FHZ → FACHHOCHSCHULE ZENTRALSCHWEIZ

HTA → HOCHSCHULE FÜR TECHNIK+ARCHITEKTUR LUZERN

Abteilung Informatik

Java

Thread Skalierung

Basisanalyse

Grundlagen der Skalierung

HTA Horw

HTA LUZERN
Technikumstrasse 21
CH—6048 Horw

T: 041-349-33-11
F: 041-349-39-60
W: www.hta.fhz.ch

Diplomarbeit 2006
Aregger Marcel
Meier Rainer

Änderungskontrolle

Version	Datum	Ausführende Stelle	Bemerkungen/Art der Änderung
1.1	2006-10-16	Rainer Meier	Initial Release
1.2	2006-10-27	Rainer Meier	Unzählige Änderungen (im CVS Log Dokumentiert)
1.3	2006-10-28	Rainer Meier	Formatierungen, Querverweise
		Marcel Aregger	Management Summary, Zusammenfassungen
1.4	2006-11-16	Rainer Meier	Code-Listings; Beschriftung + Verzeichnis, Kommentare entfernt, SMP Zeichnungen aktualisiert, Glossar aktualisiert.

Prüfung und Freigabe

Vorname/Name	Dokumentversion	Status	Datum	Visum
Rainer Meier	1.4	Final	2006-11-16	
Marcel Aregger	1.4	Final	2006-11-16	

1. Management Summary

Die vorliegende Basisanalyse legt den Grundstein der Diplomarbeit „Java-Thread-Skalierung“. Sie repräsentiert das notwendige Basiswissen aus dem Themenbereich „Skalierung von multithreaded Applikationen“ und vermittelt dem Leser spezifische Aspekte, welche diese beeinflussen.

Gestützt auf eine layerorientierte Vorgehensweise ist dieses Dokument gegliedert in die Themenbereiche Hardware, Betriebssystem, Applikation und Java-Virtual-Machine. Für jeden Bereich wurden Hintergrundinformationen rechechiert und in komprimierter Form dokumentiert. Faktoren die einen direkten oder indirekten Einfluss auf den weiteren Verlauf dieser Diplomarbeit haben, sind im Abschluss jedes Themenbereichs aufgeführt. Sie werden entsprechend ihrer Bedeutung in die nachfolgenden Phasen „Evaluation“ sowie „Umsetzung/Implementation“ eingearbeitet.

Die Skalierung eines Systems beschreibt die Eigenschaft, durch verändern von Systemressourcen wie bspw. CPU die Leistung gezielt zu verändern. Hierbei kann vertikale von horizontaler Skalierung unterschieden werden. Die Basisanalyse beschränkt sich auf die Untersuchung der vertikalen Variante, bei der Einflussgrössen in Bezug auf Single-Nodes (bspw. Server- oder Desktop-Systeme) beschrieben werden.

Hardware

Die Analyse im Bereich HW hat gezeigt, dass die parallele Verarbeitung von Programmen auf diesem Layer aufwändig und komplex ist. Es ist für Entwickler schwierig oder kaum nachvollziehbar wie „ihre“ Instruktionen auf der HW-Ebene ausgeführt werden. Die Einflussnahme auf die Verarbeitung beschränkt sich auf HW-nahe Sprachen wie C oder C++ und ist für Java-Programmierer sehr klein.

Chiparchitekturen sind komplexe Gebilde, die mehrere Aspekte/Technologien in sich vereinen. Die Menge und Anordnung von Cores, die Art wie die Speicheranbindung erfolgt oder Instruktionen abgearbeitet werden, lassen viele variable Chipdesigns zu. Die Analyse zeigt die Anwendung relevanter Technologien anhand konkreter Chipdesigns wie Intel Pentium4, Intel Core/Core2, AMD Opteron oder SUN UltraSparc T1. Die Fokussierung auf eine spezifische Eigenheiten einer Architektur wie beispielsweise die Anzahl FPU Einheiten oder die Anzahl Pipelines macht meist wenig Sinn da sie aufwendig ist, meist kleine Verbesserungen mit sich bringt und letztendlich auch die Portierbarkeit einer Anwendung negativ beeinflusst.

Obwohl Java die hardwarenahe Programmierung nicht zulässt bietet sie doch den Vorteil, dass der Bytecode zur Laufzeit mittels Just-in-Time (JIT) Compiler in Maschinencode umgewandelt wird. Somit ist es möglich ein plattformunabhängiges Programm zur Laufzeit auf Hardware-Spezifische Eigenheiten hin zu optimieren. Folgende Aspekte und Technologien im Bereich Hardware/Chipdesign haben direkten oder indirekten Einfluss auf die Umsetzungsphase und werden entsprechend berücksichtigt:

Einfluss	Aspekt/Technologie
Direkt	SMP, CMP, CMT
Indirekt	UMA/NUMA, Skalar/Superskalar, Pipeline

Betriebssystem

Die Anwendung von Threads auf Level Betriebssystem bringt den Vorteil der einfachen Erzeugung, Interprozesskommunikation oder schnellen Kontextwechsel dieser leichtgewichtigen Prozesse. Dabei sind jedoch zunehmende Verwaltungs- oder Synchronisationsaufwendungen zu berücksichtigen. Windows XP implementiert das Thread-Model bereits auf Kernel-Ebene. Bei einer 1:1-Zuordnung der Anwendungs-Threads laufen im gleichen Prozesskontext mehrere Kernel-Level Threads die vom Kernel direkt verwaltet werden. Sie können so auf mehrere Kerne verteilt werden. Die Java-Virtual-Machine (JVM) bildet als Layer zwischen Betriebssystem und Applikation die Laufzeitumgebung eines Java-Threads und bestimmt, ob diese direkt (Native Threads) oder indirekt (Green Threads) auf Threads des Betriebssystems abgebildet werden.

Unter Windows 2000/XP mit einem „priority-driven – preemtive scheduling”-System wird die Zuordnung der Rechenzeit auf Threads über die Basispriorität dieser Threads gesteuert. Sie kann direkt über die Win32-API beeinflusst werden. Die indirekte Beeinflussung ist über die Priorisierung der Ja-

va-Threads möglich unter Verwendung von Native-Threads (durch die JVM). Wie Prioritäten der Java-Ebene auf Prioritäten der Betriebssystemebene abgebildet werden, soll mit dieser Arbeit ebenfalls gezeigt werden.

Eine weitere Möglichkeit das Laufzeitverhalten in Bezug auf Threads zu beeinflussen, ist mit der Affinität gegeben. Sie ermöglicht die explizite Zuordnung von Prozessoren auf Level Prozess oder Threads und kann ebenfalls über die Win32-API gesteuert werden. Die direkte Zuordnung über die Java-API ist nicht möglich. Alternativ können aber über Systemtools entsprechende Eingriffe vorgenommen werden. Folgende Aspekte und Technologien im Bereich Betriebssystem werden weiter verfolgt:

Einfluss	Aspekt/Technologie
Direkt	Designprinzip Threads, WIN-32 Thread
Indirekt	Scheduling, Affinität

Applikation

Der Bereich Applikation beleuchtet Techniken und Probleme der parallelen Programmierung. Dabei sind Standards wie POSIX Threads, OpenMP, Thread Buliding Blocks (TBB) oder Message Passing Interfaces (MPI) Technologien, die explizit für eine parallele Programmierung entwickelt wurden oder diese unterstützen. Für die weitere Betrachtung sind Posix Threads sowie OpenMP von Bedeutung. Letzteres wurde für Java im Projekt JOMP umgesetzt und steht für diese Arbeit zu Verfügung. TBB besteht aus einer reinen C/C++ Bibliothek und MPI ist hauptsächlich für eine horizontale Skalierung interessant.

Einfluss	Aspekt/Technologie
Direkt	POSIX Threads, OpenMP
Indirekt	TBB, MPI

Java Virtual Machine

Der Bereich JVM bietet einen Überblick über die Java-API, die mit Threads ein funktionales und einfaches Instrument der parallelen Programmierung bietet. Da dem Java-Entwickler nur diese API zu Verfügung steht, ist der direkte Zugriff auf Betriebssystem-Funktionalität oder gar die Hardware nicht möglich. Konkret liegt es an der JVM die Java-Anwendungen möglichst effizient auf der vorhandenen Hardware ablaufen zu lassen.

Trotz diesem eingeschränkten Handlungsspielraum für einen Java-Entwickler können im Sinne einer optimalen bzw. effizienten Programmierung spezifische Techniken und Packages angewendet um ideale Voraussetzungen für die Verteilung von Threads zu schaffen. Das Kapitel beleuchtet in diesem Zusammenhang beispielsweise das Package `java.util.concurrent` in Java 5, die `ReentrantLock`-Klasse oder wichtige Methoden der `AtomicInteger` Klasse aus dem Package `java.util.concurrent.atomic`.

Da eine multithreaded Applikationen unweigerlich mit dem Thema Synchronisation verbunden ist, wird sie in diesem Abschnitt ausführlich hinterfragt. Neben der bekannten Block- oder Methoden-Synchronisation sind auch Themen wie „Unterbrechbare Locks“, „Granularität von Locks“ oder gar „Lockfreie Implementierungen (Compare and Swap; CAS)“ Gegenstand dieser Analyse.

Die Performance einer Java-Applikation hängt nicht nur von der effizienten Programmierung der Java-Routinen ab sondern auch vom effizienten Zusammenspiel von Java-Applikation, Java Virtual Machine (JVM), Betriebssystem und Hardware. Für spezifische Bereiche wie JIT-Compiler, Thread-Modell oder Garbage Collection werden einige Optimierungsmöglichkeiten aufgezeigt.

Einfluss	Aspekt/Technologie
Direkt	Java Threading, JOMP
Indirekt	JVM Optimierung

2. Inhaltsverzeichnis

3. Dokumentinformationen

3.1. Referenzierte Dokumente

Tabelle 1 Referenzierte Dokumente

Referenz	Beschreibung
[1]	Diehl Roger, Parallele und verteilte Systeme, Sechste Auflage, 2005
[2]	Oliver Lau, c't Ausgabe 15/2006, Seite 218ff, OpenMP
[3]	Oliver Lau, c't Ausgabe 21/2006, Seite 234ff, Thread-Baukasten/TBB
[4]	Brian Goetz, Java Concurrency in Practice, ISBN 0-321-34960-1
[5]	Diplomarbeit 2006, Aufgabenstellung V1.0 vom 12. Oktober 2006

3.2. Definitionen und Abkürzungen

Tabelle 2 Abkürzungen

Abkürzung	Beschreibung
AMD	Advanced Micro Devices
API	Application Programming Interface
ASMP	Asymmetric Multi Processing
CAS	Compare And Swap / Compare And Set
CISC	Complex Instruction Set Computing
CMP	Chip Multi Processing
CMT	Chip Multi Threading
CPU	Central Processing Unit
DDR-RAM	Double Data Rate Random Access Memory
EIST	Enhanced Intel Speed Step Technology
GC	Garbage Collection
GPL	Gnu Public License
IPC	Inter Process Communication
JIT	Just In Time Compiler
JOMP	Java OpenMP
JVM	Java Virtual Machine
MMX	Multimedia Extension
MPI	Message Passing Interface

NUMA	Non-Uniform Memory Architecture
NUMA	Non-Uniform Memory Access
POSIX	Portable Operating System Interface
POSIX	Portable Operating System Interface for UniX
RD-RAM	Rambus Dynamic Random Access Memory
RISC	Reduced Instruction Set Computing
SD-RAM	Synchronous Dynamic Random Access Memory
SIMD	Single Instruction Multiple Data
SMP	Symmetric Multi Processing
SPARC	Scalable Processor ARChitecture
SSE	Streaming SIMD extension
TBB	Thread Building Blocks
TDP	Termal Design Power
UMA	Uniform Memory Architecture
UMA	Uniform Memory Access

3.3. Links

Tabelle 3 Links

Referenz	Beschreibung
[AMD64]	Wikipedia, AMD64: http://de.wikipedia.org/wiki/AMD64
[CACHECOH]	Wikipedia, Cache coherency: http://en.wikipedia.org/wiki/Cache_coherence
[CISC]	Wikipedia, CISC: http://en.wikipedia.org/wiki/CISC
[CMP]	Wikipedia, Chip-level multiprocessing: http://en.wikipedia.org/wiki/Chip-level_multiprocessing
[CONTEXTSW]	Wikipedia, Contect Switch: http://en.wikipedia.org/wiki/Context_switch
[CORE2]	Wikipedia, Core 2: http://de.wikipedia.org/wiki/Core_2
[COREARCH]	Intel, Core Microarchitecture: http://www.intel.com/technology/architecture/coremicro/demo/demo.htm
[CPUAFFINITY]	TMurgent Technologies, White Paper Processor Affinity: http://www.tmurgent.com/WhitePapers/ProcessorAffinity.pdf
[DEADLOCK]	Wikipedia, Deadlock: http://en.wikipedia.org/wiki/Deadlock
[DEVXINTEL]	Devx, Intel Threading Tools and OpenMP: http://www.devx.com/go-parallel/Article/32724
[GCC]	GNU, GCC, http://gcc.gnu.org/
[HOTSPOT]	Sun, HotSpot Virtual Machine: http://java.sun.com/javase/technologies/hotspot/
[HOTSPOTGC]	Sun, HotSpot Garbage Colleciton Tuning with the 5.0 Java Virtual Machine: http://java.sun.com/docs/hotspot/gc5.0/gc_tuning_5.html
[HOTSPOTOPT]	Sun, HotSpot VM Options: http://java.sun.com/docs/hotspot/VMOptions.html
[HOTSPOTTHR]	Sun, HotSpot Threading: http://java.sun.com/docs/hotspot/threads/threads.html
[HTT]	Wikipedia, Hyper-Threading: http://de.wikipedia.org/wiki/Hyper-Threading
[HYPERTRANS]	Wikipedia, HyperTransport: http://en.wikipedia.org/wiki/HyperTransport
[INTELC]	Intel, Compilers : http://www.intel.com/cd/software/products/asmo-na/eng/compilers/
[INTELTBB]	Intel, Thread Building Blocks 1.0 for Windows, Linux and Mac OS : http://www.intel.com/cd/software/products/asmo-na/eng/294797.htm
[JAPIREF]	Sun, Java API Reference: http://java.sun.com/reference/api/
[JAVANUMA]	Mustafa M. Tikir, NUMA-Aware Java Heaps for Server Applications: http://www.cs.umd.edu/~hollings/papers/ipdps05.pdf
[JLS]	Sun, Java Language Specification: http://java.sun.com/docs/books/jls/

[JOMP] EPCC, OpenMP-like directives for Java: http://www.epcc.ed.ac.uk/research/jomp/

[JVMS] Sun, Java Virtual Machine Specification: http://java.sun.com/docs/books/vmspec/

[LIVELOCK] Wikipedia, Livelock: http://en.wikipedia.org/wiki/Deadlock#Livelock

[MMX] Wikipedia, MMX: http://en.wikipedia.org/wiki/MMX

[MOORE] Wikipedia, Mooresches Gesetz: http://de.wikipedia.org/wiki/Mooresches_Gesetz

[MPI] Wikipedia, Message Passing Interface:
http://en.wikipedia.org/wiki/Message_Passing_Interface

[MPI-TRIER] Alexander Greiml, Universität Trier, Message Passing Interface (MPI):
http://www.syssoft.uni-trier.de/systemsoftware/Download/Seminare/Middleware/middleware.3.book.html

[MSDNSCHED] MSDN, Scheduling Priorities:
http://windowssdk.msdn.microsoft.com/en-us/library/ms685100.aspx

[MSNUMA] Microsoft, NUMA Support für Windows Server 2003:
http://www.microsoft.com/windowsserver2003/evaluation/features/comparefeatures.mspx

[MULTIPROC] Wikipedia, Multiprocessing: http://en.wikipedia.org/wiki/Multiprocessing

[MUTEX] Wikipedia, Mutex: http://de.wikipedia.org/wiki/Mutex

[NETBURST] Wikipedia, Netburst: http://en.wikipedia.org/wiki/NetBurst

[NUMA] Wikipedia, Non-Uniform Memory Access:
http://en.wikipedia.org/wiki/Non-uniform_memory_access

[OPENMP] OpenMP, Homepage: http://www.openmp.org/

[OPENMPSUN] Sun, OpenMP Support in Sun Studio Compilers and Tools:
http://developers.sun.com/sunstudio/articles/studio_openmp.html

[OPENMPWP] Wikipedia, OpenMP: http://de.wikipedia.org/wiki/OpenMP

[OPENSPARC] OpenSparc, offene Hardware-Entwicklung auf der Basis der unter GPL freigegebenen UltraSparc T1 Spezifikation: http://www.opensparc.net/

[OPTERON] Wikipedia, AMD Opteron: http://de.wikipedia.org/wiki/AMD_Opteron

[P4DTDP] Intel, Pentium D Processor Thermal Specifications (DualCore):
http://www.intel.com/cd/channel/reseller/asmo-na/eng/products/desktop/processor/processors/pentium-d/tech/216412.htm

[P4TDP] Intel, Pentium 4 Processor Thermal Specifications:
http://www.intel.com/cd/channel/reseller/asmo-na/eng/products/desktop/processor/processors/proc_dsk_p4_ee/tech/99346.htm

[PARALLELISM] Wikipedia, Parallel computing: http://en.wikipedia.org/wiki/Parallel_computing

[PENTIUM4] Wikipedia, Pentium 4: http://de.wikipedia.org/wiki/Pentium_4

[PERFTOOLS] ZDNet, System-Performance im Visier: Die besten Tools:
http://www.zdnet.de/downloads/weekly/14/weekly_280-wc.html

[PIPELINE] Wikipedia, Pipeline: http://en.wikipedia.org/wiki/Pipeline_(computer)

[POSIXTUTOR] Mark Hays, POSIX hreads Tutorial:
http://math.arizona.edu/~swig/documentation/pthreads/

[PROCEXP] Sysinternals Process Explorer Overv;
http://www.microsoft.com/technet/sysinternals/utilities/ProcessExplorer.mspx

[PTEXPL] IBM, POSIX threads explained:
http://www-128.ibm.com/developerworks/linux/library/l-posix1.html

[RISC] Wikipedia, RISC: http://en.wikipedia.org/wiki/RISC

[SCALABILITY] Wikipedia, Scalability: http://en.wikipedia.org/wiki/Scalability

[SCALAR] Wikipedia, Scalar processor: http://en.wikipedia.org/wiki/Scalar_processor

[SIMD] Wikipedia, SIMD: http://en.wikipedia.org/wiki/SIMD

[SMT] Wikipedia, Simultanous Multi Threading:
http://en.wikipedia.org/wiki/Simultaneous_multithreading

[SSE] Wikipedia, Streaming SIMD Extensions:
http://en.wikipedia.org/wiki/Streaming_SIMD_Extensions

[STARVATION] Wikipedia, Resource starvation: http://en.wikipedia.org/wiki/Resource_starvation

[STHREAD] Wikipedia, Super-Threading: http://en.wikipedia.org/wiki/Super-threading

[SUNSPARC] Wikipedia, Sun SPARC: http://de.wikipedia.org/wiki/Sun_SPARC

[SUPSCALAR] Wikipedia, Superscalar: http://en.wikipedia.org/wiki/Superscalar

[TASKMANOV] Microsoft, Task Manager Overview:
http://www.microsoft.com/resources/documentation/windows/xp/all/proddocs/en-us/taskman_whats_there_w.mspx?mfr=true

[THREAD] Wikipedia, Thread: http://de.wikipedia.org/wiki/Thread_(Informatik)

[UMA] Wikipedia, Uniform Memory Access:
http://en.wikipedia.org/wiki/Uniform_Memory_Access

[WIN2KDEV] Windows 2000 developers's guide (ISBN 3-8272-5702-6):
http://www.mut.de/media_remote/katalog/bsp/3827257026bsp.pdf

4. Einleitung

4.1. Der Begriff der Skalierung

Mit der Skalierbarkeit eines Systems meint man allgemein die Fähigkeit die Leistung durch hinzufügen bzw. entfernen von Ressourcen zu verändern. Im Optimalfall ist die Skalierung linear. Dies würde bedeuten, dass eine Anwendung in einem System mit doppelten Ressourcen doppelt so schnell arbeiten kann.

Skalierbarkeit sowohl auf Hardware- wie auch auf Software-Ebene ein gewünschtes Attribut. Insbesondere sind bei der Entwicklung einer Applikation selten exakte Daten über die spätere Lastsituation vorhanden. Reicht die Leistung der Applikation nicht aus kann dies durch die Erweiterung des Systems (z.B. Hardware-Ausbau oder hinzufügen weiterer Cluster-Nodes) geschehen. Ist die Anwendung aber nicht skalierbar, so kann dadurch kein oder nur ein geringer Leistungszuwachs (im Extremfall sogar eine Leistungsverminderung) eintreten.

Dabei ist die Skalierung grundsätzlich in zwei Kategorien unterteilbar:

- Vertikale Skalierung
- Horizontale Skalierung

Unter dem Begriff der vertikalen Skalierung versteht man die Ressourcenerweiterung eines einzelnen Knotens (engl. Node) um dessen Leistung zu erhöhen. Diese Form der Skalierung ist essentiell wichtig für Applikationen, die auf einem einzigen Knoten ausgeführt werden. Beispielsweise für Single-Node Server oder Desktop Anwendungen.

Unter dem Betriff der horizontalen Skalierung versteht man in der Software-Technik die Möglichkeit weitere Knoten (engl. Nodes) zum System hinzuzufügen um die Leistung zu erhöhen. Hierbei spricht man auch von der Skalierung von verteilten Systemen wie High-Performance Cluster oder Grid.

Im Optimalfall ist eine Anwendung natürlich sowohl vertikal als auch horizontal skalierbar. Insbesondere wenn nicht alle Knoten über dieselben Ressourcen verfügen ist eine vertikale Skalierung auf den einzelnen Knoten auch dort wichtig.

In unserer Arbeit werden wir uns auf die vertikale Skalierung von Software konzentrieren. Insbesondere geht es um die Ausleuchtung der Skalierung auf modernen Multi-Core bzw. Multi-Prozessor Systemen.

Weiterführende Informationen:

- Wikipedia, Scalability: [SCALABILITY]

4.2. Warum überhaupt Parallelisierung?

Die Parallelisierung basiert auf den Prinzip, dass sich eine Aufgabe meistens in mehrere Teilaufgaben zerlegen lässt, die dann unabhängig voneinander abgearbeitet werden können. Da die verteilten Teilaufgaben dann gleichzeitig abgearbeitet werden kann so das Resultat schneller zur Verfügung stehen.

