
    }
}
  
```

```
t.start();
    }
}
```

Codebeispiel 2.2: Erzeugung von Threads durch Vererbung

Die Klasse `MyWorker` ist hier von `Thread` abgeleitet (❶). Die auszuführenden Aktionen werden in der überschriebenen `run`-Methode implementiert (❷). Über den Konstruktor wird dem Thread ein Name zugewiesen (❸). Erst durch den Aufruf von `start` wird `run` von einem neu gestarteten Thread ausgeführt (❹). Statt `Thread.currentThread` kann auch direkt `this` verwendet werden, da ein `MyWorker`-Objekt einem Java-Thread entspricht (❺).

In dem Beispiel greifen bereits alle drei Threads konkurrierend auf eine Ressource zu, nämlich auf den `OutputStream` von `System.out`. Die `println`-Methode von `System.out` besitzt einen Serialisierungsmechanismus (Lock), sodass immer nur ein Thread sie ausführen kann. Damit ist gewährleistet, dass sich die Ausgaben nicht gegenseitig überschreiben. Sie können aber durchaus vermischt werden, z. B.:

```
Hallo Welt von Worker 0
Hallo Welt von Worker 1
Die ID von Worker 0 ist 1
Hallo Welt von Worker 2
Die ID von Worker 2 ist 3
Die ID von Worker 1 ist 2
```

Hinweis

Nie den Thread aus seinem Konstruktor heraus starten!

Falls man die hier erläuterte Erzeugung von Threads benutzt, sollte man nie die `start`-Methode direkt im Konstruktor aufrufen. Es könnte nämlich passieren, dass der zugehörige Thread sofort gestartet wird, noch bevor der Konstruktor abgearbeitet ist. Die in dem Zusammenhang aufgerufene Methode `run` greift dann unter Umständen auf Variablen zu, die möglicherweise noch gar nicht vollständig initialisiert wurden. Insbesondere betrifft dies dann abgeleitete Klassen, bei denen erst immer die Konstruktoren der Oberklassen abgearbeitet werden.

Darüber hinaus wird hierdurch auch das *Liskov'sche Substitutionsprinzip* verletzt, das besagt: Ein Objekt einer Unterklasse sollte immer so behandelt werden können, wie es die Oberklasse vorsieht. Ein Objekt der abgeleiteten Klasse ist in diesem Fall auch ein `Thread`-Objekt, das norma-

lerweise erzeugt und dann gestartet wird. Würde man z. B. mit der Unterklasse wie mit einem gewöhnlichen `Thread`-Objekt umgehen, würde das erneute Starten eine Ausnahme auslösen.

2.2.2 Thread-Erzeugung mit `Runnable`-Objekten

Die im vorherigen Abschnitt erläuterte `Thread`-Erzeugung durch Ableitung hat verschiedene Nachteile. Zum einen unterstützt Java keine Mehrfachvererbung und zum anderen entspricht ein Objekt einer abgeleiteten Klasse noch keinem Programmfluss, sodass der Typname etwas irreführend ist. Erst durch den Aufruf von `start` wird ein `Thread` und somit der Programmfluss gestartet.

In der Praxis sollte deshalb allein schon aus Entwurfsgründen der nebenläufig auszuführende Code von dem Träger des Programmflusses getrennt werden. Java stellt hierfür das funktionale Interface `java.lang.Runnable` zur Verfügung.

```
@FunctionalInterface
public interface Runnable
{
    public abstract void run();
}
```

Codebeispiel 2.3: Das funktionale Interface `Runnable`

Man benötigt also ein Objekt einer Klasse mit dem `Runnable`-Interface oder einen entsprechenden Lambda-Ausdruck. In der `run`-Methode werden die Anweisungen implementiert, die von einem `Thread` abgearbeitet werden sollen.

Instanzen der Klasse können dann `Thread`-Objekten zugewiesen werden. Hierzu stehen folgende Konstruktoren zur Verfügung:

- `public Thread(Runnable target)`
- `public Thread(Runnable target, String name)`

Über die zweite Möglichkeit kann dem ausführenden `Thread` explizit ein Name zugeordnet werden.

Abbildung 2-3 zeigt schematisch den Ablauf. Nachdem das `Runnable`-Objekt erzeugt ist, wird dessen Referenz an ein neu angelegtes `Thread`-Objekt übergeben. Das Starten des eigentlichen Threads erfolgt über die `start`-Methode. Die `run`-Methode des `Thread`-Objekts delegiert den Kontrollfluss an die des `Runnable`-Objekts.