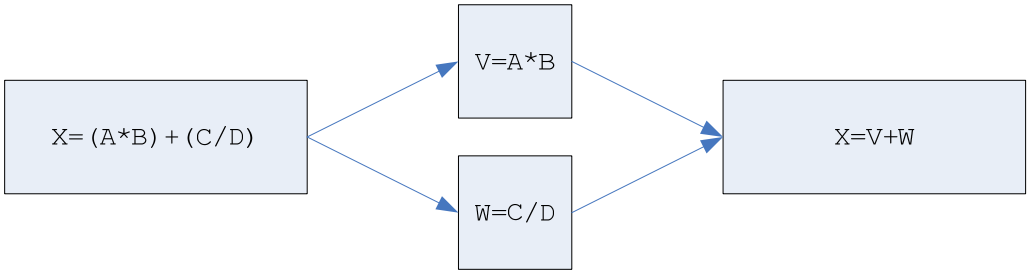


Abbildung 1 Grundprinzip paralleler Verarbeitung

Der Nachteil der parallelen Verarbeitung liegt hier in der Synchronisation der Aufgaben und der Konsolidierung der Teilresultate. Häufig sind die Teilaufgaben auch nicht vollständig ganz unabhängig zu erledigen sondern benötigen Zwischenresultate anderer Teilaufgaben. Im ungünstigsten Fall kann diese Synchronisation und Konsolidierung mehr Zeit in Anspruch nehmen als durch die Parallele Verarbeitung eingespart wird.

Trotz diesen Problemen gewinnt die parallele Verarbeitung von Informationen aktuell immer mehr an Bedeutung. Dies liegt einerseits daran, dass die Hardware-Hersteller an physikalische Grenzen stossen was die Verarbeitungsgeschwindigkeit der Recheneinheiten angeht. Hier spielen Faktoren wie Strukturgrössen, Schaltgeschwindigkeiten von Transistoren und Verlustleistung eine wesentliche Rolle. Aktuell geht der Trend klar weg vom Gigahertz-Rennen hin zu mehr und intelligenteren Verarbeitungseinheiten. Dadurch setzt sich auch das in der Informatik bekannte „Mooresche Gesetz“ (siehe auch [MOORE]) trotz physikalischer Grenzen fort. Es besagt, dass die Komplexität von integrierten Schaltkreisen sich etwa alle 18 Monate verdoppelt.

Die Parallele Verarbeitung an sich ist ein Konzept, welches schon sehr lange existiert. Schon früh hat man erkannt, dass gewisse Aufgaben in externe Recheneinheiten ausgelagert werden können um die Haupteinheit (CPU) zu entlasten. Heutige Systeme besitzen eine Vielzahl von Prozessoren, die alle in einem gewissen Masse unabhängig voneinander arbeiten. Als Beispiele seien hier Grafikprozessoren, Netzwerkprozessoren, Hardware-RAID-Controller oder auch Sound-Prozessoren genannt. Diese Verteilung der Aufgaben an unterschiedliche Hardware wird auch asymmetrisches Multi-Prozessing genannt (siehe Kapitel 5.2).

Diese unterscheiden sich aber in so fern von Multi-Core bzw. Multi-Prozessor Systemen, dass sie alle eine spezialisierte Aufgabe erfüllen und dem Hauptprozessor diese Aufgabe abnehmen können. Das Hauptprogamm wird nicht aufgeteilt und läuft als einzelner Ausführungsstrang auf dem Prozessor ab.

Da Multi-Core Prozessoren häufig mit niedrigeren physikalischen Taktraten betrieben werden als Single-Core Prozessoren werden Anwendungen, welche nur eine Recheneinheit benutzen, durch deren Einsatz tendenziell langsamer. Die Gesamtleistung des Prozessors liegt aber aufgrund mehrfach vorhandener Recheneinheiten höher. Die Kunst der Programmierung besteht nun darin die anstehenden Aufgaben sinnvoll auf die zur Verfügung stehenden Einheiten zu verteilen um schnellstmöglich zum Ergebnis zu kommen.

Weiterführende Informationen:

- Wikipedia, Parallel_computing: [PARALLELISM]
- Wikipedia, Mooresches Gesetz: [MOORE]

4.3. Skalierung als System

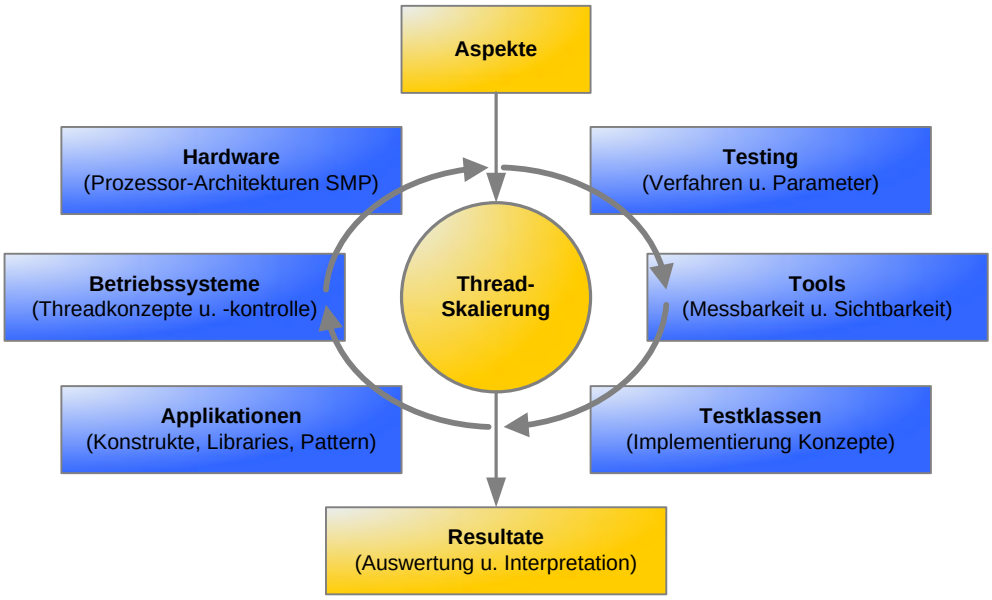


Abbildung 2 Skalierung als System

Zum Begriff der Skalierung gehören Aspekte der Hardware, des Betriebssystems und der Applikationen. Dieses Dokument beleuchtet all diese Aspekte:

- Hardware: Siehe Kapitel 5.
- Betriebssysteme: Siehe Kapitel 6.
- Applikationen: Siehe Kapitel 0 (allgemein) und Kapitel 8 (Java).

Auf der anderen Seite steht die Belegbarkeit der Skalierbarkeit anhand von Tests. Dazu werden einerseits Verfahren und Parameter als auch wichtige Tools kurz erwähnt. Auf der Applikationsebene liegt der Fokus in dieser Hinsicht auf der Implementierung und allgemeinen Konzepten der Skalierbarkeit.

5. Hardware

Dieses Kapitel vermittelt einen Eindruck über die Architektur aktueller Mikroprozessor-Systeme. Die Übersicht soll dabei helfen die Zusammenhänge und Möglichkeiten der Programmierung solcher Systeme besser zu verstehen. Dabei liegt der Fokus auf den Hardware-Aspekten welche eine parallele Verarbeitung ermöglichen.

5.1. Skalierbarkeit der Hardware

Unter dem Begriff der Hardware-Skalierung wir allgemein die Fähigkeit verstanden durch hinzufügen von Ressourcen die Systemleistung zu erhöhen. Die Systemperformance skaliert aber grundsätzlich nie linear. Insbesondere wirkten sich Hardware-Erweiterungen in den seltensten Fällen 1:1 auf die Software-Performance aus. Dies liegt insbesondere darin begründet, dass einige Komponenten (Bus-Systeme) sich nicht einfach erweitern lassen. Beispielsweise kann die menge des Speichers oder die Taktrate des Prozessors verdoppelt werden. Dies wird aber nur in Sonderfällen eine Verdoppelung der Anwendungsleistung zur Folge haben und zwar nur so lange bis ein Bussystem oder eine andere Komponente den Flaschenhals (engl. Bottleneck) darstellt.

Wir wollen uns hier aber nicht auf die Skalierung der Hardware selber sondern auf die Hardware-Aspekte, die eine Skalierung der darauf aufbauenden Software ermöglichen, konzentrieren.

Aktuelle Hardware- und insbesondere Prozessor-Architekturen sind sehr komplexe Gebilde. Abbildung 3 gibt einen Überblick über die wichtigsten Komponenten und Architekturen. Im Folgenden werden die einzelnen Komponenten kurz erklärt. Es versteht sich von selbst, dass es sich hier nur um eine Übersicht und nicht um eine Detaillierte Spezifikation der Komponenten handelt.

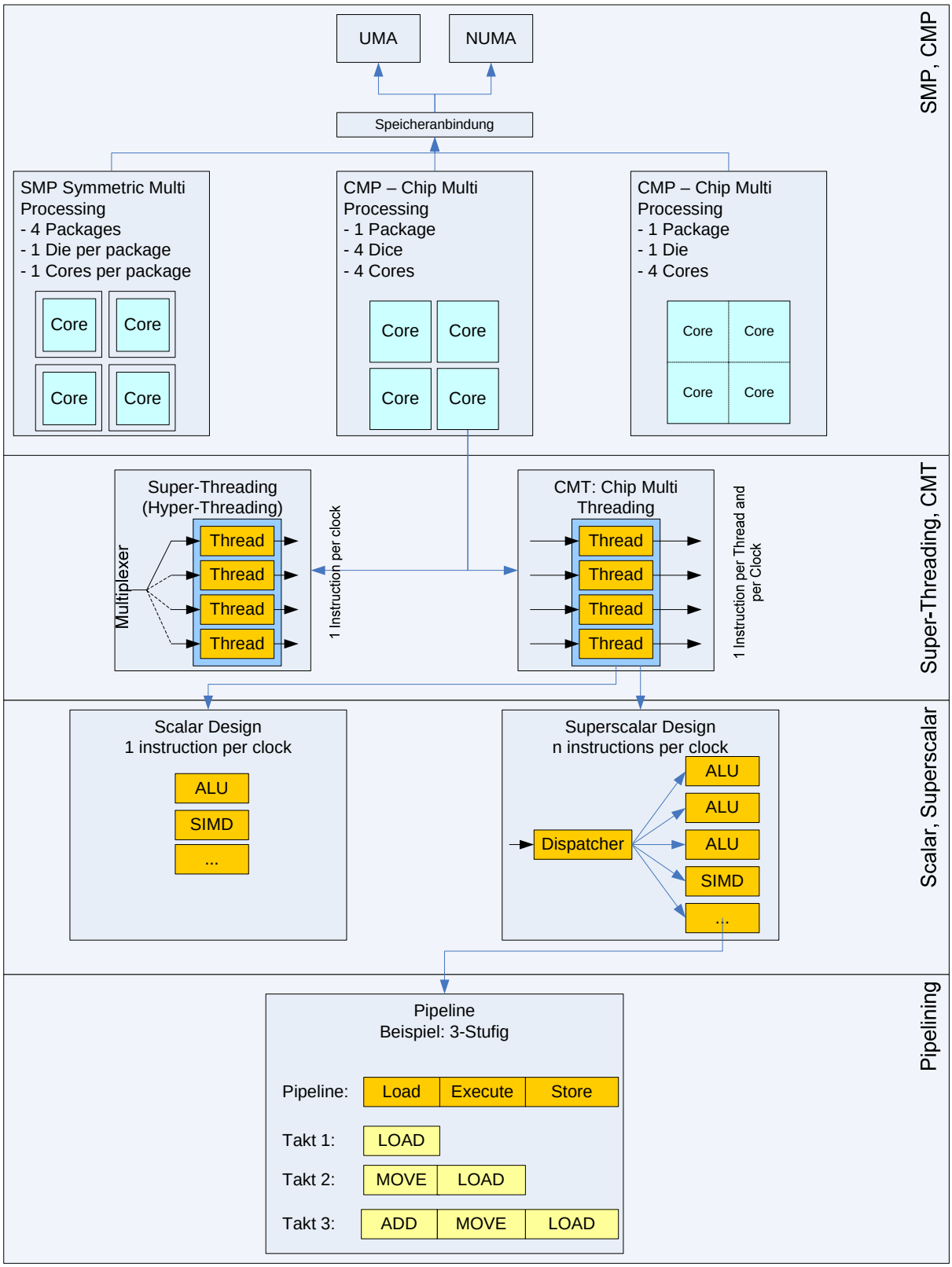


Abbildung 3 Hardware Architekturen

5.2. SMP / ASMP / CMP

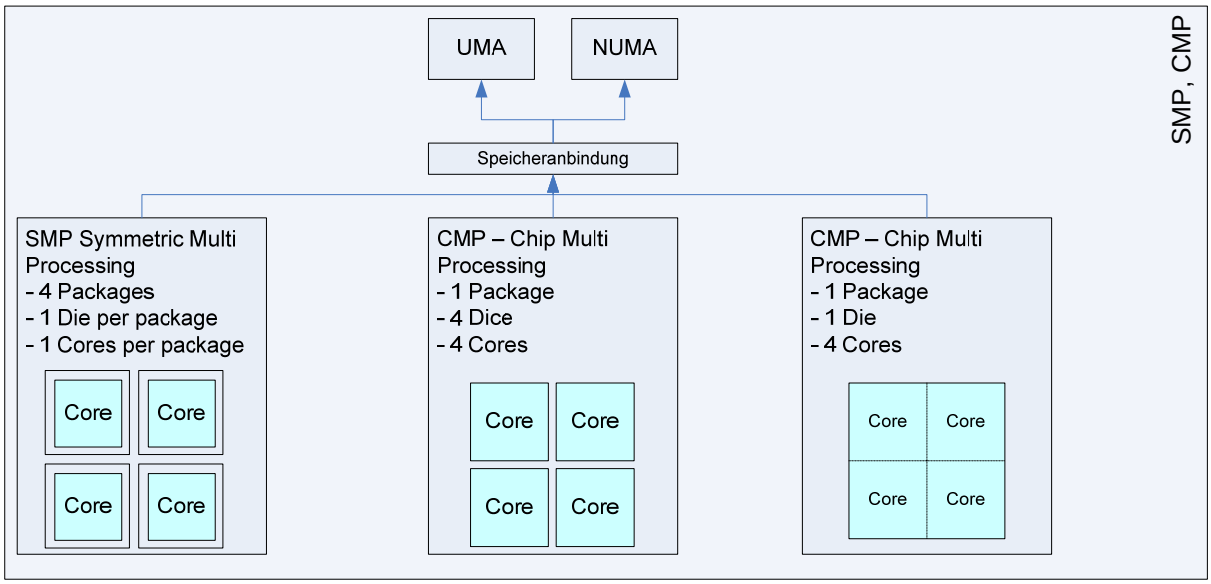


Abbildung 4 SMP, ASMP, CMP

Unter einem SMP (Symmetric Multi Processing) System versteht man eine Architektur bei der alle Prozessoren (2 oder mehr) gleichberechtigt arbeiten. Das heisst, dass jeder von ihnen jede Aufgabe übernehmen kann. Eine Modifikation davon stellen ASMP (Asymmetric Multi Processing) Systeme dar. Bei diesen Architekturen werden gewisse Aufgaben und/oder Ressourcen fest einer Recheneinheit zugewiesen. Beispielsweise ist es praktikabel alle Interrupt-Handling Routinen oder I/O Operationen auf einer dedizierten CPU abzuarbeiten. Der Vorteil von ASMP Architekturen liegt im einfacheren Design (z.B. nur eine CPU braucht Zugriff zum I/O Bus). Der Nachteil gegenüber SMP Systemen liegt darin, dass mit ASMP Systemen häufig nicht die optimale Performance erzielt werden kann.

In gewisser Weise sind alle heutigen PC-Systeme ASMP Systeme da sie für diverse Aufgaben spezialisierte Chips verwenden. Beispielsweise sitzt auf der Grafikkarte meist ein leistungsfähiger 3D-Prozessor, im Chipsatz häufig ein Hardware-RAID Controller und bei Musikfans ein schneller Sound-Chip. All diese Prozessoren nehmen der CPU einige Aufgaben ab sind aber dedizierte Prozessoren für diese Aufgabe.

Chip Multi Processing (CMP) bezeichnet die in jüngster Zeit immer häufig gewordenen Multi-Core Architekturen. Hierbei teilen sich die einzelnen Cores häufig den Cache oder Teile davon. Um die Frage zu klären, ob Architekturen mit mehreren Kernen gleichzusetzen sind mit SMP Architekturen muss man den internen Aufbau mit einbeziehen. Einige Multi-Core Architekturen besitzen zwei physikalisch getrennte Kerne in einem Chip-Gehäuse. Andere wiederum verwenden gemeinsame Caches oder gar Funktionseinheiten und sind auf einem Die vereint (siehe Abbildung 4). Getrennte Kerne verhalten sich nach aussen tendenziell eher wie SMP Systeme. Bei „verschmolzenen“ Kernen kann es dagegen einerseits zu Vorteile (gemeinsame und grössere Caches, schnelle interne Kommunikation) als auch Nachteile (teure Produktion, ungewollte Laufzeit-Abhängigkeiten) kommen.

Allen Architekturen gemeinsam ist aber die Tatsache, dass sie mehrere Recheneinheiten zur Verfügung stellen und ihr volles Potential nur bei parallel abzuarbeitenden Aufgaben ausschöpfen können.

Weiterführende Informationen:

- Wikipedia, Multiprocessing: [MULTIPROC]
- Wikipedia, CMP: [CMP]

5.2.1. UMA/NUMA

Die Verarbeitungseinheiten müssen natürlich auch auf den (gemeinsamen) Speicher zugreifen können. Dafür haben sich im Wesentlichen zwei Technologien durchgesetzt: UMA bzw. NUMA.

Uniform Memory Access (UMA) bezeichnet eine Architektur bei der sich die Prozessoren einen gemeinsamen Speicherbus teilen. Aus diesem Grunde ist auch der Speicherzugriff auf alle Speicherzellen für alle Prozessoren gleich schnell.

Non-Uniform Memory Access (NUMA) Architekturen arbeiten im Gegensatz dazu mit lokalem Speicher. Jeder Prozessor (Node genannt) hat dabei schnellen Zugriff auf den an ihm direkt angeschlossenen Speicher. Trotzdem kann jeder Node über entsprechende Kommunikationskanäle auf den Speicher der anderen Nodes zugreifen. Dieser wird dann als „remote Memory“ bezeichnet. Prinzipbedingt ist der Zugriff auf entfernten (remote) Speicher deutlich langsamer als auf lokal angebundenen. Dafür ist der Zugriff auf den lokalen Speicher üblicherweise schneller als bei einem UMA System. Den Zugriff auf entfernten Speicher und die getrennten Caches der Prozessoren werfen weitere Probleme bei der Cache-Synchronisierung auf. Dies wird auch als Cache Kohärenz bezeichnet (siehe auch [CACHECOH]). Um die Caches konsistent zu halten müssen die Nodes bei Veränderungen die anderen Nodes informieren. Solche Aktualisierungen belasten den Systembus und können die Leistung durchaus auch beeinflussen. Hier soll aber nicht näher darauf eingegangen werden. Heute kümmern sich alle NUMA Systeme automatisch um die Cache Kohärenz, deshalb wird hier die Bezeichnung NUMA als Synonym für die korrekte Bezeichnung ccNUMA (Cache Coherent NUMA) verwendet. Trotz der automatischen Behandlung durch die Hardware kann durch die Verwendung von entferntem Speicher ein Engpass auf dem Systembus und somit ein Leistungseinbruch auftreten.

Beide Technologien haben ihre Vor- und Nachteile. Insbesondere bezogen auf die Geschwindigkeit der Speicherzugriffe. Bei einem UMA-System lässt sich die Geschwindigkeit besser voraussagen als auf einem NUMA System. Bei einem NUMA-System sollte das Betriebssystem die Architektur kennen um Prozesse mit vielen Speicherzugriffen auf einem Prozessor auszuführen an dem die benötigten Daten lokal vorhanden sind. Achtet das Betriebssystem nicht darauf, so kann es vorkommen, dass der Grossteil der Daten aus entferntem Speicher stammt und dadurch massiv langsamer zur Verfügung steht. Ausserdem werden dadurch die Bussysteme zwischen den Prozessoren unnötig belastet. Die Fähigkeit des Betriebssystems mit diesen NUMA-Eigenschaften umzugehen wird mit dem Attribut NUMA-Awareness bezeichnet.

Leider ist die Betriebssystem-Unterstützung noch nicht durchgängig vorhanden. Nach unseren Erkenntnissen unterstützt Windows XP nur in der 64-bit Version und in der 32-bit Version mit dem Kernel-Flag „/PAE“ NUMA. Ausserdem unterstützen einige Windows Server 2003 Versionen NUMA. Siehe dazu auch [MSNUMA].

Weiterführende Informationen:

- Wikipedia, Uniform Memory Access: [UMA]
- Wikipedia, Non-Uiform Memory Access: [NUMA]
- Wikipedia, Cache coherency: [CACHECOH]
- Microsoft, Windows Server 2003 NUMA Support: [MSNUMA]

5.3. Super-Threading, CMT

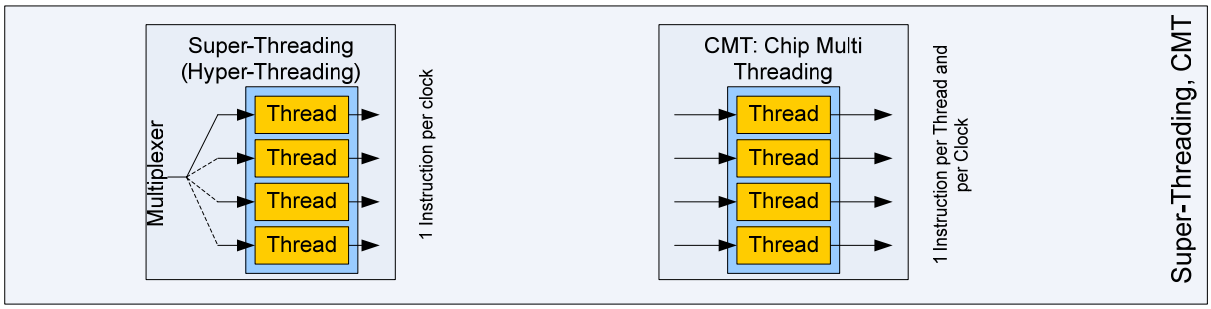


Abbildung 5 Super-Threading, CMT

Wie im Kapitel 5.2 beschrieben geht der Trend in Richtung mehrerer parallel arbeitender Prozessorkerne (Cores). Innerhalb des Prozessors setzt sich dieser Trend fort. Viele aktuelle Prozessoren sind Optimiert auf die parallele Abarbeitung der Instruktionen. Einige gehen aber noch einen Schritt weiter und bieten parallele Abarbeitungspfade für mehrere Threads. Auch hier gibt es mehrere unterschiedliche Implementierungen.

Super-Threading ermöglicht dem Prozessor pro Taktzyklus eine Instruktion eines einzelnen Threads zu laden. Da dieser einerseits meist mehrere Zyklen zur Bearbeitung braucht und andererseits häufig auf Speicherzugriffe warten muss ist es oft nicht weiter tragisch, dass eine Thread-Verarbeitungseinheit nicht bei jedem Zyklus eine neue Instruktion bekommt. Warten auf Speicher wird Memory Stall genannt (siehe Abbildung 6). Durch weitere Optimierung der Recheneinheit (siehe auch Kapitel 5.4)ist es aber durchaus möglich, dass die Verarbeitungseinheiten leer laufen und auf neue Instruktionen warten müssen. Beispielsweise arbeitet Intels Hyper-Threading Technologie (siehe auch [HTT]) nach einem modifizierten Super-Threading Verfahren. Beim Pentium 4 wurde das Hyper-Threading eingeführt um die internen Verarbeitungseinheiten (ALU, FPU, SSE usw.) besser auslasten zu können.

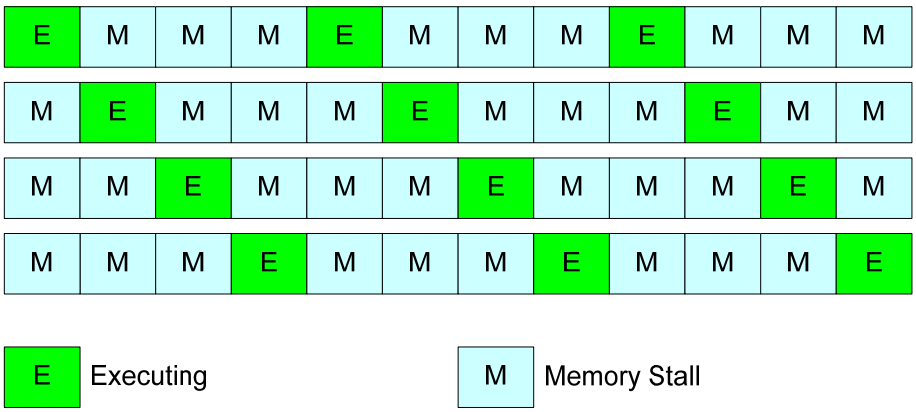


Abbildung 6 Verarbeitung gemäss Super-Threading

Chip Mult Threading (CMT) erlaubt es pro Taktzyklus und Thread eine Instruktion zu lesen. Das heisst, dass bei einem 4-fach CMT System pro Taktzyklus 4 Instruktionen eingelesen werden können (siehe Abbildung 7). Dies stellt natürlich höhere Anforderungen an die innerhalb der Thread-Ausführungseinheiten liegenden Rechenwerke, erhöht aber auch die Ausführungsgeschwindigkeit und Auslastung/Effizienz des Prozessors.

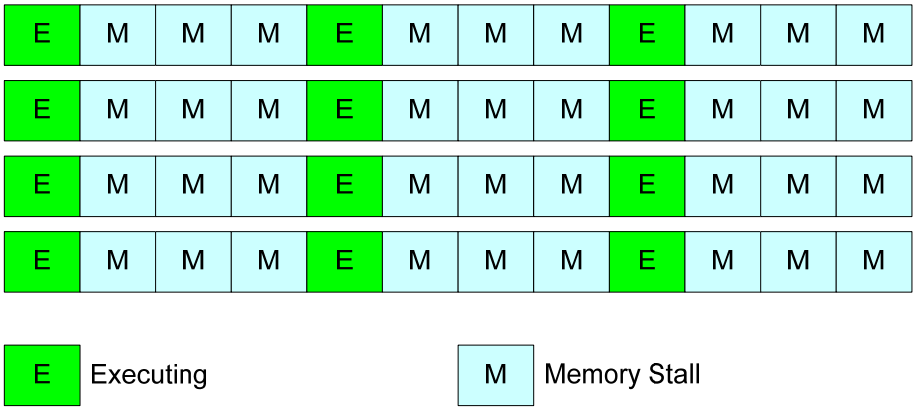


Abbildung 7 Verarbeitung gemäss CMT

In der Praxis laufen natürlich beide Verfahren nicht ganz so geordnet ab wie auf den Illustrationen dargestellt. Die Speicherlatenzzeiten (Memory Stall) variieren je nach angesprochenem Speicher. Beim Super-Threading bedeutet dies, dass die Einheiten nicht optimal ausgelastet sind sobald zwei Threads rechenbereit sind und Instruktionen eingelesen werden sollten. Bei CMT könnten dann alle Thread-Verarbeitungseinheiten zeitgliech wieder mit Instruktionen versorgt werden. Wegen unterschiedlicher Latenzzeiten ist der Extremfall, dass 4 Threads gleichzeitig wieder Rechenbereit sind eher unwahrscheinlich.

Weiterführende Informationen:

- Wikipedia, Super-Threading: [STHREAD]
- Wikipedia, Simultanous Multi Threading: [SMT]
- Wikipedia, Hyper-Threading: [HTT]

5.4. Skalar, Superskalar

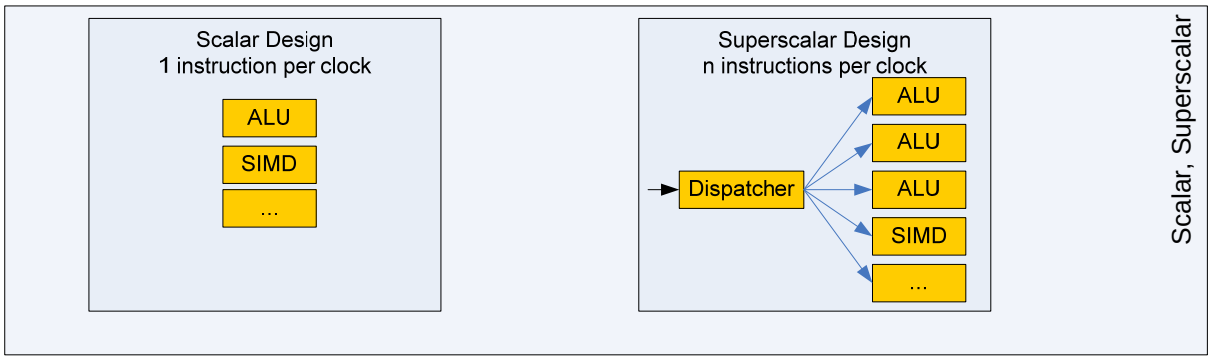


Abbildung 8 Skalar, Superskalar

Skalare Prozessoren arbeiten immer nur an einer Instruktion gleichzeitig und benutzen dabei eine einzige Funktionseinheit. Dies ist zwar einfach, aber ineffizient. Insbesondere liegen alle unbenutzten Prozessorteile dabei brach.

Durch eine superskalare Architektur wird versucht dieses Manko zu beheben. Dies geschieht indem mehrere Recheneinheiten von einem Dispatcher gefüttert werden. Die Aufgabe des Dispatchers ist es die Instruktionen an die freien Einheiten zu übertragen. Dies erlaubt die gleichzeitige Belegung von ALU, FPU oder weiteren Einheiten was einer höheren Effizienz zu Gute kommt. Der Nachteil darin besteht natürlich im höheren Hardware-Aufwand und dem komplexen Dispatching Mechanismus. Beispielsweise können nicht alle Instruktionen parallel auf verschiedenen Einheiten ausgeführt werden wenn sie Resultate einer anderen Instruktion benötigen.

Um die Geschwindigkeit eines superskalaren Prozessors weiter zu erhöhen können auch mehrere Einheiten der gleichen Sorte eingebaut werden. Die meisten aktuellen CPUs besitzen beispielsweise mehrere ALUs.

Aufgrund das einfacheren Designs eines RISC Prozessors (siehe auch [RISC]) ist die Implementierung mehrere Recheneinheiten dort viel einfacher als bei CISC Prozessoren (siehe auch [CISC]). Deshalb wurde diese Technologie bei x86 Prozessoren erst ende der 90er Jahre mit dem Pentium Pro (P6) eingeführt wobei sie für RISC Prozessoren schon anfangs der 80er Jahre eingesetzt wurde. Erst die interne Umsetzung der CISC-Befehle in sogenannte „micr-ops“ erlaubte diese Architektur-Änderung. Daraus ist auch noch ein weiterer wichtiger Aspekt abzuleiten: Heutige x86 Prozessoren zerlegen die x86 Instruktionen in Micro-OPs und arbeiten diese (soweit möglich) parallel ab. Dies kann insbesondere die Reihenfolge der Ausführung beeinflussen.

In den Recheneinheiten selbst liegen ausserdem noch weitere Möglichkeiten der Parallelisierung. Die meisten heutigen x86 Prozessoren bieten einen erweiterten Befehlssatz der hauptsächlich auf die Manipulation grosser Datenmengen optimiert ist. Die erste Erweiterung dieses Typs war die MMX-Erweiterung von Intel (siehe auch [MMX]). Weitere Instruktionen kamen dann mit dem SSE Befehlssatz (siehe auch [SSE]) in diversen Versionen hinzu. Bei all diesen Erweiterungen handelt es sich um sogenannte SIMD (Single Instruction Multiple Data) Befehle (siehe auch [SIMD]). Insbesondere in der Multimediatechnik treten häufig Probleme auf bei denen mehrere Datensätze mit derselben Operation bearbeitet werden müssen. Beispielsweise wenn zu allen Komponenten eines RGBA-Pixels (32-bit) ein Wert addiert werden muss. Ohne SIMD-Instruktionen müssten die 8-bit Komponenten einzeln geladen, verändert und wieder abgespeichert werden. Was einer grossen Anzahl Instruktionen entspricht. Für solche Operationen bieten sich SIMD-Instruktionen and welche dann die Modifikation aller Pixelkomponenten mit einem Befehl erledigt. Intern kann der Prozessor die Modifikation dann automatisch parallel ausführen.

Weiterführende Informationen:

- Wikipedia, Skalare Architektur: [SCALAR]
- Wikipedia, Superskalare Architektur: [SUPSCALAR]
- Wikipedia, RISC: [RISC]

- Wikipedia, CISC: [CISC]
- Wikipedia, MMX: [MMX]
- Wikipedia, SSE: [SSE]
- Wikipedia: SIMD: [SIMD]

5.5. Pipeline

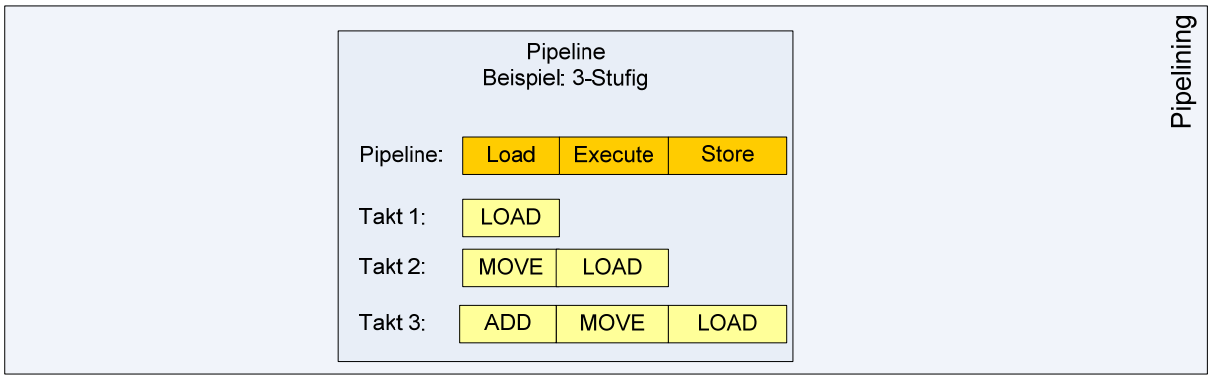


Abbildung 9 Pipeline

Die Abarbeitung einer Instruktion erfordert meistens mehrere Stufen und dauert mehrere Taktzyklen. Die gesamte Arbeitsstrecke wird als Pipeline bezeichnet. Im Beispiel in Abbildung 9 wird exemplarisch eine 3-Stufige Pipeline dargestellt in der eine Instruktion grundsätzlich in 3 Schritten abgearbeitet wird. Die erste Stufe besteht aus dem laden der Instruktion und den dazu benötigten Daten. In der zweiten Stufe wird die Instruktion abgearbeitet und in der dritten wird das Ergebnis zurückgeschrieben.

Würde der Prozessor im Beispiel kein Pipelining unterstützen, so könnte der nächste Befehl erst eingelesen werden, wenn der vorherige beendet ist und die Pipeline verlassen hat. Alle heutigen Prozessoren implementieren aber das Pipelining. Hier kann die nächste Instruktion sofort geladen werden sobald die vorhergehende die erste Stufe verlassen hat.

Als Beispiel kann folgender Code analysiert werden:

```
LOAD  #40,A      ; load 40 to register A
MOVE  A,B        ; copy register A to register b B
ADD   #20,B      ; add 20 to register B
STORE B, 0x300   ; store register B into memory cell 0x300
```

Listing 1 Pipelining Assembler-Code Beispiel

Tabelle 4 Abarbeitung einer Pipeline

Taktzyklus	Aktionen
Takt 1	Der LOAD Befehlt wird aus dem Speicher gelesen (aber noch nicht ausgeführt)
Takt 2	Der LOAD Befehl wird ausgeführt. Gleichzeitig wird der MOVE Befehl aus dem Speicher gelesen.
Takt 3	Der LOAD Befehl speichert den geladenen Wert in das Register A. Gleichzeitig wird der MOVE Befehl ausgeführt. Da dieser aber vom vorhergehenden LOAD Befehl abhängt muss dieser warten bis der LOAD Befehl abgearbeitet ist. Gleichzeitig wird in diesem Taktzyklus die nächste Instruktion (ADD) geladen.
…	…

Wie gut zu erkennen ist erlaubt das Pipelining im Optimalfall die volle Auslastung aller Stufen der Pipeline. In der Praxis führen aber Abhängigkeiten (wie bei der LOAD/MOVE Kombination) zu möglichen Wartezyklen. Im Beispiel handelt es sich um eine sehr kurz gehaltene, beispielhafte Pipeline. In der Praxis liegen typische Pipeline-Längen zwischen 10 und 20 Stufen. Es gibt aber auch Prozessor-Designs mit über 1000 Stufen. Der Pentium 4 beispielsweise besitzt mit 31 Stufen eine extrem lange Pipeline. Darin liegt einer der Hauptgründe warum der Intel Pentium 4 sehr hohe Taktraten erreicht. Doch dazu gleich mehr.

Der Vorteil einer langen Pipeline liegt darin, dass die einzelnen Stufen sehr einfach gebaut (einfachere Logik-Elemente) auf sind. Damit kann die Taktrate erhöht werden. Im Optimalfall kann so ein höherer

Durchsatz erreicht werden. In unserem Beispiel würde nach 3 Taktzyklen Verzögerung das erste Ergebnis bereitstehen (Ergebnis der LOAD-Operation). Danach wäre theoretisch pro Taktzyklus ein weiteres Ergebnis möglich (bei voller Auslastung der Pipeline). Die Verzögerung, die durch die Durchlaufzeit entsteht wird Latenzzeit genannt.

Der Nachteil von langen Pipelines liegt im Programmablauf begründet. Praktisch alle Programme verzweigen sich intern durch bedingte Sprungbefehle (Branch). Tritt ein solcher Sprung im Programm auf gibt es zwei Möglichkeiten für den Prozessor. Einerseits kann er die Pipeline „anhalten“ und muss warten bis der letzte eingegebene Befehl abgearbeitet wurde. Dies ist nötig weil das Ziel des Sprunges erst nach der Abarbeitung des letzten Befehles in der Pipeline feststeht. Dies kostet aber viel Zeit da der erste Befehl nach dem Sprung (bzw. der Sprungbefehl selbst) erst die gesamte Pipeline durchlaufen muss und dadurch eine hohe Latenz im Programmfluss entsteht.

Um das Problem der Sprünge zu entschärfen versucht der Prozessor anhand der bekannten Daten den Sprung vorherzusagen (Branch prediction). Dies funktioniert dank Moderner Algorithmen etwa in 80% der Fälle. Der Prozessor kann dann also die Pipeline weiter befüllen und muss nicht auf das Ergebnis warten. Kritisch wird es nur, wenn der vorhergesagte Sprung falsch ist. In diesem Fall muss die gesamte Pipeline verworfen und neu befüllt werden. Dies hat natürlich für den nächsten Befehl wieder eine hohe Latenz zur Folge.

Lange Pipelines sind also von Vorteil um die Taktrate hoch zu halten aber nachteilig wenn viele Sprünge eintreten. Für wissenschaftliche Berechnungen ohne viele Sprünge (bzw. gut vorhersehbare) ist eine lange Pipeline eher vorteilhaft. Für Desktop-Anwendungen mit schlecht vorhersehbaren Ereignissen können falsche Sprungvorhersagen aber einen massiven Einfluss auf den Durchsatz haben. Dies ist einer der Gründe warum beispielsweise die Leistung des Intel Pentium 4 Prozessors eng an extrem hohe Taktraten geknüpft ist. Auf der anderen Seite steigt mit der Taktrate üblicherweise auch die Betriebsspannung und somit die Verlustleistung eines Prozessors quadratisch. Dies führte zuletzt zu TDP (Thermal Design Power) Werten über 130W (siehe auch [P4TDP] und [P4DTDP]) was selbst Intel dazu bewegte Abstand von der verwendeten NetBurst (siehe auch [NETBURST]) Architektur zu nehmen.

Weiterführende Informationen:

- Wikipedia, Pipelining: [PIPELINE]
- Wikipedia, Netburst: [NETBURST]
- Intel Pentium 4 Processor Thermal Specifications: [P4TDP]

5.6. Konkrete Prozessor-Designs

In Diesem Kapitel werden einige der wichtigen Prozessor-Designs mit ihren speziellen Vorzügen und Nachteilen etwas genauer betrachtet.

5.6.1. Intel Pentium 4

Im Jahre 2000 stellte Intel einen komplett überarbeiteten Prozessor vor. Er bot einige Neuerungen im Vergleich zur bereits 5 Jahre alten Architektur des P6 (Pentium Pro, Pentium II, Pentium III).

Abbildung 10 Intel Pentium 4

Damals war das Gigahertz-Rennen noch in vollem Gange und böse Zungen behaupten, dass Intel die NetBurst Architektur mit ihrer überlangen Pipeline (siehe auch Kapitel 5.5) nur eingeführt hat um mit höheren Taktraten gegenüber der Konkurrenz zu glänzen. Tatsächlich erwies sich die NetBurst Technologie als Sackgasse. Mit der Taktrate stieg auch die Verlustleistung (steigt Quadratisch zur Betriebsspannung und linear zur Taktrate) und führte zu unlösbaren Kühlproblemen. Die ursprünglich geplanten 6 GHz und mehr wurden nie erreicht. Die überlange Pipeline setzte ausserdem einen gewaltigen Aufwand in der internen Architektur voraus. Ausserdem benötigt der Prozessor eine schnelle Speicheranbindung weshalb Intel auf die RD-RAM Technik setzte. Diese erwies sich aber als zu teuer. Mit herkömmlichem SD-RAM wurde die CPU massiv ausgebremst und auf den fahrenden DDR-Zug sprang Intel viel zu spät auf.

All diese Probleme haben dazu geführt, dass Intel das NetBurst Konzept über Bord geworfen hat und auf Basis des ursprünglichen Pentium 3 zuerst den Pentium-M und anschliessend die Core/Core 2 (siehe Kaptiel 5.6.2) Mikroarchitektur entwickelt hat. Neue Prozessoren basierend auf der NetBurst Architektur werden nicht mehr produziert. Insbesondere für die Marketingstrategen wirkt sich dies aber katastrophal aus. Bis anhin vertrauten sie darauf den Kunden die einfache Formel „Hohe Taktrate=Hohe Leistung“ zu verkaufen. Insbesondere war dies effektiv da der Konkurrent AMD mit ihrer Architektur zwar die Leistung aber nie die Taktraten der NetBurst Prozessoren erreichte. Jetzt müssen die selben Marketing-Strategen ihren Kunden beibringen, dass ihre neuen Core 2 Prozessoren mit noch niedrigeren Taktraten als der Konkurrent trotzdem mehr Leistung bringen.

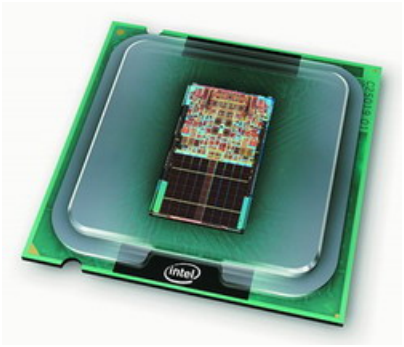
Trotz aller Probleme brachte der Pentium 4 einige Interessante Konzepte und Neuerungen. Insbesondere ist hier das Hyper-Threading (HTT) zu erwähnen. Der Pentium 4 mit HTT war der erste virtuelle Multi-Corre Porzessor für den Desktop-Einsatz. Die zweite Recheneinheit war aber physikalisch nicht vorhanden sondern wurde nur durch einige zusätzliche Register und Einheiten (ca. 5% zusätzliche Chip-Fläche) bereitgestellt. Dadurch konnte die interne Auslastung verbessert werden wodurch eine Geschwindigkeitsvorteil von 15-30% (je nach Applikation auch massiv weniger) erzielt werden konnte.

Hyper-Threading basiert auf dem Prinzip des Super-Threading (siehe Kapitel 5.3).

Weiterführende Informationen:

- Wikipedia, Pentium 4: [PENTIUM4]
- Wikipedia, NetBurst: [NETBURST]
- Wikipedia, Hyper-Threading: [HTT]
- Intel, Pentium 4 TDP: [P4TDP]
- Intel, Pentium D TDP (DualCore): [P4DTDP]

5.6.2. Intel Core/Core 2



Wie in Kapitel 5.6.1 erwähnt hat sich für Intel die NetBurst Architektur als Sackgasse erwiesen. Basierend auf der Architektur der P6 (Pentium Pro, Pentium II, Pentium III) wurde ein Prozessor namens Pentium-M entwickelt. Der Pentium-M wurde als reiner Mobil-Prozessor vermarktet und stellt die Basis der Centrino-Plattform dar. Die Weiterentwicklung wurde von Intel „Core“ getauft und kommt in der zweiten Generation als Desktop-Prozessor auf den Markt.

Abbildung 11 Intel Core 2

Die Core 2 Prozessoren unterscheiden sich wesentlich vom Design des Pentium 4. Insbesondere basieren sie wie erwähnt nicht mehr auf NetBurst sondern auf der Core Mikroarchitektur. Die Pipeline (siehe Kapitel 5.5) wurde massiv verkürzt. Dies beschränkt einerseits die maximale Taktrate, erlaubt aber andererseits eine Effizienzsteigerung (Operationen pro Megahertz). Die Abkehr von den Stromfressenden Pentium 4 Boliden wird auch durch die erstmalige Integration von EIST (Enhanced Intel Speed Step) untermauert. Bisher war SpeedStep nur in Mobilprozessoren verfügbar. EIST erlaubt die dynamische Regulierung der Taktfrequenz und Spannungen im Betrieb um bei geringer Last die Stromaufnahme (und somit die Wärmeabgabe) zu reduzieren.

Um die Effizienz des Prozessors weiter zu erhöhen hat Intel eine Reihe von neuen Technologien entwickelt. Beispielsweise die so genannte „Macro-OP Fusion“. Dadurch kann der Prozessor mehrere Instruktionen zu einer einzigen zusammenfassen und diese in einem Schritt erledigen (z.B. Addition und Multiplikation).

Insgesamt erreicht die Core Architektur eine massiv höhere Effizienz als die NetBurst Architektur was auch der Grund ist warum Intel diese nicht mehr weiterentwickelt und keine neuen Prozessoren mit NetBurst mehr auf den Markt wirft.

Hyper-Threading für den Core/Core 2 sind allerdings nicht in Sicht weil dies im aktuellen Design nicht vorgesehen ist. Stattdessen ist der Core 2 auf Multi-Core Anwendung ausgelegt und ist anfangs auch nur als „Core 2 Duo“ Prozessor in Dual-Core Ausführung erhältlich. Später soll aber auch ein günstiger Core 2 Solo folgen. Noch vor Ende 2006 wird bereits der Core 2 Quad im Einzelhandel erwartet. Dieser wird intern als zwei zusammengeschaltete Core 2 Duo aufgebaut sein (1 Package, 2 Dice, 4 Kerne). Bei den aktuellen Core 2 Duo Prozessoren sind beide Kerne auf einem einzigen Die aufgebaut (1 Package, 1 Die, 2 Kerne). Siehe dazu auch Kapitel 5.2.

Weiterführende Informationen:

- Intel, Core Microarchitecture: [COREARCH]
- Wikipedia, Core 2: [CORE2]

5.6.3. AMD Opteron / Athlon 64

Beim Opteron handelt es sich um die Server-Version des für Desktop-PCs bekannten Athlon 64 Prozessors (auch als K8 bekannt). Im Gegensatz zum Desktop-Prozessor beinhaltet der Opteron nur minimale Anpassungen wie einen modifizierten Speicherkontroller und zusätzliche Kommunikationskanäle für die Mehrprozessor-Kommunikation.

Abbildung 12 AMD Opteron

Der Auffälligste Unterschied der K8 Architektur gegenüber der K7 Architektur (Athlon Classic, Athlon, Athlon XP) besteht in der 64-bit Erweiterung. Im Gegensatz zu Intels Itanium wurde der Prozessor nicht komplett 64-bittig aufgebaut sondern lediglich der 32-bit Prozessor um 64-bit Instruktionen erweitert. Dadurch entfällt bei der Ausführung von 32-bit Code die sehr langsame Emulationsschicht. Deswegen handelt es sich im Grunde auch nicht um einen „echten“ 64-bit Prozessor sondern um einen 32-bit Prozessor mit 64-bit Erweiterungen. AMD hat erkannt, dass die Umstellung auf 64-bit nicht durch einen klaren Schnitt der Architektur machbar ist und die Umstellung der Anwendungen einige Zeit in Anspruch nehmen wird. Mittlerweile hat Intel dies auch eingesehen und die nach AMD benannten AMD64 Erweiterungen (siehe auch [AMD64]) unter dem Namen „Intel 64“ (vormals EM64T genannt) lizenziert.

Mehrprozessorsysteme (SMP) auf Basis des Opteron Prozessors stellen die aktuell bekannteste Implementation eines ccNUMA Systems dar (siehe Kapitel 5.2.1). Die Prozessoren kommunizieren dabei über direkte HyperTransport Kanäle miteinander (siehe [HYPERTRANS]).

Sowohl vom Opteron als auch vom Athlon 64 sind Dual-Core Varianten erhältlich. einzelner Ein Dual-Core Prozessor (Opteron 1xx Serie oder Athlon 64 X2) verhält sich dabei wie ein UMA System, da der gesamte Speicher lokal angebunden ist. In Multi-Sockel Umgebungen (Opteron 2xx, 4xx, 8xx mit 2-8 CPUs) können zwar auch Dual-Core Prozessoren eingesetzt werden, dort verhält sich aber jeder Sockel wie ein Node im ccNUMA System. Beide Kerne können den lokal angebundenen Speicher schnell ansprechen aber müssen entfernten Speicher über die HyperTransport Links ansprechen.

Weiterführende Informationen:

- Wikipedia, AMD Opteron: [OPTERON]
- Wikipedia, AMD64: [AMD64]
- Wikipedia, HyperTransport: [HYPERTRANS]

5.6.4. Sun UltraSparc T1 (Niagara)

Die Bezeichnung SPARC steht für “Scalable Processor ARChitecture“ und bezeichnet eine Prozessorarchitektur. Die Architektur wurde ursprünglich von Sun Microsystems entwickelt und später als offene Architektur von der Non-Profit Organisation SPARC International weiterentwickelt.

Dank der offenen Spezifikation konnten auch andere Hersteller wie Texas Instruments oder Fujitsu SPARC-Kompatible Prozessoren herstellen.

Abbildung 13 UltraSparc T1

Der aktuellste Prozessor dieser Serie ist der UltraSparc T1 (Codename „Niagara) der Firma Sun Microsystems. Der Prozessor beinhaltet einige sehr Interessante Design-Aspekte. Beispielsweise verfügt er mit 8 integrierten Kernen und 4-fach Chip Multi-Threading über 32 logische Einheiten auf einem Chip. Zu beachten ist dabei aber, dass Sun zwar von CMT (Chip Multi-Threading) spricht aber der Chip pro Taktzyklus nur eine Instruktion decodieren kann. Dies entspricht streng genommen dem Super-Threading und nicht CMT (siehe Kapitel 5.3).

Bemerkenswert ist bei dem Chip insbesondere die Leistungsaufnahme. Trotz den 8 Kernen und 32 logischen Thread-Verarbeitungseinheiten benötigt der Chip weniger als 80 Watt. Dieser Umstand hat Sun wohl auch zum Marketing-Schlagwort „CoolThreads“ geführt. Sun führt in Vergleichen auch immer wieder gerne die Einheit „Watt pro Thread“ an. Hier liegt der Chip mit ~2Watt pro Thread den Faktor 20 unter aktuellen Intel Xeon oder den „Power“-Prozessoren von IBM.

Die Speicheranbindung geschieht über einen gemeinsamen Crossbar-Switch. Daher verhält sich der Prozessor wie ein UMA System (siehe Kapitel 5.2.1)

Es versteht sich von selbst, dass ein solcher massiv paralleler Prozessor entsprechend programmiert werden muss. Single-Threaded Applikationen laufen darauf nur sehr zäh ab. Dies hängt auch mit der moderaten Taktrate von 1.0 bis 1.2GHz zusammen. Sein gesamtes Leistungspotential kann dieser Prozessor nur ausspielen wenn er mit vielen (unabhängigen) Threads arbeiten kann. Somit ist er prädestiniert für parallele Anwendungen wie Webserver und Datenbankserver wo sehr viele einzelne Anfragen bearbeitet werden müssen.

Sun hat das Design des Prozessors anfangs des Jahres 2006 unter der Open-Source Lizenz GPL veröffentlicht. Seither hat sich eine beachtliche Community gebildet um die Weiterentwicklung voranzutreiben.

Weiterführende Informationen:

- OpenSPARC, Offene Weiterentwicklung: [OPENSPARC]
- Wikipedia, Sun SPARC: [SUNSPARC]

5.7. Zusammenfassung und Fazit

Es ist gut zu erkennen, dass die parallele Verarbeitung auf Hardware-Ebene extrem aufwändig ist. Für den Programmierer ist kaum mehr nachvollziehbar wie die Instruktionen einer High-Level Entwicklungsumgebung in Hardware ausgeführt werden. Am ehesten geht das noch mit hardwarenahen Sprachen wie C oder C++ aber selbst auf Assembler-Ebene sind die Verarbeitungen kaum mehr Nachvollziehbar. Beispielsweise können mehrere Operationen zu einer zusammengefasst (Macro-OP Fusion) oder eine einzelne Instruktion in mehrere zerlegt und parallel verarbeitet werden (Micro-OPs).

Eine Optimierung auf eine spezifische Hardware-Architektur macht nur selten Sinn da sie sehr aufwändig sein kann, unter Umständen nur eine kleine Verbesserung bringt und die Ausführungsgeschwindigkeit auf ansonsten kompatiblen Prozessoren beeinträchtigen kann. Kaum ein Software Hersteller bietet auf eine spezielle CPU optimierte Software an. Das führt auch dazu, dass häufig nur der kleinste, gemeinsame Nenner der Funktionen benutzt wird. Heutige Compiler bieten zwar häufig die Möglichkeit auf gewisse Architekturen zu optimieren. Soll das Programm aber beispielsweise auf allen x86 gleichermassen laufen so wird man eher darauf verzichten. Vereinzelt bieten Software-Hersteller von Performancekritischen Anwendungen (Compiler-)optimierte Binärdateien an aber das stellt eher die Ausnahme dar.

Allgemein kann man sagen, dass die Hersteller viel unternehmen um bestehenden Code auf neuer Hardware schneller ablaufen zu lassen. Diese Optimierung hat aber mittlerweile (physikalische und technische) Grenzen Erreicht. Nun versuchen die Hardware Hersteller durch Bereitstellung mehrere Parallel arbeitende Einheiten die zur Verfügung stehende Gesamtleistung zu erhöhen. Dies funktioniert aber nur, wenn die Software sich auch parallel verarbeiten lässt (Stichwort: Multi-Threading).

Der Grosse Vorteil von Multi-Threading besteht darin, dass die Software auch noch funktioniert, wenn die Hardware keine parallele Verarbeitung unterstützt. In diesem Fall werden einfach alle Threads Sequenziell (bzw. scheibchenweise) abgearbeitet. Der dadurch erhöhte Verwaltungsaufwand und die resultierende sinkende Gesamtleistung auf Uni-Prozessor Maschinen kann meist in Kauf genommen werden. Ausserdem kann die Anzahl Threads dynamisch (auch zur Laufzeit) an die Hardware angepasst werden. Somit sind bei entsprechender Programmierung keine Sonderversionen für spezielle Systeme nötig.

5.8. Auswirkungen auf die Aufgabenstellung

Gemäss der Aufgabenstellung untersuchen wir die Skalierbarkeit von Anwendungen auf aktueller Multi-Core/Multi-Threading Hardware unter Windows mit Fokus auf die Java-Programmierung. Insbesondere die Java-Programmierung lässt sehr wenig Spielraum für die hardwarenahe Programmierung (siehe Kapitel 8). Java bietet aber eine sehr gute Basis zur Multi-Threaded Programmierung. Dies beinhaltet eine breite Basis vorhandener Klassen (auch Thread-Safe) und die relativ einfache Handhabung von Threads. Somit liegt es nahe hauptsächlich die Skalierung auf Thread-Ebene zu betrachten.

Java bietet ausserdem noch den Vorteil, dass der Bytecode zur Laufzeit mittels Just-in-Time (JIT) Compiler (siehe Kapitel 8.4.1) in Maschinencode umgewandelt wird. Somit ist es möglich ein plattformunabhängiges Programm zur Laufzeit auf Hardware-Spezifische Eigenheiten hin zu optimieren.

Die in diesem Kapitel aufgeführten Hardware-Architekturen und Eigenschaften können dabei helfen die Theoretischen Möglichkeiten auf einer Plattform abzuschätzen und die Ergebnisse besser zu verstehen.

Einige der hier aufgeführten Technologien werden für den weiteren Verlauf dieser Arbeit direkte Bedeutung haben und einige werden nur am Rande (beispielsweise bei der Interpretation der Resultate) eine Rolle spielen

Tabelle 5 Technologien mit direktem Einfluss auf die Arbeit

Technologie	Beschreibung

SMP, CMP	Gemäss der Aufgabenstellung (siehe [5]) ist die Software-Entwicklung auf Mehrprozessor und Multi-Core Maschinen zu betrachten. Diese Technologien gehören also zur zentralen Aufgabenstellung.
CMT	Sollte die Testplattform CMT unterstützen, dann ist dies sicher zu berücksichtigen und abzuklären in wie Fern die einzelnen Threads auf einer CPU parallel ablaufen können.

Tabelle 6 Technologien mit indirektem Einfluss auf die Arbeit

Technologie	Beschreibung
UMA/NUMA	Je nach Verfügbarer Hardware und der darauf laufenden Software kann die Speicher-Architektur einen Einfluss auf die Ausführungsgeschwindigkeit haben. Da uns aber eh kein System bekannt ist, welches auf derselben Hardware UMA und NUMA anbietet ist ein direkter Vergleich sowieso nicht möglich. Die Architektur der Testplattform sollte aber bei den Tests im Hinterkopf behalten werden. Dies einerseits um die Ergebnisse interpretieren zu können und andererseits um eventuelle Optimierungen vornehmen zu könen.
Skalar Superskalar	Da heutige CPUs alle superskalar sind braucht dies nicht näher betrachtet zu werden. Die Performance der CPUs hängt aber zu einem guten Teil von der dadurch erzielten Auslastung der Recheneinheiten ab. Bis so tief in die Hardware-Ebene werden wir aber aus zeitlichen Gründen keine Analyse machen können.
Pipeline	Auch hier gilt dasselbe wie für die Skalarität. Aus zeitlichen Gründen werden wir keine Analyse bis auf die Stufe der Pipeline durchführen können.

6. Betriebssysteme

6.1. Einleitung

Neben der Aufgabe vorhandene Geräte zu verwalten und verschiedenen Softwareanwendungen eine abstrakte Schnittstelle zur Hardware zu Verfügung zu stellen, übernimmt das Betriebssystem auch die Prozess- und Prozessorverwaltung. Im Kontext der Skalierung paralleler Software-Anwendungen bedeutet dies, dass die Zuweisung von Rechenzeit eines oder mehrerer Prozessoren bzw. Prozessorkerne an mehrere Prozesse über das Betriebssystem optimiert gesteuert wird. Im Hinblick auf diese Verwaltung von Prozess und Prozessoren ergeben sich folgende Aufgabenbereiche:

- Prozesserzeugung und Prozessterminierung
- Prozesswechsel
- Verwaltung der Prozesskontrollblöcke
- Prozessablaufplanung und Zuteilung (Scheduling und Dispatching)
- Prozesssynchronisation und Interprozesskommunikation
- Zuteilung von Adressraum an Prozesse
- Interrupt- und Trapbehandlung

Ziel dieses Kapitels ist es, die grundlegenden Aspekte im Bereich Betriebssystem, Prozesse und Prozessmanagements zu vermitteln um die Threadkontrolle auf Level OS beurteilen zu können. Dieser Abschnitt soll das Threadhandling unter Windows XP offen legen und letztendlich Möglichkeiten aufzeigen wie dieses beeinflusst werden kann.

6.2. Windows XP

Die Analyse der Skalierbarkeit einer Java-Applikation soll gemäss Vorgaben auf dem OS Windows XP untersucht werden. Windows XP (NT 5.1) ist ein Betriebssystem der Firma Microsoft und wurde im Oktober 2001 lanciert. Es ist der technische Nachfolger von Windows 2000 (NT 5.0) mit Windows NT-Kern. Zusätzlich löste es Windows ME der MS-DOS-Linie in der Version „Home Edition“ als Nachfolger in der Produktlinie für Heimanwender bzw. Privatnutzer ab. Die MS-DOS-Linie wurde von Microsoft eingestellt. Von Windows XP existieren zahlreiche Varianten. Für diese Arbeit sind folgende Ausführungen denkbar:

Die „Professional Edition“

Für den Einsatz in Unternehmen entwickelt, enthält Funktionen wie bspw. Fernverwaltung (Remote Control), Dateiverschlüsselung (EFS), zentrale Wartung mittels Richtlinien oder die Nutzung von mehreren Prozessoren (SMP).

Die „Home Edition“

Preiswerte Variante um einige Eigenschaften der Professional Edition gekürzt, basiert jedoch auf demselben NT-Kern.

Windows XP „x64 Edition“

Eine spezielle 64-Bit Version, die ausschliesslich für AMD- und Intel-Prozessoren mit x86-64-Erweiterung entwickelt wurde. Sie läuft nicht auf 64-Bit-Prozessoren anderer Hersteller und ist ansonsten identisch zu Windows XP Professional. Die x64 Edition ist als OEM- und als System-Builder-Lizenz erhältlich. Im Zusammenspiel zwischen Prozessor und Betriebssystem kann auch eine konventionelle 32-Bit-Software ausgeführt werden. Somit ist es nicht erforderlich, dass die auszuführenden Programme als 64-Bit-Version vorliegen müssen. Dieses Verfahren des x64-Prozessors wird auch Mixed-Mode genannt - also das Ausführen von 64- und 32-Bit-Software gleichzeitig auf einem Prozessor. Jedoch ist es erforderlich, dass die Treiber als 64-Bit-Version vorliegen. Die Treiber werden vom Hardware-Hersteller für das Betriebssystem hergestellt und zur Verfügung gestellt.

6.2.1. Interne Struktur

Windows 2000/XP ist als Schichtenarchitektur implementiert. Hierbei können die Schichten des Kernel- und User-Mode unterschieden werden. Der Hardware Abstraction Layer (HAL), die Ausführungsschicht (Executive) und der eigentliche Kernel laufen im „Protected-Mode“. Sie haben einen geschützten Speicherbereich der gegen „schädliche“ Prozesse aus dem Userspace abgeschottet ist (bspw. fehlerhafte Applikationen). Windows NT/2000/XP ist in C, zu kleineren Teilen in C++ programmiert. Wenige Softwareteile, die direkt die Hardware ansprechen, sind auch in Assemblersprache kodiert.

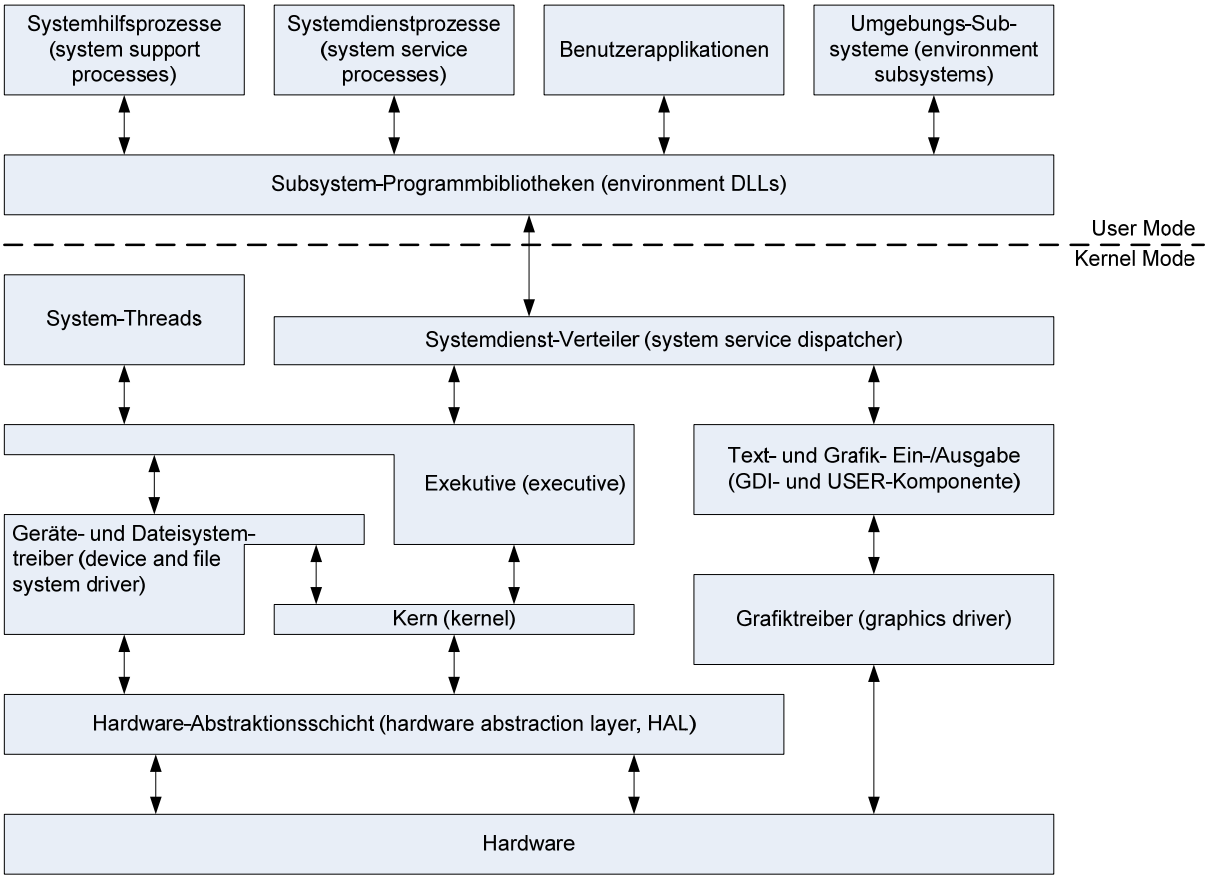


Abbildung 14 Interne Struktur Windows NT/2000/XP

Der Windows Kernel

Der Kernel von Windows 2000/XP ist die zentrale Kommunikationsschnittstelle für die Module der Ausführungsschicht die Aufgaben wie Interrupt- oder Exceptionhandling oder das Scheduling wahrnimmt. Der Kernel ist objektorientiert implementiert und nutzt Dispatcher- und Control-Objekte um seine Aufgaben zu erledigen. Thread-Objekte sind dabei Dispatcher-Objekte die immer zu einem spezifischen Prozess assoziiert werden können und vom Kernel direkt koordiniert werden. Das Timer-Objekt ist ein weiteres wichtiges Dispachter-Objekt, welches die verbrauchte CPU-Zeit überwacht und über allfällige Timeouts oder abgelaufene Zeitscheiben (Scheduling) informiert.

6.3. Das Prozess Modell

6.3.1. Begriff des Prozesses

Ein Prozess wird im allgemeinen Sprachgebrauch als „Programm in Ausführung“ bezeichnet. Er benötigt eine Anzahl physikalischer und logischer Ressourcen wie bspw. Prozessor (CPU), I/O-Geräte, Arneitsspeicher, damit er ablaufen kann.

Eine andere, nachvollziehbare Definition eines Prozesses ist „Der Prozess als Code und Daten im Arbeitsspeicher plus zugehöriger Kontext (Register im Prozessor, Stack, Puffer, Filehandles)“ also Programmanweisungen und Daten, die im Hauptspeicher liegen sowie der Registerbelegung und Verwaltungsinformationen für diesen Prozess.

Pro betrachtete Zeiteinheit kann in einer Einprozessor-Architektur ohne spezielles Zutun (bspw. Chip Multi Threading) gleichzeitig nur ein Prozess ausgeführt werden. Die Prozesse können aber „logisch-parallel“ ablaufen indem der Prozessor nach einer festgelegten Strategie zwischen den Prozessen umschaltet wird (Multiplexing). Die Strategie mit der diese Prozesse auf CPUs verteilt werden wird durch den Scheduler des Betriebssystems bzw. deren Scheduling-Algorithmus bestimmt.

6.3.2. Der Prozesskontext

Die Vergabe von Rechenzeit von einem laufenden an einen lauffähigen Prozess wird als Kontextwechsel (engl. Context-Switch) bezeichnet. Das verwaltende Betriebssystem muss dabei in der Lage sein, den aktuellen „Zustand“ eines Prozesses zu speichern, um diesen zu einem späteren Zeitpunkt wieder aktivieren bzw. reproduzieren zu können. Die Implementierung auf Level OS erfolgt über Prozesstabellen mit einem „Process Control Block“ (PCB) für jeden Prozess.

Der PCB wird auch als Kontext eines Prozesses bezeichnet und enthält die 4 Elemente:

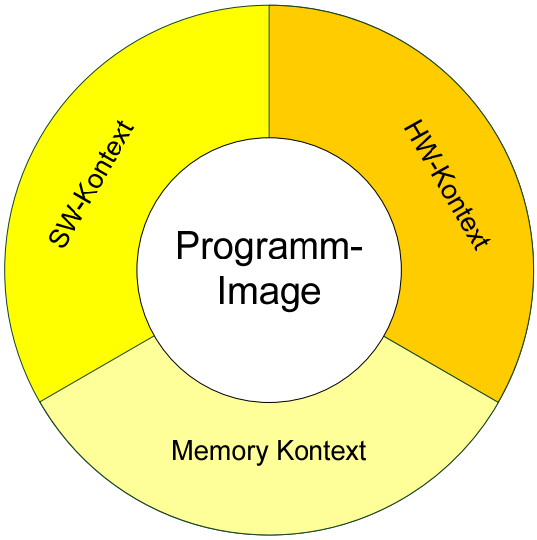


Abbildung 15 Process Control Block

Hardware Kontext

- Zustand des Prozessors, Abbild der Register
- Programm-Counter (PC)
- Prozess-Status-Register (PS)
- Stack-Pointer (SP)
- Allgemeine Prozessor-Register (R#)

Software Kontext

- Verwaltungsdaten für Prozess-Ressourcen
- Prozess-ID
- Zustand Prozess (running, ready, blocked)
- Informationen über I/O (bspw. Offene Files)
- Privilegien

Memory Kontext

- Beschreibung des Adressraumes des Prozessors (bspw. Obere/untere Grenze des zugeteilten Speicherbereichs)

6.3.3. Context-Switch

Ein Context-Switch nennt man den Vorgang, wenn das OS die Abarbeitung eines Prozesses nach einem Interrupt unterbricht und mit einem anderen Prozess bzw. Routine weiterfährt. Für einen Context-Switch, muss das OS den Zustand des bestehenden Prozesses im PCB speichern und den PCB des neuen Prozesses laden. Dieser Vorgang kostet Zeit und kann als „nichtproduktiver Overhead“ bezeichnet werden. Mann kann daher für solche Prozesse folgendes festhalten:

- Die Erzeugung von Prozessen ist sehr aufwendig
- Der Context-Switch ist Abhängig vom PCB ebenfalls aufwendig

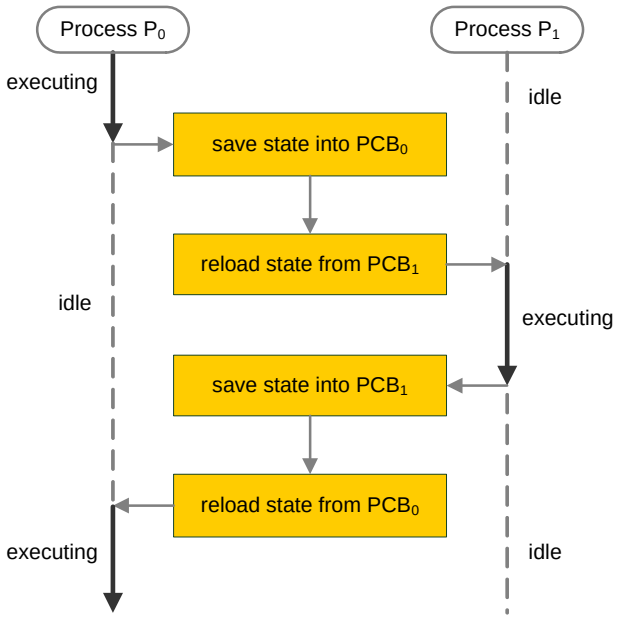


Abbildung 16 PCB

6.3.4. Klassifizierung von Prozessen

Mit Bezug auf den bei der Initialisierung zugewiesene Adressraum eines Prozesses werden im Allgemeinen folgende zwei Kategorien gebildet:

Schwergewichtige Prozesse

- Besitzt eigenen Adressraum
- Prozesswechsel erfordert auch einen Wechsel des Adressraumes

Leichtgewichtige Prozesse (Threads)

- Besitzen gemeinsamen Adressraum
- Prozesswechsel erfordert keinen Wechsel des Adressraumes

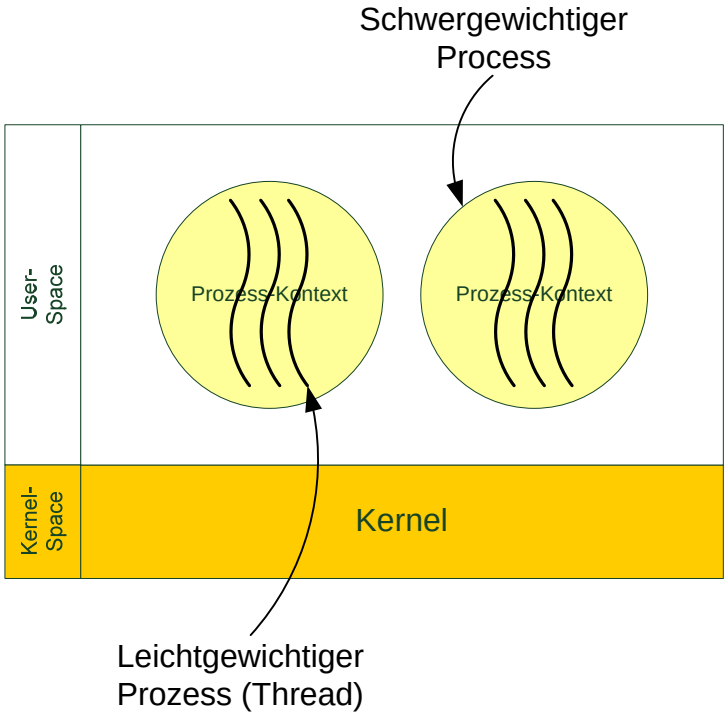


Abbildung 17 Schwer- und leichgewichtige Prozesse

6.3.5. Privilegierungsstufen im OS

In älteren Generationen von Betriebssystemen war es möglich, Programme zu schreiben die auch auf Speicherbereiche des Betriebssystems zugreifen konnten. Durch bewusste oder unbewusste Programmierfehler konnte das Betriebssystem so zum Absturz gebracht werden. Diese Problematik führte zur Einführung von Privilegierungsstufen für Prozesse mit denen Benutzerprozesse in einer Art „Sandbox“ gekapselt werden können. Das Verständnis dieses Models ist wichtig, weil es ein essentieller Unterschied zwischen einem Prozess und Thread darstellt.

In einem „autoritären“ Betriebssystem erfolgt die Implementierung von Prozessen unter Verwendung verschiedener Privilegierungsstufen, die den sichtbaren bzw. verwendbaren Adressraum und Befehlssatz eines Prozesses eingrenzen. In diesem Sicherheitskonzept kann ein so genannter „unprivilegierter“ Prozess (bspw. Ring 3) nicht direkt auf HW-Ressourcen zugreifen oder gar den Speicher eines „privilegierteren“ Prozesses im Ring 0-2 beschreiben.

Intel-Prozessoren seit dem 386er unterscheiden 4 verschiedene Sicherheitsstufen (Modi) über welche die Privilegierung stufenweise eingeschränkt wird. Der Modus mit dem stärksten Schutz wird als **Kernel-Mode** und der mit dem geringsten als **User-Mode** bezeichnet.

Die verbreiteten Betriebssysteme für x86 (dazu gehören Linux und Windows) nutzen lediglich 2 der 4 möglichen CPU-Ringe. Im Ring 0 werden der Kernel und alle Hardwaretreiber ausgeführt, während die Anwendungssoftware im unprivilegierten Ring 3 arbeitet.

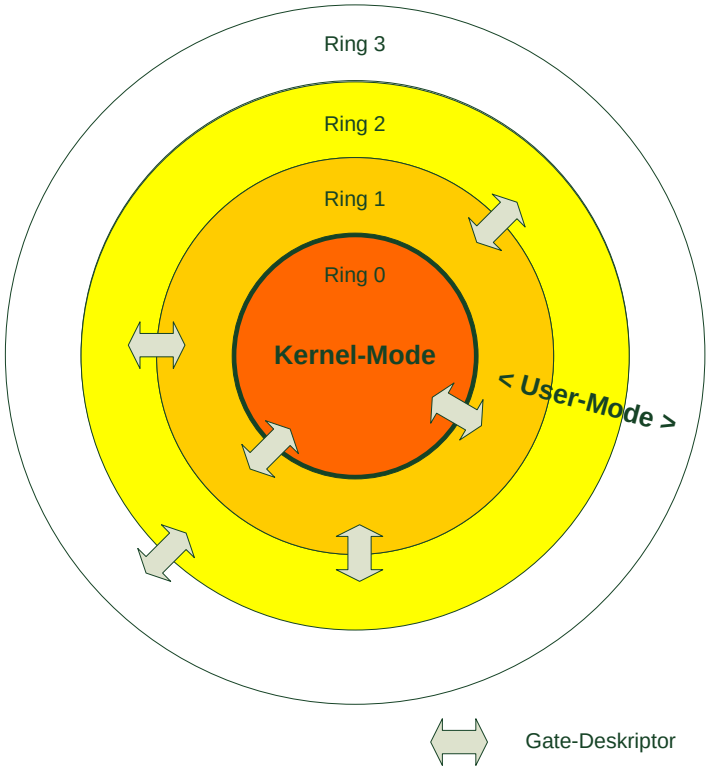


Abbildung 18 User- und Kernel-Mode

Will nun ein Benutzerprozess die Dienste des Betriebssystems nutzen, kann er dies über Systemcalls realisieren. Dabei erfolgt ein Context-Switch bei dem die Kontrolle des Programms auf das Betriebssystem übergeht. Das OS im Kernel-Mode hat Zugriff auf alle Ressourcen und Speicherbereiche und kann so die geforderte Aufgabe erfüllen. Nach erfolgreichem Abschluss erfolgt ein erneuter Context-Switch und der Übergang in den User-Mode.

Das “Schlupfloch” oder die Möglichkeit für einen weniger privilegierten Prozess, die API des Kernels zu nutzen, wird über Gate-Deskriptoren realisiert. Sie repräsentieren einen kontrollierten Übergang von einem privilege-Niveau zu einem anderen. Die Deskriptoren sind in der Global Deskriptor Table (GTD) gespeichert, die für alle Prozesse zugänglich ist.

6.4. Das Thread-Modell

Ein Thread ist die ausführbare Einheit (execution) eines Prozesses. Threads erweitert die oben beschriebene Prozessdefinition um die Möglichkeit, mehrere voneinander unabhängige „Ausführungsfäden“ in einer Prozessumgebung (Prozesskontext) laufen zu lassen.

Threads sind leichtgewichtig, d.h. sie bestehen aus einem Programm-Counter (PC), einem Register-Set und einem Stack. Sie laufen im Kontext eines Prozesses ab und teilen dabei mit einem „Peer-Thread“ (Thread im gleichen Prozess-Kontext) Elemente wie bspw. Adressraum, globale Variabeln, oder geöffnete Dateien.

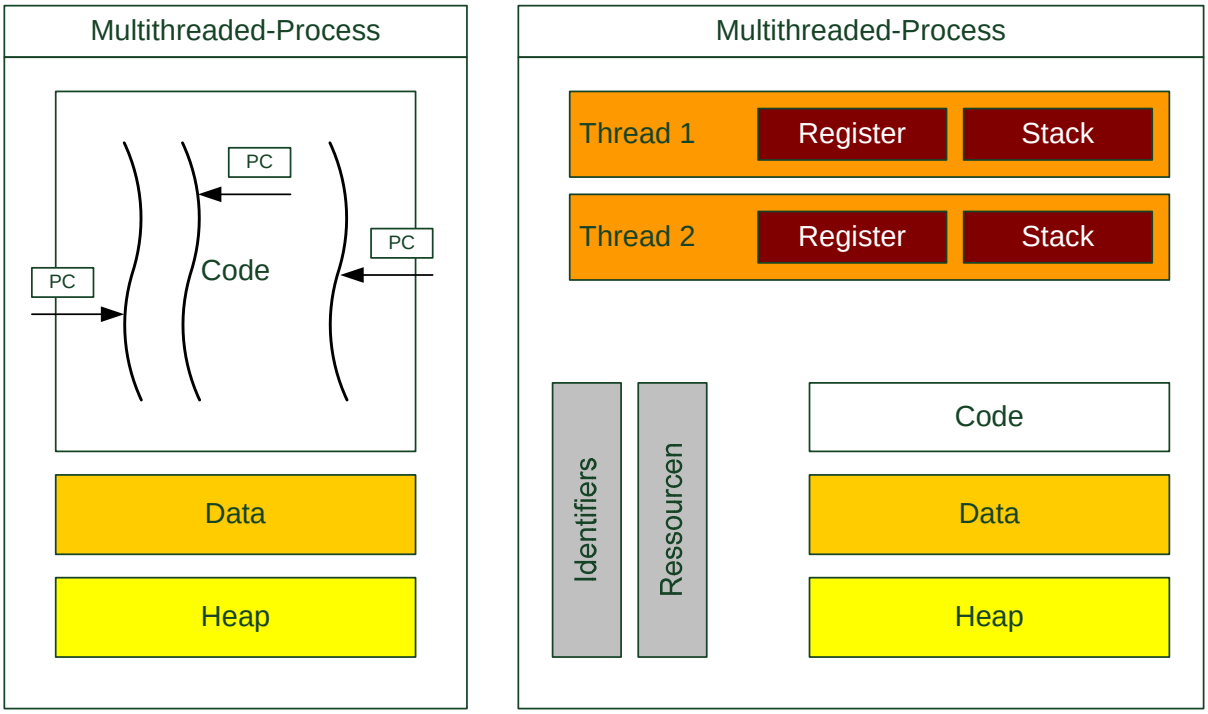


Abbildung 19 Multithreaded Process

In einem Multithreaded-Process (ein Prozess mit 2...n Threads) werden vom Prozesskontext gemeinsam genutzte Elemente der Threads bereitgestellt. Durch die gemeinsame Nutzung entsteht der Vorteil, das Threads mit wenig Aufwand erzeugt werden können (unter der Voraussetzung, dass der Prozess bereits existiert). Der Context-Switch von Threads ist ebenfalls effizienter da nicht der “ganze” Prozesskontext ausgetauscht werden muss. Bei einem Threadwechsel muss lediglich der Threadkontext berücksichtigt werden.

Da Threads, die demselben Prozess zugeordnet sind, den gleichen Adressraum verwenden, ist eine Kommunikation zwischen diesen Threads von vorneherein möglich (Interprozesskommunikation). Diese Vorteile durch gemeinsam genutzte Kontexte, birgt aber den Nachteil, dass mit Synchronisations-Massnahmen gezielt Konflikte im Zugriff auf Speicherbereiche abgefangen werden müssen.

6.4.1. Der Threadkontext

Der Threadkontext hängt im Wesentlichen von der Prozessor-Architektur ab. Ein Kontextwechsel umfasst im Allgemeinen das Sichern und Laden folgender Daten eines Threads:

Threadkontext

- Program counter
- Statusregister des Prozessors
- Weitere Register des Prozessors
- User- und Kernel-Stackpointer
- Pointer zum Adressraum in dem der Thread läuft

6.4.2. Klassifizierung von Threads

Es existieren zwei grundsätzlich verschiedene Arten, wie Threads implementiert werden können. „Richtiges“ Threading wird die Methodik genannt, in der Threads im Kernel implementiert werden (Kernel-Space). Dabei kennt der Kernel jeden Thread was eine Voraussetzung für die Verteilung dieser Threads auf mehrere CPUs ist.

Eine Spezialisierung ist die Implementation im User-Space. Dabei sieht der Kernel nur den Prozess mit dem Initialthread, nicht aber die 2...n Threads die möglicherweise in diesem einen Prozesskontext laufen. Es ist offensichtlich, dass die Verwaltung solcher Threads nicht vom Kernel übernommen werden können. User-Level-Threads benötigen eine Runtime-Umgebung welche die Verwaltung dieser Threads übernimmt. Da die Verwaltung bei der Runtime-Umgebung liegt, kann der Kernel diese User-Threads auch nicht auf mehrere CPUs verteilen.

Als Mischform der beiden oben genannten Varianten kann die Hybride Implementierung betrachtet werden. Sie verbindet die Vorteile beider Implementierungsarten.

Kernel-Level-Threads (Abbildung 1:1)

Die Verwaltung dieser Thread erfolgt über eine Thread-Tabelle, die im Kernel angelegt ist. Analog der üblichen Prozesstabelle mit den PCBs, ist im Kernadressraum auch die Threadtabelle angelegt. Der Kernel-Thread besitzt folgende charakteristische Eigenschaften:

- Thread Im Kerneladressraum
- Kernel verwaltet Prozesse und Threads
- Operation zur Verwaltung von Threads über Systemcalls
- Kernel besitzt Prozesstabelle (PCB) und Threadtabelle
- Ausführung der Threads erfordert keine Laufzeitumgebung

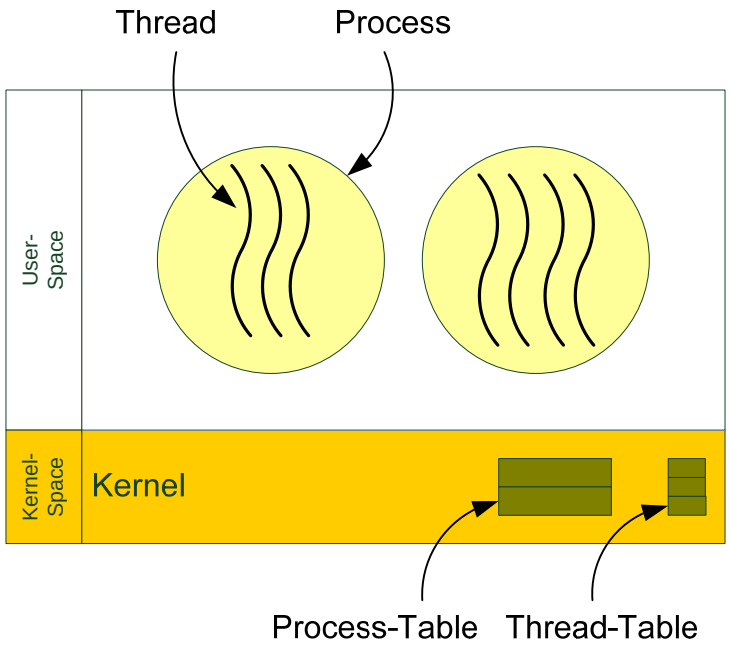


Abbildung 20 Kernel-Level-Thread

Vorteile Kernel-Level-Threads

- Keine blockierenden Systemaufrufe
- Threads können vom Kernel direkt auf CPUs verteilt werden
- Portierbarkeit der Applikationen geringer (Threadoperationen über Systemcalls)

Nachteile Kernel-Level-Threads

- Threadwechsel benötigt Mode-Wechsel zum Kern (Kontext-Switch)
- Erzeugen, Beenden, etc. benötigt Systemcall (Kontext-Switch)
- Aufwand für Erzeugung grösser (bspw. Solaris/Sparc2;):

- Benutzer-Thread 52 µSec
- Kernel-Thread 350 µSec
- Prozess 1700 µSec

Kernel-Level-Threads (KLT) finden in Windows NT/2000/XP und Linux Anwendung

User-Level-Threads (Abbildung 1:n)

Die Implementierung und Verwaltung dieser Threads erfolgt über ein Runtime-System auf Library-Ebene. Analog dem Kernel verwaltet die Runtime-Umgebung Thread mittels Thread Control Blocks (TCBs). Diese Art Threads zu realisieren, hat den Vorteil, dass sie unabhängiger vom OS gestaltet werden kann und somit die Poritierbarkeit einer Applikation eher gegeben ist. Der Context-Switch ist ebenfalls performanter da er ohne Einwirkung des Kernels gemacht werde kann.

Im Kontext der Skalierbarkeit von Applikationen zeigt sich hier aber der klare Nachteil, dass mehrere User-Threads (im gleichen Prozesskontext) vom Kernel nicht auf CPUs verteilt werden können. Der Kernel kennt lediglich den Initialthread des Prozesses. Ein weiterer Nachteil zeigt sich dadurch, dass bei einem blockierenden Systemaufruf eines Threads seine Peer-Threads ebenfalls blockieren.

Der User-Level-Thread hat folgende charakteristische Eigenschaften:

- Thread im Benutzeradressraum
- Kernel kennt Threads in Prozess nicht
- Kernel verwaltet nur „gewöhnlicher“ Prozess (PCB)
- Threads durch Laufzeitumgebung verwaltet (TCB)

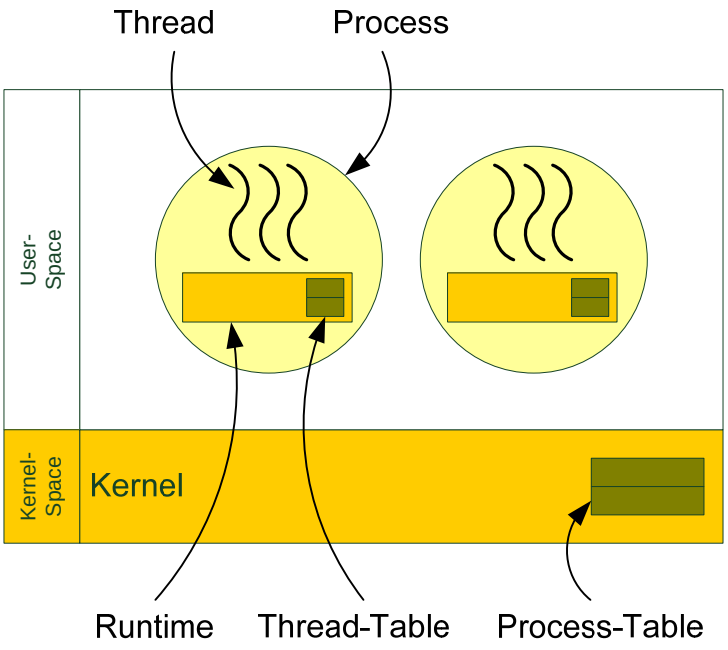


Abbildung 21 User-Level-Thread

Vorteile User-Level-Threads

- Für Thread Wechsel benötigt kein Kernel-Mode Privilegien
- Anwendungsspezifisches Scheduling der Threads möglich
- Unabhängig vom Betriebssystem.

Nachteile User-Level-Threads

- Blockierender Systemaufruf möglich
- Scheduling muss von der Laufzeitumgebung sichergestellt werden
- Keine Verteilung auf mehrere CPU möglich

User-Level-Threads finden unter Unix und Linux Anwendung.

Hybride-Threads (Abbildung m:n)

Hybride Threads sind eine Kombination von Kernel- und User-Level-Threads. Ziesetzung ist dabei, die Vorteile beider Modelle zu vereinen. In einem kombinierten System erfolgt die Thread-Erzeugung vollständig im Userspace, ebenso Teile des Scheduling und der Synchronisation.

Die User-Level- Threads werden einer geringeren oder gleichen Anzahl von Kernel-Level-Threads zugeordnet. Möglich wird dies durch das Konzept der „leichtgewichtigen Prozesse“ (LWP), die „Mittler“ im Userspace für die Kernel Threads sind. Der Entwickler hat die Möglichkeit zu bestimmen, welche und wie viele Threads auf einen LWP abgebildet werden sollen.

Die Threads sind in diesem Fall ungebunden, da sie nach der Implementierung um einen LWP konkurrieren. Die LWPs besitzen zu den User-Level- Threads eine 1:n Beziehung und eine1:1 Beziehung zu den Kernel-Level-Threads. Sie werden analog der 1:1-Abbildung als Kernel-Level-Threads behandelt.

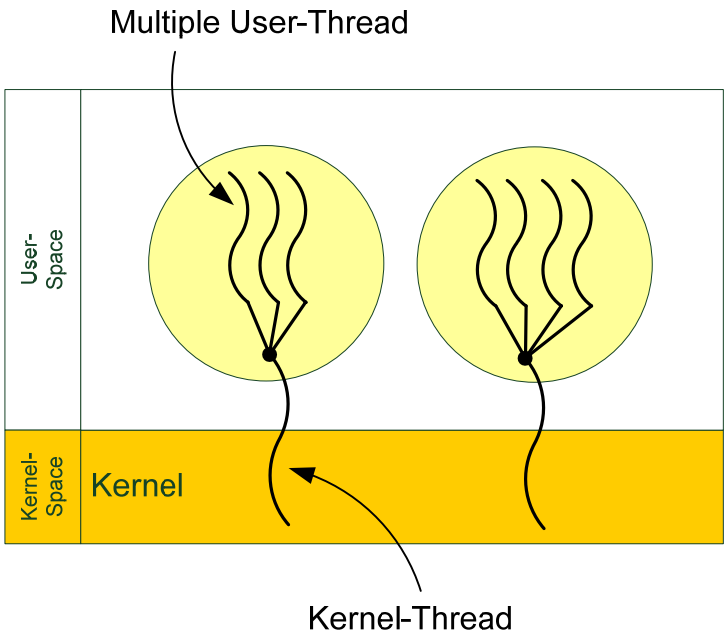


Abbildung 22 Hybride Threads

6.5. Prozessmodell Windows

Unter Windows werden Prozesse als Objekte implementiert. Die Prozesse sind gleichgestellt, d.h. sie stehen auf der gleichen Ebene. Die baumartige Prozesshierarchie wie unter UNIX, in der 1 Vaterprozess 1..n Kindsprozesse haben kann, existiert unter Windows nicht. Windows kennt in Bezug auf ihr Prozessmodell die Objekttypen Job, Prozess, Thread und Fiber.

6.5.1. Objekttypen

Job

Ein Job ist eine Menge von Prozessen, die gemeinsame BM-Quoten, Zeitbegrenzungen und ev. Zugriffsbeschränkungen haben.

Prozess

Ein Prozess ist ein Objekt, das Betriebsmittel (Ressourcen) einschliesslich eines Adressraums hat. Jeder Prozess hat einen geschützten Adressraum von 4 GB.

Thread

Ein Thread ist ein Ausführungspfad innerhalb eines Prozesses. Jeder Prozess startet mit einem Initialthread. Anschliessend können beliebige zusätzliche Threads erzeugt werden. Diese Thread-Funktionalität auch bekannt als Win32-Thread ist direkt in die Kernel32.dll eingefügt und steht für die Windows Betriebsystem der NT-Familie (Win2000, XP, Vista) zu Verfügung.

Da die Entwicklung von .NET-Anwednungen mit .NET-Threads von Microsoft gezielt gefördert wird, verliert der Win32-Thread in diesem Bereich an Bedeutung. Allerdings ist für nicht .NET-Applikationen und im Bereich der Echtzeitprogrammierung der Win32-Thread nach wie vor wichtig.

Win32-Threads bieten auch eine grosse Funktionsvielfalt von Mutexen und Semaphoren bis Message-Queues und Task-Pools. Win2000 und WinXP realisieren diese Threads als Kernel-Threads.

Fiber

Ein Fiber kann unter Windows als „Thread in einem Thread“ betrachtet werden. Er liegt vollständig im Benutzerbereich und ist somit für den Kernel nicht sichtbar. Das Scheduling eines solchen Fibers ist nonpreemtive, d.h. dass ein Fiber muss die Kontrolle über ein Win32-Thread selbständig übergeben. Der aktuelle Fiber übernimmt für die Zeit der Ausführung die Identität des Threads in dem er läuft. Blockiert ein Fiber, blockieren auch die anderen Fibers dieses Threads bzw. der Thread selbst.

Fibers können schnell gestartet und auf eine CPU verteilt werden, da es sich um eine vom Kernel unabhängige User-Level Implementierung handelt. Die Umschaltung von Fibers ist wesentlich effizienter als eine Threadumschaltung.

Die Unterstützung nur einer CPU fällt dabei nicht ins Gewicht, da man bereits durch eine geeignete Aufteilung der Win32-Threads für eine gute Lastenbalancierung sorgen kann. Geeignet sind Fibers für Anwendungen, bei denen kurze Antwortzeiten im Vordergrund stehen.

6.5.2. Abbildung von Threads

Windows 2000/XP realisiert mit seiner Prozess/Thread-Architektur das Kernel-Level-Thread-Modell (1:1-Zuordnung) in dem in einem Prozesskontext mehrere Kernel-Level-Threads laufen die vom Kernel direkt verwaltet werden. Der Kernel verteilt dabei nach einem „priority-driven, preemtive scheduling system,“ die CPU(s) an mehrere Threads unterschiedlicher Prozesse.

6.5.3. Threadzustände

Windows 2000/XP definiert in Bezug auf Threadzustände folgende Möglichkeiten:

Tabelle 7 Threadzustände in Windows

Zustand	Code	Beschreibung
Waiting	5	Thread wartet auf I/O, falls Ressource verfügbar folgt Zustand Ready
Ready	1	Thread ist lauffähig (vom Scheduler berücksichtigt)
Running	2	Thread in Ausführung
Standby	3	Nächster Thread in Ausführung (ausser es folgt Änderung Priorität oder Interrupt)
Transition	6	Ablaufbereiter Thread, dessen Ressourcen nicht verfügbar sind
Terminated	4	Beendeter Thread

Zu einem bestimmten Zeitpunkt kann nur ein Thread pro Prozessor(kern) im Zustand Running sein. Alle anderen lauffähigen Threads sind dann im Zustand Waiting oder Ready. Ein laufender Thread wird ausgeführt bis eine der folgenden Übergangsbedingung eintritt:

- Der Thread überschreitet die zulässige Ausführungszeit (Timesclice, Quantum)
- Ein höher priorer Thread geht in den Zustand Waiting über

Der laufende Thread wird durch ein I/O-Ereignis in den Zustand Waiting versetzt

6.6. Das Prozessmodell Java

Seit Java 1.0 sind Threads fester Bestandteil der Java-Standardbibliothek. Durch verschiedene Implementierungen der JRE gibt es aber auch Unterschiede im verwendeten Thread-Modell. Ältere Implementierungen beispielsweise nutzen noch keine Kernel-Threads womit die Voraussetzung für Multi-Prozessor-Unterstüzung nicht gegeben ist.

6.6.1. Klassifizierung

In einem Java-System gibt es zwei Arten von Threads: User Threads und Daemon Threads. Normalerweise werden Daemon Threads durch das System erzeugt. Daemon Threads werden bis auf eine Ausnahme gleich behandelt wie User Threads. Sie haben eine Priorität, haben dieselben Methoden und Zustände. Das einzige Mal wo die Java Virtual Machine prüft ob es sich bei einem Thread um einen Daemon- oder einen User Thread handelt, ist wenn ein Thread terminiert. Handelt es sich bei diesem Thread um einen User Thread, so wird geprüft, ob noch weitere User-Threads vorhanden sind. Ist dies nicht der Fall, sind also keine oder nur noch Daemon-Threads vorhanden, so terminiert das Programm. Der Grund dafür ist einfach. Daemon Threads sind als Server-Threads für die User Threads gedacht. Gibt es keine User-Threads mehr, so gibt es nichts mehr zu bedienen. Ein typisches Beispiel für einen Daemon Thread ist der Garbage Collector.

6.6.2. Erzeugung

In Java ist ein Thread eine Instanz einer von der Klasse Thread abgeleiteten Klasse. Dabei wird die Methode `run()` überschrieben und darin definiert, was der Thread tun soll. Eine solche Klasse kann weitere Methoden haben, die nichts mit dem Thread zu tun haben. All diese Methoden, sowie alle Instanzvariablen einer solchen Klasse werden, abgesehen von den Sichtbarkeiten, gleich behandelt wie andere Klassen. Ein Thread kann nur einmal gestartet werden! Soll die `run()`-Methode ein zweites Mal ausgeführt werden, muss zuerst eine neue Instanz erzeugt werden.

Eine andere Möglichkeit besteht darin, dass die Thread-Klasse das Interface `Runnable` implementiert. Dieses Interface definiert eine einzige Methode `run()` ohne Parameter. Die Klasse Thread selber implementiert Runnable. Dies wird insbesondere dann benötigt, wenn die Thread-Klasse bereits von einer anderen Klasse abgeleitet ist (Java unterstützt keine Mehrfachvererbung)

Siehe dazu auch Kapitel 8.1.1.

6.6.3. Kontrolle

Die Kontrolle bzw. das Laufzeitverhalten von Java-Threads kann durch den Aufruf mehrerer Methoden gezielt gesteuert werden.

Siehe dazu auch Kapitel 8.1.1.

Die Wichtigsten Methoden der Threadkontrolle sind ebenfalls im Kapitel 8.1.1 erklärt.

6.6.4. Laufzeitumgebung eines Thread

Die Java Virtual Machine (JVM) bildet die Laufzeitumgebung der Threads in Java. Sie ist in Software realisiert und bildet einen Layer zwischen der Hardware-Plattform und dem Java-Programm. Der JVM obliegen für die korrekte und sichere Ausführung von Java-Programmen zahlreiche Aufgaben. Eine zentrale Aufgabe ist die Verwaltung von Java-Threads. Da die JVM nicht a priori auf Multithreading-Funktionalität des darunterliegenden Betriebssystems zählen kann, muss diese Aufgabe selber in die Hand nehmen können. Über einen Threadverwalter werden Threads in einem Zeitmultiplex-Verfahren „zwangssequenzialisiert“. Das bedeutet, dass zur Laufzeit laufende Threads zu Gunsten eines anderen unterbrochen werden und der Bytecode-Ausführer immer nur ein Thread gleichzeitig ausführt.

6.6.5. Abbildung auf OS-Threads

Die Art und Weise wie Java Threads auf OS-Threads abgebildet werden, entscheidet letztendlich darüber ob eine Skalierung dieser Threads bzw. Applikation überhaupt möglich ist. Sind die Java-Threads für das Betriebssystem nicht sichtbar, können diese auf einem SMP-System auch nicht verteilt werden. Die Kombination JVM-/OS-Ausführung führt in diesem Zusammenhang zu „green“- oder „native“ Threads.

Green-Threads

Green Threads sind simulierte Threads innerhalb des Virtual-Machine-Prozesses. Sie werden in der Virtual Machine selbst realisiert weil das Betriebssystem keine Threads unterstützt.

Native Threads

Unterstützt das Betriebssystem des Rechners, auf dem die JVM läuft, Threads direkt, so nutzt die Laufzeitumgebung diese Fähigkeit in der Regel. In diesem Fall haben wir es mit nativen Threads zu tun. Unter Windows NT/2000/XP werden Java-Threads auf Threads im Betriebssystem abgebildet, die innerhalb des Virtual-Machine-Prozesses ablaufen

6.7. Prozessverwaltung durch Scheduling

In einem Multi-Threaded Anwendung konkurrieren mehrere lauffähige Prozesse um die Rechenzeit des Prozessors. Das Scheduling-Modul der Ausführungseinheit des Betriebssystems hat die Aufgabe diese Rechenzeit an die Prozesse in der Ready-Queue zu verteilen. Die Zuweisung erfolg dabei nicht willkürlich sonder bspw. nach dem Optimierungsprinzip folgender Bereiche:

- Fairness der Prozesse (kein Starving)
- Berücksichtigung der Wichtigkeit von Prozessen (Priority)
- Maximale HW-Auslastung
- Maximaler Durchsatz des Systems (kleine Abarbeitungszeit)
- Minimale Scheduling Aufwand
- Tolerierbare Antwortzeit (Turaround-Time)

Diese Aspekte werden in individuellen Scheduling-Algorithmen umgesetzt, die Aufgrund verschiedener Prozess-Messgrössen entscheiden, ob und wann ein Prozess die Rechenzeit abgeben muss. Die effektive Laufzeit eines Prozesses bis zum nächsten Context-Wechsel wird (dynamisch) beeinfluss durch:

- Laufzeitverhalten des Prozesses (I/O- oder CPU-lastig)
- Aktuelle (relative) Wichtigkeit des Prozesses
- Zulässigkeit einer Unterbrechung (preemtion)
- Ressourcenbedarf und effektiver –verbrauch des Prozesses

6.7.1.1. Scheduling-Strategien

Die Theorie definiert grundsätzlich zwei verschiedene Ansätze, wie einem Prozess die Rechenzeit entzogen werden kann:

Nonpreemtives Scheduling

Einem Prozess kann die CPU nicht entzogen werden. Diese wird nur freiwillig abgegeben bspw. bei einem I/O-Request oder nach der Terminierung

Preemtives Scheduling

Einem Prozess kann die CPU jeder Zeit zu Gunsten eines anderen Prozesses entzogen werden bspw. durch einen höher prioren Prozess oder nach Ablauf der Zeitscheibe.

6.7.1.2. Scheduling-Algorithmen

Die nachfolgende Ausführung über 3 bekannte Scheduling-Algorithmen hat nicht den Anspruch einer vertieften Beschreibung dieser Verfahren. Ziel ist es hier, dem Leser die Begriffe wieder ins Bewusstsein zu rufen für eine spätere Zuordnung.

FIFO-Scheduling

Diese Verfahren nimmt keine Rücksicht auf Wichtigkeit oder Laufzeitverhalten von Prozessen. Ein neu erzeugter Prozess wird in der Ready-Queue eingeordnet und muss mit der Ausführung warten, bis alle vor ihm liegenden Prozesse abgearbeitet sind. Die Prozessorzeit wird hier nach dem nonpreemtive-Ansatz nur freiwillig abgegeben. Falls ein Prozess aus einem blocked-in den ready-Zustand gelangt wird er wieder am Ende der Queue eingereiht (FCFS; first come first served)

Round Robin-Scheduling

Das Round-Robin-Verfahren kann vom letztgenannten FIFO-Verfahren abgeleitet werden. Die Prozesse werden der Reihe nach in die Ready-Queue abgelegt und wie geordnet auch abgearbeitet (FI-

FO). Der entscheidende Unterschied zum Round-Robin liegt darin, dass die maximale Ausführungszeit eines Prozesses durch die Timeslice (Zeitscheibe, Quantum) begrenzt ist. Einem Prozess der diese Grenze erreicht, wird die CPU auf jeden Fall entzogen. Die Systemcharakteristik hängt hier von der Grösse dieser Timeslice ab – wird sie gross gewählt verhält sich das System wie FIFO_Scheduled, wird sie (zu) klein gewählt wird das System durch Context-Wechsel überladen.

Priority-Scheduling

Das Priority-Verfahren ordnet die Ready-Queue nicht nach der Ankunftszeit eines Prozesses sondern nach deren Priorität. Prozesses erhalten unter Berücksichtigung der oben genannten Prozess-Messgrössen (bspw. Laufzeitverhalten) Prioritäten zwischen 0 bis 31. Diese bestimm die Reihenfolge in der Queue und den Zeitpunkt der Abarbeitung. Das Verhalten während der Laufzeit des Prozesses ist preemtive oder nonpreemtive:

- **Preemtive**: Erscheint ein höher priorer Thread wird dem laufenden Thread die CPU entzogen.
- **Nonpreemtive**: Ist die ready-Queue nach Prioritäten geordnet, verhält sich dieses Verfahren wie FIFO-Scheduled.

Das letztgenannte birgt das Problem des Starving (Verhungern von Prozessen mit tiefer Priorität). Lösung bietet hier die Erweiterung in Form von „Priority-Feedback-Scheduling“ bei dem die Priorität wartender Prozesse laufend inkrementiert wird oder Prozessen mit Laufzeit die Priorität dekrementiert wird.

6.8. Prozessverwaltung Windows

Das Scheduling unter Windows 2000/XP erfolgt auf der Ebene von Threads und nicht auf Prozessebene. Windows implementiert ein „priority-driven – preemtive scheduling”-System wo jener lauffähige Thread mit der höchsten Basispriorität die Prozessorzeit erhält. Wird ein Thread selektiert, läuft er maximal den festgelegten Timeslice (Quantum) ab und wird von einem gleichen oder höher prioren Thread abgelöst. Windows 2000/XP implementiert mit diesen Mechanismen der klassische Round-Robin-Ansatz.

Windows definiert insgesamt 32 Prioritätsstufen (0...31) wobei 0 die tiefste- und 31 die höchste Priorität repräsentiert. Die Priorität eines Threads setzt sich zusammen aus:

- der Priority Class seines Prozesses und
- dem Priority Level des Threads im Kontext dieses Prozesses

Priority Class und Priority Level werden kombiniert um die Base Prority des Threads zu bestimmen. Diese Base Priority wird verwendet um die Scheduling Entscheidungen zu treffen.

6.8.1. Priority Class

Unter Windows 2000/XP gehört jeder Prozess zu einer der folgenden Priority Classes:

Tabelle 8 Priority Class

Prioritätsklassen
IDLE_PRIORITY_CLASS
BELOW_NORMAL_PRIORITY_CLASS
NORMAL_PRIORITY_CLASS
ABOVE_NORMAL_PRIORITY_CLASS
HIGH_PRIORITY_CLASS
REALTIME_PRIORITY_CLASS

Per Default ist die Priority Class eines Prozesses mit NORMAL_PRIORITY_CLASS definiert. Mit der SetPriorityClass-Methode kann die die Priority Class eines Prozesses verändert werden. Die GetPriorityClass-Methode gibt die aktuelle Priority Class zurück.

SetPriorityClass Priority Class setzen

```
BOOL WINAPI SetPriorityClass(
   HANDLE hProcess,
   DWORD dwPriorityClass
);
```

Listing 2 SetPriorityClass

Attribut	Beschreibung
dwPriorityClass	Siehe Tabelle 8 Priority Class

GetPriorityClass Priority Class abfragen

```
DWORD WINAPI GetPriorityClass(
   HANDLE hProcess
);
```

Listing 3 GetPriorityClass

6.8.2. Priority Level

Die nachfolgende Liste zeigt die möglichen Priority Levels innerhalb der Priority Classes der Prozesse:

Tabelle 9 Priority Level

Prioritätslevel
THREAD_PRIORITY_IDLE
THREAD_PRIORITY_LOWEST
THREAD_PRIORITY_BELOW_NORMAL
THREAD_PRIORITY_NORMAL
THREAD_PRIORITY_ABOVE_NORMAL
THREAD_PRIORITY_HIGHEST
THREAD_PRIORITY_TIME_CRITICAL

Threads werden per Default mit dem Priority Level THREAD_PRIORITY_NORMAL erzeugt. Das bedeutet, dass die Thread Priorität der Prozess Priorität entspricht. Mit der SetThreadPriority-Methode kann nach der Erzeugung der Threads die relative Priorität eines Threads innerhalb der Threads dieses Prozesses verändert werden. Den aktuelle Priority Level gibt die GetThreadPriority-Methode zurück.

`SetThreadPriority` Priority Level setzen

```
BOOL WINAPI SetThreadPriority(
  HANDLE hThread,
  int nPriority
);
```

Listing 4 SetThreadPriority

Attribut	Beschreibung
`nPriority`	Siehe Tabelle 9 Priority Level

`GetThreadPriority` Priority Level abfragen

```
DWORD WINAPI GetPriorityClass(
  HANDLE hProcess
);
```

Listing 5 GetThreadPriority

6.8.3. Base Priority

Die Base Priority eines Threads ist eine Kombination aus Priority Class und Priority Level. Sie bestimmt ob ein Thread im Vergleich zu einem anderen Thread den Vorzug erhält oder nicht. Die Werte können aus der folgenden Auflistung entnommen werde (Auszug aus Gesamtliste)

Tabelle 10 Auszug aus der Thread-Priority Tabelle

#	Process Priority Class	Thread Priority Level
1	IDLE_PRIORITY_CLASS	THREAD_PRIORITY_IDLE
1	BELOW_NORMAL_PRIORITY_CLASS	THREAD_PRIORITY_IDLE
1	NORMAL_PRIORITY_CLASS	THREAD_PRIORITY_IDLE
1	ABOVE_NORMAL_PRIORITY_CLASS	THREAD_PRIORITY_IDLE
1	HIGH_PRIORITY_CLASS	THREAD_PRIORITY_IDLE
2	IDLE_PRIORITY_CLASS	THREAD_PRIORITY_LOWEST
3	IDLE_PRIORITY_CLASS	THREAD_PRIORITY_BELOW_NORMAL
4	IDLE_PRIORITY_CLASS	THREAD_PRIORITY_NORMAL
4	BELOW_NORMAL_PRIORITY_CLASS	THREAD_PRIORITY_LOWEST
5	IDLE_PRIORITY_CLASS	THREAD_PRIORITY_ABOVE_NORMAL
5	BELOW_NORMAL_PRIORITY_CLASS	THREAD_PRIORITY_BELOW_NORMAL
5	Background NORMAL_PRIORITY_CLASS	THREAD_PRIORITY_LOWEST

[…]

20	REALTIME_PRIORITY_CLASS	-4
21	REALTIME_PRIORITY_CLASS	-3
22	REALTIME_PRIORITY_CLASS	THREAD_PRIORITY_LOWEST
23	REALTIME_PRIORITY_CLASS	THREAD_PRIORITY_BELOW_NORMAL
24	REALTIME_PRIORITY_CLASS	THREAD_PRIORITY_NORMAL
25	REALTIME_PRIORITY_CLASS	THREAD_PRIORITY_ABOVE_NORMAL
26	REALTIME_PRIORITY_CLASS	THREAD_PRIORITY_HIGHEST
27	REALTIME_PRIORITY_CLASS	3
28	REALTIME_PRIORITY_CLASS	4
29	REALTIME_PRIORITY_CLASS	5
30	REALTIME_PRIORITY_CLASS	6
31	REALTIME_PRIORITY_CLASS	THREAD_PRIORITY_TIME_CRITICAL

Die Gesamtliste ist unter [MSDNSCHED]einsehbar.

Weiterführende Informationen:

- MSDN, Scheduling Priorities: [MSDNSCHED]

6.8.4. Priority Boosts

Jeder Thread besitzt eine dynamische Priorität, die Base-Priority. Sie wird vom Scheduler benutzt um Entscheidungen zu treffen. Das System kann diese Base-Priority dynamisch verändern um die Optimierungsprinzipien wie Fairness oder maximaler Durchsatz zu realisieren. Es findet keine dynamische Anpassungen im Bereich von Priority Level 16...31 statt.

6.8.5. Prozesse erzeugen

Die Erzeugung eines Prozesses unter Windows 2000/XP erfolgt mit der `CreateProcess`-Methode. Sie besitzt zahlreiche Parameter, von denen hier nicht alle dokumentiert werden.

`CreateProcess`	Erstellt und startet einen Prozess

Beispiel; Erzeugen und starten eines Prozesses mit eigener Konsole

```
STARTUPINFO si;
ZeroMemory(&si, sizeof(STARTUPINFO));
si.cb = sizeof(STARTUPINFO);
PROCESS_INFORMATION pi;
BOOL fCreated = CreateProcess(_T("C:\\foo.exe"),
                              NULL,
                              NULL,
                              NULL,
                              FALSE,
                              CREATE_NEW_CONSOLE,
                              NULL,
                              _T("C:\\ ").
                              &si,
                              &pi);
HANDLE hProcess = pi.hProcess; //Process-Handle
```

Listing 6 CreateProcess

Es werden nur diejenigen Attribute/Werte beschrieben, die einen Einfluss auf die Skalierung haben können.

Tabelle 11 Prozessattribut dwCreationFlag

Attribut	Beschreibung
`dwCreationFlag`	Steuert die Prioritätsklasse und die Erzeugung des Prozesses. Kann diverse Werte annehmen.

Tabelle 12 Werte von dwCreationFlag

Mögliche Werte von dwCreationFlag	Beschreibung
`CREATE_NEW_CONSOLE`	Prozess erhält eine eigene Konsole, erbt die Konsole des übergeordneten Prozesses nicht
IDLE_PRIORITY_CLASS	Threads laufen nur falls System im Leerlauf
`BELOW_NORMAL_PRIORITY_CLASS`	Threads laufen auf Prioritätsstufe zwischen `IDLE_PRIORITY_CLASS` und `NORMAL_PRIORITY_CLASS`
`NORMAL_PRIORITY_CLASS`	Thread ohne bestimmte Anforderungen an den Scheduling
`ABOVE_NORMAL_PRIORITY_CLASS`	Thread laufen auf Prioritätsstufe zwischen `NORMAL_PRIORITY_CLASS` und `HIGH_PRIORITY_CLASS`
`HIGH_PRIORITY_CLASS`	Thread für zeitkritische Aufgaben innerhalb einer minimalen Zeitverzögerung
`REALTIME_PRIORITY_CLASS`	Threads mit höchst möglicher Priorität (mit Vorsicht anzuwenden!)

6.8.6. Threads erzeugen

Threads unter Windows 2000/XP können auf 4 verschiedene Arten erzeugt werden:

- Starten eines neuen Prozesses
- Aufrufen der Win32-API-Funktion CreateThread
- Aufrufen der Funktion _beginthread aus der C-Laufzeitbibliothek
- Aufrufen der Funktion _beginthreadex aus der C-Laufzeitbibliothek

Nachfolgend wird aufgezeigt, wie mit der `CreateThread`-Methode ein Thread erzeugt werden kann. Die Priorität der Threads wird über die `CreateProcess`-Methode bestimmt oder nachfolgend mit `SetThreadPriority` dynamisch geändert.

`CreateThread` Erstellt und startet einen Thread

Beispiel; Erzeugen und starten eines Threads mit eigener Konsole

```
long WINAPI ThreadEntry(LPARAM lparam)
{
   //  …
}
unsigned long nThreadID;
HANDLE hThread = CreateThread(NULL,
                                0,
                                (LPTHREAD_START_ROUTINE)ThreadEntry,
(                               void*)szHello,
                                0,
                                &nThreadID);
```

Listing 7 CreateThread

6.8.7. Affinität von Prozessen

Unter einer Prozess-Affinität versteht man die logische Zuordnung eines Prozesses zu einem Prozessor. Sie wird bspw. angewendet um auf SMP-Architekturen das CPU-Hopping (stetiger CPU-Wechsel eines Threads) oder das Cache-Trashing (stetiger Wechsel des Cache-Inhalts durch wechselnde Threads) zu verhindern.

6.8.8. Affinität unter Windows XP

Unter Windows 2000/XP wird per Default ein Prozess oder Thread irgendeinem verfügbaren Prozessor zugeordnet. Will man hier korrigierend eingreifen, kann auf Level Prozess oder Thread eine Prozess-Affinität realisiert werden. Auf einem Mehrprozessorsystem kann so erreicht werden, dass Prozesse oder Threads auf verschiedenen, zugewiesenen Prozessor(kernen) laufen können.

Die Affinitätsmaske ist eine DWORD-Variable für die gilt:

- Bit 0 (niederwertigste) entspricht erster CPU
- Bit 1 entspricht zweiter CPU

Die Affinitätsmaske unter Windows lässt sich für den ganzen Prozess oder einzelne Threads eines Prozesses definieren. Soll die Thread-Affinität gesetzt werden, kann folgende Methode verwendet werden:

`SetThreadAffinityMask` Setzt die Affinität eines Threads

Beispiel; Ausführen eines Thread auf der zweiten CPU erzwingen

```
#include <windows.h>
#include <tchar.h>

Int _tmain()
{
   // Pseudohandle für Thread ermitteln
   HANDLE hThread = GetCurrentThread();
   DWORD dwAffinity = 0x02; // Nur auf zweitem Prozessor ausführen!
   DWORD dwOldAffinity = SetThreadAffinityMask(hThread, dwAffinity);
   _tprintf(_T("Allte Affinitätsmaske: %x\n", dwOldAffinity);
   […]
   return 0;
}
```

Listing 8 SetThreadAffinityMask

Soll die Prozess-Affinität gesetzt werden, kann folgende Methode verwendet werden:

`SetProcessAffinityMask` Setzt die Affinität eines Prozesses

Beispiel; Ausführen eines Prozesses auf der zweiten CPU erzwingen

```
#include <windows.h>
#include <tchar.h>

Int _tmain()
{
   // Pseudohandle für Thread ermitteln
   HANDLE hProcess = GetCurrentProcess();
   DWORD dwAffinity = 0x02; // Nur auf zweitem Prozessor ausführen!
   BOOL fSetAffinity = SetProcessAffinityMask(hProcess, dwAffinity);
   […]
   return 0;
}
```

Listing 9 SetProcessAffinityMask

Die unbedingte Zuweisung eines Threads oder Prozesses auf einen bestimmten Prozessor kann vorteilhaft sein, kann aber auch zu unerwarteten Performanceeinbussen führen. Wenn bspw. ein Thread mit Prozessor-Affinität durch einen anderen Thread auf diesem Prozessor blockiert wird, kann er nicht auf andere Systemressourcen ausweichen.

Dieses Problem kann umgangen werden indem mit der Funktion `SetThreadIdealProcessor` nur ein bevorzugter Prozessor definiert wird. Ist der Thread auf „seinem“ Prozessor blockiert verhindert das System die Nutzung eines anderen Prozessors nicht.

`SetThreadIdealProcessor` Setzt den idealen Prozessor für einen Thread

Beispiel; Ausführen eines Threads auf einem bevorzugten Prozessor

```
#include <windows.h>
#include <tchar.h>

Int _tmain()
{
   // Pseudohandle für Thread ermitteln
   HANDLE hThread = GetCurrentThread();
   DWORD dwPrefferedProc = 0x02; // Prozessor 2 bevorzugen
   DWORD dwPrevious = SetThreadIdealProcessor(hThread, dwPrefferedProc);
   if( dwPrevious == -1 )
         ReportError();
   […]
}
```

Listing 10 SetThreadIdealProcessor

Weiterführende Informationen:

- Windows 2000 developers's guide (ISBN 3-8272-5702-6): [WIN2KDEV]

6.8.9. Skalierbarkeit durch Affinität

Die Skalierbarkeit unter Anwendung der Prozess-Affinität ist durchaus denkbar. Das zu erwartende Systemverhalten ist aber nicht immer offensichtlich. System-Threads die im „Verborgenen" laufen und deren Scheduling können zu unerwarteten Ergebnissen führen.

Weiterführende Informationen:

- TMurgent Technologies, White Paper Processor Affinity: [CPUAFFINITY]

6.9. Prozessverwaltung Java

Werden Java Threads nicht auf Betriebssystem-Threads abgebildet, übernimmt die JVM das Scheduling der Threads. Die Java Virtual Maschine Specification definiert nicht abschliessend, welche Scheduling-Methode anzuwenden ist. Folglich ist die Verteilung unter Verwendung von „Green Threads“ abhängig von der effektiven Implementierung der JVM.

Das am häufigsten umgesetzte Verfahren basiert auf der „priority-driven“-Entscheidung, ob ein Thread Rechenzeit erhält oder nicht. Hierbei besitzt jeder Thread eine Priorität aus einem festgelegten Wertebereich. Kommt ein höher priorisierter Thread, wird ihm die CPU zu Verfügung gestellt. Falls Threads gleicher Wichtigkeit aufeinandertreffen, wendet der Scheduler das „Round-Robin“-Verfahren an (Quantum).

Die Priorität eines Threads wird bei der Erzeugung des Threads vergeben (vererbt) und kann vom Programmierer bewusst verändert werden. Die Java-Laufzeitumgebung ändert aber die einmal gesetzte Priorität von Threads nicht selbständig (vergl. Priority-Levels unter Windows). Die Abstufung der Prioritäten umfasst:

Tabelle 13 Thread Prioritäten

Prioritätswert	Beschreibung
`public final static int MIN_PRIORITY = 1`	Minimalpriorität eines Threads
`public final static int NORM_PRIORITY = 5`	Standardpriorität eines Threads.
`public final static int MAX_PRIORITY = 10`	Maximalpriorität eines Threads

Die Priorität eines aktiven Threads kann abgefragt und innerhalb des oben genannten Bereiches nach belieben gesetzt werden:

Tabelle 14 Thread Prioritäten abfragen

Methode	Beschreibung
`public int getPriority()`	Aktuelle Priorität abfragen
`public void setPriority(int newPriority)`	Neue Priorität setzen

Mehr Informationen im Kapitel 8.1.1.

Java bietet auch Methoden, mit denen das Laufzeitverhalten bzw. der Scheduler beeinflusst werden kann (bspw.):

Tabelle 15 Java Scheduler

Methode	Beschreibung
public static void yield()	Vorschlag Threadwechsel
public final void join() public final void join(long millis)	(max.) warten bis Thread beendet ist
public static void sleep(long millis)	Thread pausieren lassen

Siehe dazu auch Kapitel 8.1.1.

6.10. Windows API

In Bezug auf die Windows Prozesse werden von der API folgende Methoden zu Verfügung gestellt:

Tabelle 16 Windows API zur Prozessverwaltung (Auszug)

Funktion	Beschreibung
`CreateProcess`	Erzeugt einen neuen Prozess und Thread mit der security identification des Aufrufers
`CreateProcessAsUser`	Erzeugt einen neuen Prozess und Thread mit einem spezifischen security token
`OpenProcess`	Gibt den Handle dieses Prozess-Objektes zurück
`ExitProcess`	Beendet Prozess mit notify aller eingebundenen DLLs
`TerminateProcess`	Beendet Prozess ohne notify aller eingebundenen DLLs
`GetProcessTimes`	Sammelt Zeitinformationen wie lange der Prozess im User-/Kernelmode gelaufen ist

In Bezug auf die Windows Threads werden von der API folgende Methoden zu Verfügung gestellt:

Tabelle 17 Windows API zur Thread Verwaltung (Auszug)

Funktion	Beschreibung
`CreateThread`	Erzeugt neuen Thread
`CreateRemoteThread`	Erzeugt einen neuen Thread in einem anderen Prozess
`OpenThread`	Öffnet einen Thread
`ExitThread`	Normales Beenden eines Thread
`TerminateThread`	Terminiert ein Thread
`GetThreadTimes`	Gibt Zeitinformationen eines Threads zurück
`GetCurrentProcess`	Gibt den Pseudo-Handle eines Threads zurück
`GetCurrentProcessID`	Gibt die ID des aktuellen Threads zurück
`GetThreadId`	Gibt die ID eines spezifischen Threads zurück
`Get/SetThreadContext`	Liefert oder ändert die CPU-Register eines Threads

6.11. Prozesse überwachen

Performance Counter für Prozesse

Windows stellt einige Leistungsindikatoren (Counter) zu Verfügung mit dem Prozesse überwacht werden können:

Tabelle 18 Windows Performance Counter für Prozesse (Auszug)

Counter	Beschreibung
Process: % Privileged Time	Prozentuale-Laufzeit der Threads eines Prozesses im Kernel-Mode in einem spezifischen Intervall
Process: % User Time	Prozentuale-Laufzeit der Threads eines Prozesses im User-Mode in einem spezifischen Intervall
Process: % Processed Time	Prozentuale-Laufzeit der Threads eines Prozesses in einem spezifischen Intervall (Privileged-Time + User-Time)
Process: % Elapsed Time	Verstrichene Zeit seit der Erzeugung des Prozesses
Process: % ID Process	Definiert die Prozess-ID (Achtung Wiederverwendung!)
Process: % Thread Count	Gibt die Anzahl Threads eines Processes zurück

Performance Counter für Threads

Windows stellt auch einige Leistungsindikatoren (Counter) zu Verfügung mit dem Threads überwacht werden können:

Tabelle 19 Windows Performance Counter für Threads

Counter	Beschreibung
Process: Priority Base	Gibt die aktuelle Base Priority des Prozesses zurück (Start-Priorität des Threads)
Thread: % Privileged Time	Prozentuale-Laufzeit der Threads im Kernel-Mode in einem spezifischen Intervall
Thread: % User Time	Prozentuale-Laufzeit der Threads im User-Mode in einem spezifischen Intervall
Thread: % Processor Time	Prozentuale-Laufzeit der Threads in einem spezifischen Intervall (Privileged-Time + User-Time)
Thread: % Context Switches/Sec	Anzahl Kontextwechsel per Sekunde des Systems
Thread: % Elapsed Time	Gibt die CPU-Zeit in Sekunden zurück die ein Thread insgesamt konsumiert hat
Thread: % ID Process	Definiert die Prozess-ID eines Threads (Achtung Wiederverwendung!)
Thread: % ID Thread	Definiert die Thread-ID eines Threads (Achtung Wiederverwendung!)
Thread: Priority Base	Gibt die aktuelle Base Priority des Threads zurück (Kann ungleich Start-Priorität des Threads sein)
Thread: Priority Current	Gibt die dynamische Priorität des Threads zurück

Thread: Thread State	Gibt den aktuellen Status/Zustand des Threads zurück (Wert 0...7)
Thread: Thread Wait Reason	Gibt den Grund für den Zustand WAIT zurück (Wert 0...19)

6.12. Profiling Prozesse

Das Profiling des Systems hat zum Ziel, die oben genannten Leistungsindikatoren von Prozessen und Threads sichtbar zu machen um das Systemverhalten analysieren zu können.

Für diese Tätigkeit gibt es zahlreiche Tools die mit mehr oder weniger Leistungsumfang Hinweise über folgende Grössen liefern können:

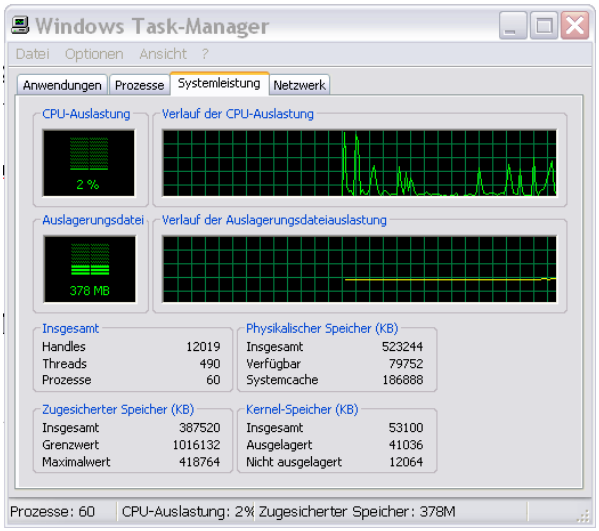
- Prozess-/Thread-ID
- Konsumierte CPU-Zeit Prozess/Thread
- Anzahl Threads pro Prozess
- Priorität von Threads (Basis)

Nachfolgend werden mehrere Beispiele solcher Instrumente grob vorgestellt ohne eine vertiefte Analyse deren Funktionalität durchzuführen.

6.12.1. Windows TaskManager

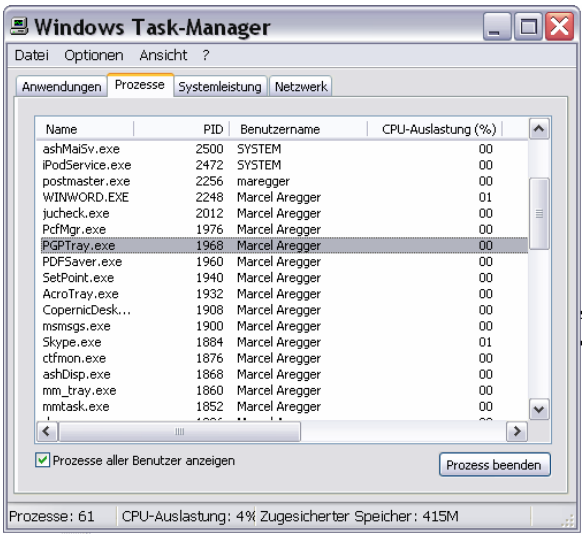
Der Windows Task Manager ist auf jedem Windows 2000/XP System verfügbar und bietet im Wesentlichen Informationen über Programme und Prozesse die auf dem System laufen. Er zeigt weiter einige Indikatoren in Bezug auf die aktuelle Systemleistung.

Performance



Das Performance Tab zeigt in dynamischer Form Werte der CPU-Performance wie bspw. CPU-Auslastung (insgesamt) oder der Verlauf dieser Auslastung.

Aktive Prozesse

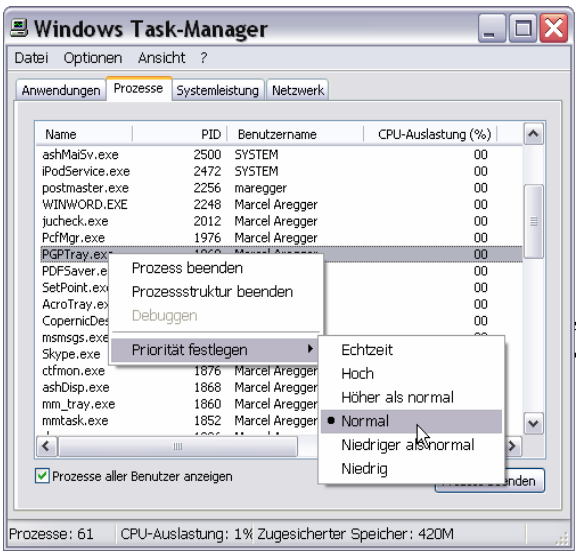


Das Process Tab zeigt die aktiven Prozesse des System. In dieser Spaltenübersicht können Prozesse selektiert, beendet oder nach belieben Leistungsindikatoren angezeigt werden.

Leistungsindikatoren

Der Taskmanager bietet Leistungsindikatoren gemäss Auflistung.

Priorität von Prozessen



Für jeden Prozess kann die Priority Class des Prozesses geändert werden (siehe 6.8.1)

Grundfunktionalität

Tabelle 20 Funktionalitäten Windows Task Manager

Windows Task Manager	Ja	Nein
Prozess-ID	✓	
Thread-ID		✓
Konsumierte CPU-Zeit Prozess	✓	
Konsumierte CPU-Zeit Thread		✓
Anzahl Threads pro Prozess	✓	
Prozess Affinität (SMP)	✓	

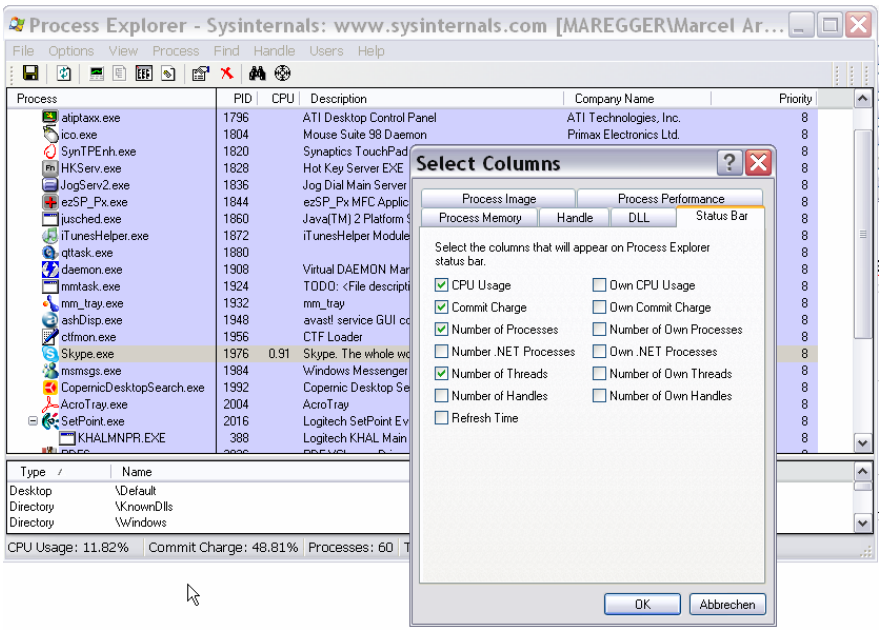
Weiterführende Informationen:

- Microsoft, Task Manager Overview: [TASKMANOV]

6.12.2. Process Explorer

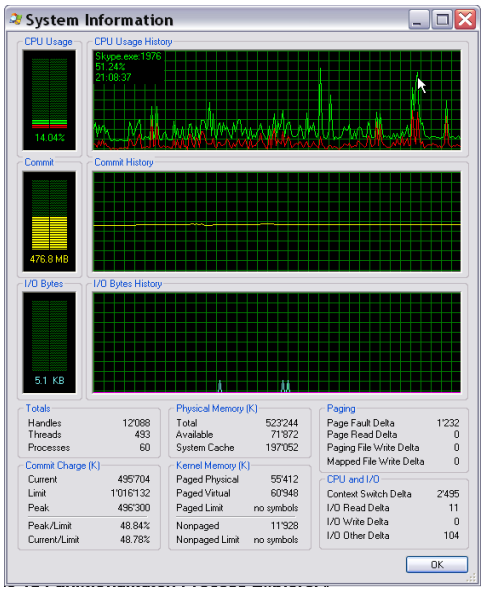
„Process Explorer“ eine Freeware von SysInternals ist eine Spezialisierung des Windows-Task Managers. Er erweitert das oben beschriebene Tool um zahlreiche Zusatzfunktionen und bietet im Wesentlichen mehr Informationen über die Threads eines spezifischen Prozesses.

Aktive Prozesse u. Leistungsindikatoren



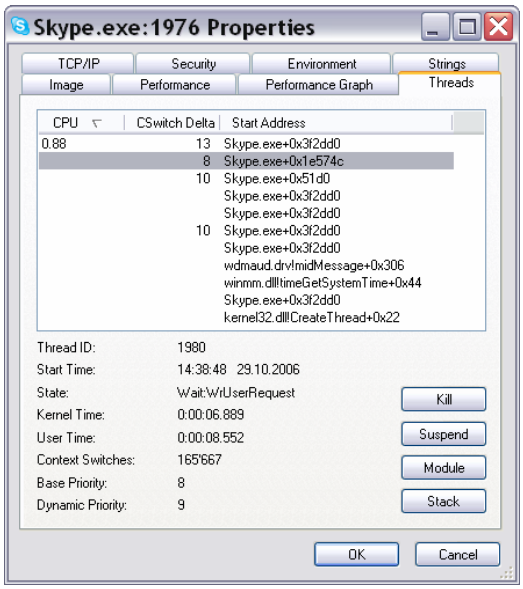
Symbole ergänzen die Übersicht der Prozesse wodurch diese besser identifiziert werden können. Es können zahlreiche Leistungsindikatoren in die Spalten eingefügt oder entfernt werden.

Performance-Übersicht



Der Performance Graph kann für alle oder einen spezifischen Prozess angezeigt werden. In der Gesamtansicht kann ein Peak selektiert werden wodurch die spezifische CPU-Usage angezeigt wird.

Thread-Übersicht



Im Tab Threads werden zahlreiche Informationen über die Threads eines Prozesses gezeigt.

Grundfunktionalität

Tabelle 21 Funktionalitäten Process Explorer

Windows Task Manager	Ja	Nein
Prozess-ID	✓	
Thread-ID	✓	
Konsumierte CPU-Zeit Prozess	✓	
Konsumierte CPU-Zeit Thread	✓	
Anzahl Threads pro Prozess	✓	
Prozess Affinität (SMP)	✓	

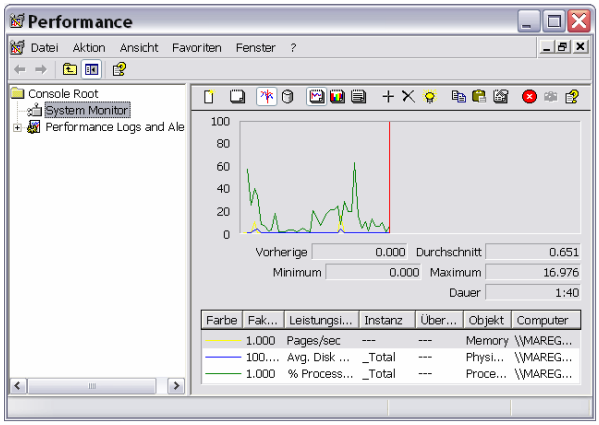
Weiterführende Informationen:

- Sysinternals Process Explorer Overview: [PROCEXP]

6.12.3. Performance Monitor

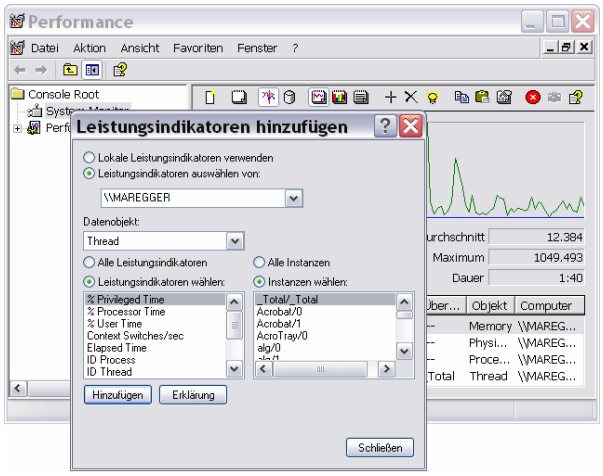
Der Performance Monitor ist ebenfalls auf Windows 2000/XP verfügbar. Mit diesem Systemmonitor kann die Leistung des lokalen Computers sowie anderer Computer im Netzwerk gemessen werden. Im Speziellen können mit dem Systemmonitor Leistungsdaten in Echtzeit gesammelt werden.

Leistungsobjekte



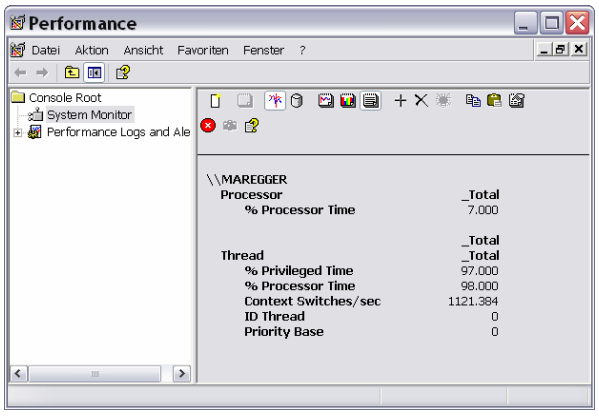
Die Überwachung erfolgt für mehrere frei wählbare Leistungsobjekte (bspw. Process, Thread, Processor)

Leistungsindikatoren

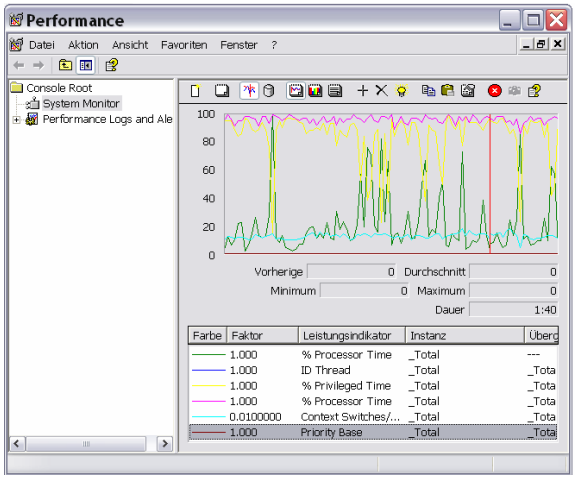


Für jedes Leistungsobjekt können zahlreiche Leistungsindikatoren (Counter) angezeigt werden (siehe 6.11)

Darstellung Resultate



Die Daten können in Form eines Grafen oder in Form eines Berichtes angezeigt werden.



Grundfunktionalität

Tabelle 22 Funktionalitäten Performance Monitor

Performance Monitor	Ja	Nein
Prozess-ID	✓	
Thread-ID	✓	
Konsumierte CPU-Zeit Prozess	✓	
Konsumierte CPU-Zeit Thread	✓	
Anzahl Threads pro Prozess	✓	
Prozess Affinität (SMP)	✓	

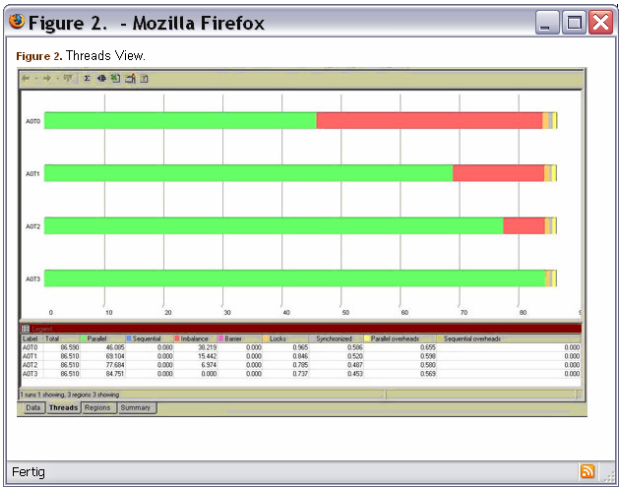
Weiterführende Informationen:

- ZDNet, System-Performance im Visier: Die besten Tools: [PERFTOOLS]

6.12.4. Intel Thread Profiler

Der Intel Thread Profiler 3.0 for Windows ist ein Performance Tuning Tool das speziell dafür entwickelt wurde, multithreaded Applikationen auf Basis von OpenMP auf Mehrprozessor-Systemen zu testen. Schwerpunkt liegt dabei auf dem Load Imbalancing und Synchronisation Impact.

Intel Thread Profiler



Load Imbalancing

Das Tool zeigt die Ausnutzung der logischen/physischen Cores eines Prozessors (hier Dual Core mit HT-Technologie und 4 Threads).

Synchronisation Impact

Das Tool ermöglicht auch eine spezifische Analyse in Bezug auf den (negativen) Einfluss der Synchronisation auf die Performance.

Grundfunktionalität

Intel Thread Profiler	Ja	Nein
Load Imbalancing	✓	
Synchronisation Impact	✓	

Das Tool ist grundsätzlich Kostenpflichtig (ca. CHF 450.-), kann aber als Trial Version für 30 Tage kostenlos getestet werden.

Weiterführende Informationen:

- Devx, Intel Threading Tools and OpenMP: [DEVXINTEL]

6.13. Zusammenfassung und Fazit

Einem Betriebssystem sind in Bezug auf Prozesse und Threads zahlreiche Aufgaben zugeordnet. Neben der Erzeugung und dem Zuweisen von Adressraum muss das Betriebssystem in einer Ablaufplanung permanent verfügbare CPU-Zeit auf die konkurrierenden Prozesse und Threads verteilen. Pro Zeiteinheit kann dabei nur ein Prozess auf einem Prozessor(Kern) ausgeführt werden. Für ein Singlecore-System (exkl. Intel-HT) führt das zu einer rein „logischen-Parallelität“ (pseudo-Parrallelität). Multicore Systeme hingegen können hier „echte“ Parallelität bieten und Prozesse gleichzeitig ablaufen lassen. Im Kontext der Skalierung von multithreaded Applikationen stellt sich somit die Frage, wie auf Level Betriebssystem die Verteilung von CPU-Zeit auf mehrere Prozesse/Threads optimiert bzw. beeinflusst werden kann.

Der Begriff Prozess ist eng verknüpft mit dem Begriff „Prozess-Kontext“, der den aktuellen Zustand eines Prozesses repräsentiert bzw. speichert. Ein Wechsel von einem laufenden auf einen lauffähigen Prozess bedeutet auch ein Kontextwechsel d.h. speichern des aktuellen Kontext und laden des Nachfolgenden. Dieser Vorgang ist zeitraubend und wird oftmals als „nicht produktiver Overhead“ bezeichnet. Threads sind „leichtgewichtige“ Prozesse die mit anderen Threads im gleichen Prozesskontext den Adressraum teilen (Peerthreads). Durch den „kleineren“ Thread-Prozesskontext sind Threads einfacher und schneller zu erzeugen, unterbrechen oder letztendlich zu entfernen. Der gemeinsame Adressraum bietet implizit eine einfachere Interprozess-Kommunikation.

Windows XP implementiert ein Kernel-Level-Thread-Model. In dieser 1:1-Zuordnung laufen im gleichen Prozesskontext mehrere Kernel-Level-Threads die vom Kernel direkt verwaltet werden. Der Kernel besitzt neben der Prozess-Tabelle auch eine Thread-Tabelle wodurch er in die Lage versetzt wird, Threads direkt zu verwalten bzw. auf verschiedene CPUs zu verteilen. Unter Windows sind „Fibers“ von Threads zu unterscheiden. Ein „Fiber“ kann als “Thread in Thread” bezeichnet werden und ist eine Kernel unabhängige User-Level Implementierung. Das Scheduling eines Fibers ist nonpreemtive d.h. die CPU muss selbständig an den Win32-Thread abgegeben werden.

Die Java-Virtual-Machine (JVM) bildet die Laufzeitumgebung eines Java-Threads. Durch ihre Implementierung wird festgelegt, ob ein Java-Thread direkt auf einen Thread des Betriebssystem abgebildet werden kann. Erfolgt die Abbildung direkt (1:1) ist die Voraussetzung für eine individuelle Verteilung auf mehrere CPU durch das OS gegeben.

Ob ein Prozess Rechenzeit zugeteilt bekommt oder nicht, hängt vom Scheduler des Betriebssystems ab. Unter Windows 2000/XP erfolgt diese Zuweisung auf der Ebene von Threads und nicht auf Prozessebene. Windows implementiert ein „priority-driven – preemtive scheduling”-System wo jener lauffähige Thread mit der höchsten Basispriorität die Prozessorzeit erhält. Die Basispriorität ist also die ausschlaggebende Grösse um den Zuspruch durch den Scheduler zu erhalten. Die Basis-Priorität eines Threads ist eine Kombination aus der Priority-Class seines Prozesses und des eigenen Priority-Levels.

Die Priorität eines Prozesses kann bei der Erzeugung mitgegeben oder nachträglich mit der Methode `SetPriorityClass` geändert werden. Threads „erben“ per Default die Priorität ihres Prozesses die aber nachfolgend auch beeinflusst werden kann (`SetThreadPriority`). Der Begriff „Priority Boosts“ beschreibt die dynamische Anpassung der Basis-Priorität von Threads durch das Betriebssystem. Im Priority-Level 1...15. hat das Betriebssystem so die Möglichkeit nach spezifischen Optimierungsprinzipien das Systemverhalten zu steuern.

Java-Threads besitzen ebenfalls eine Priorität die mit der Methode `setPriority(int newPriority)` im Bereich von [1...5...10] (Min_ ...Norm_ ... Max_) geändert werden kann. Sie wird bei der Verwendung von „Green-Threads“ (simulierte Threads innerhalb des VM-Prozess) von der VM dazu verwendet Scheduling-Enscheidunge zu treffen.

Soll ein Prozess einem Prozessor zugeordnet werden spricht man von Prozess-Affinität. Windows ordnet per Default Prozessor-Ressourcen willkürlich zu. Mit der Definition einer Affinitätsmaske kann aber auf Level Prozess und Thread die entsprechende Affinität gesetzt werden (`SetProcessAffinityMask` | `SetThreadAffinityMask`). Diese unbedingte Zuweisung kann sich aber sehr nachteilig auswirken weil andere Prozessoren für diese Prozesse oder Threads gesperrt sind. Die Verwendung von `SetThreadIdealProcessor` setzt einen präferenzierten Prozessor und lässt ein Ausweichen auf andere Ressourcen zu.

Prozesse und Threads könne im OS über sogenannte Leistungsindikatoren (Performance-Counter) überwacht werden. In einem Profiling kann mit Hilfe dieser Indikatoren das Systemverhalten mit Prozessen und Threads sichtbar gemacht und interpretiert werden. Das Windows OS bietet mit Tools wie “Task manager” oder “Performance manager” eigene Instrumente zur Messung solcher Indikatoren.

6.14. Auswirkungen auf die Aufgabenstellung

Nicht alle nachfolgend aufgeführten Einflussbereiche die aus der Detailanalyse des Betriebssystems resultieren können gleichermassen für die weitere Analyse oder Implementierung genutzt werden. Ziel ist es, Aspekte mit direktem Einfluss auf die Aufgabestellung in die nachfolgenden Projektphasen zu übernehmen bzw. einzuarbeiten.

Tabelle 23 Aspekte mit direktem Einfluss auf die Arbeit

Aspekt	Beschreibung
Designprinzip	Die Prozesstheorie hat gezeigt, dass die Anwendung „leichtgewichtiger“ Threads zahlreiche Vorteile mit sich bringt. Aus dem gemeinsamen Adressraum resultieren wenig Overhead beim Kontextwechsel und eine vereinfachte Interprozesskommunikation. Trotzt Vorteilen sind Grössen wie Verwaltungsaufwand, Ressourcenbedarf oder Schedulingverhalten mit Threads zu berücksichtigen.
WIN-32 Thread	Kernel-Level-Threads von Windows werden vom Kernel verwaltet und können somit in einer Mehrprozessor-Architektur auch verteilt werden. Mit einer entsprechenden VM-Implementierung werden Java-Threads auf solche Native-Threads abgebildet.

Tabelle 24 Aspekte mit indirektem Einfluss auf die Arbeit

Technologie	Beschreibung
Scheduling	Der Windows Scheduler implentiert auf Level Threads ein “priority-driven” scheduling. Die entscheidende Grösse ist hierbei die Basispriorität des jeweiligen Kernel-Threads. Soll das Schedulingverhalten beeinflusst werden, muss über die WinAPI die Priorität von Prozessen oder Threads beeinflusst werden. Da mit einer Java-Anwendung die WinAPI ausser Reichweite ist, kann die Priorität auf Level OS nicht direkt beeinflusst werden. Im Verlauf soll aber untersucht werden, wie die VM Java-Thread-Prioritäten (1...5...10) auf Prioritäten des OS abbildet. Weiter besteht die Möglichkeit mit diversen System(Tools) die Priorität aktiver Prozesse/Threads „manuell“ zu beeinflussen. Das Systemverhalten kann mit „manuellem“ Ändern der Priorität untersucht werden.
Affinität	Windows erlaubt die Prozess- wie auch die Thread-Affinität, bei der ein Prozess oder einzelne Threads einem Prozessor zugeordnet werden können. Über die WinAPI kann eine Zuweisung von Prozess oder Thread durchgeführt werden. Auf Level VM oder Java-Applikation kann die Affinität nicht direkt beeinflusst werden. Es besteht aber die Möglichkeit die Folgen einer Zuweisung mit (System)Tools zu provozieren bzw. untersuchen.

7. Applikationen

7.1. Allgemeine Eigenschaften paralleler Programme

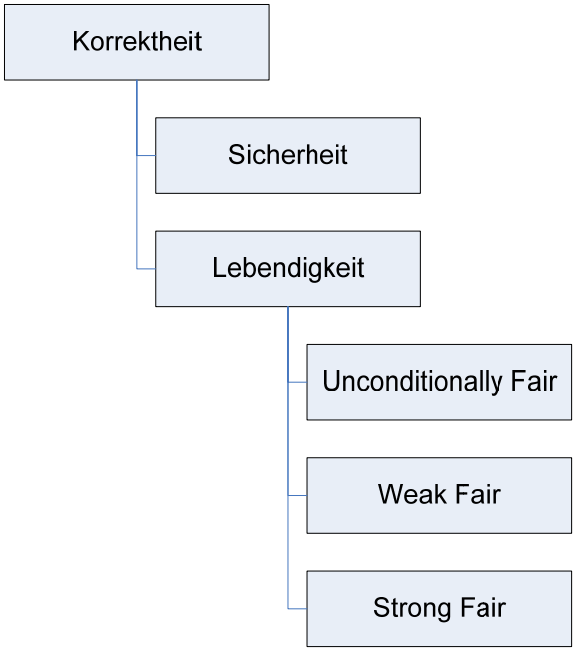


Abbildung 23 Korrektheit von Programmen

Zur Korrektheit eines Programms gehört einerseits die Sicherheit und andererseits die Lebendigkeit. Sicherheit bedeutet, dass keine Interferenzen in kritischen Bereichen sowie auch keine Verklemmungen (Deadlock) auftreten. Die Lebendigkeit lässt sich allgemein damit umschreiben, dass weder Livelocks noch so genannte Starvation auftreten.

Ein Deadlock entsteht in einer Situation in der mehrere Prozesse auf die Freigabe einer Ressource warten die durch einen anderen Prozess blockiert ist. Sind die Ressourcen zirkulär angeordnet so entsteht eine Wartesituation aus der die involvierten Prozesse selber nicht mehr herauskommen. Siehe dazu auch [DEADLOCK].

Ein Livelock führt ebenfalls dazu, dass die Prozesse nicht mehr weiter arbeiten können. Allerdings warten sie dabei nicht wie beim Deadlock auf eine Ressource sondern verändern ihren Status dabei um weiter arbeiten zu können. Tun sie das so, dass sie bei jeder Veränderung weiterhin blockiert werden nennt man dies Livelock Siehe dazu auch [LIVELOCK].

Unter Starvation versteht man den Zustand eines Prozesses in dem er auf Ressourcen wartet und diese nie bekommt. Dies kann beispielsweise passieren, wenn ein Worker-Pool als FILO Queue (First-In-Last-Out) realisiert wird und immer mehr Worker in den Pool gelegt werden als herausgenommen werden. Dann bleiben einige Worker im Pool liegen und werden ihre Aufgabe nie erledigen können. Siehe dazu auch [STARVATION].

Zur Lebendigkeit gehören weitere Attribute welche mit den Attributen Unconditionally Fair, Weak Fair und Strong Fair bezeichnet werden. Wie die Bezeichnung schon aussagt geht es dabei um die Fairness während der Programmausführung. Ist ein Programm „Unconditionally Fair“ so sorgt das Programm nicht manuell für Fairness aber kann beispielsweise durch den Scheduler zur Abgabe der Rechenzeit gezwungen werden. Arbeitet der Scheduler nach dem „Weak Fair“ Prinzip (was bei den meisten der Fall ist) so kann für den Prozess nicht garantiert werden, dass er genau an der gewünschten Stelle unterbrochen wird. Bei „Strong Fair“ Scheduling würde dies sichergestellt.

7.2. Technologien zur Parallelisierung

In diesem Kapitel werden allgemeine Technologien zur parallelen Verarbeitung kurz beschrieben.

7.2.1. Prozesse

Die einfachste Möglichkeit eine Aufgabe parallel zu verarbeiten ist die Aufteilung auf mehrere Prozesse. Diese Methode zieht aber einige Nachteile mit sich. In heutigen Multitasking-Betriebssystemen laufen verschiedenste Prozesse ab. Das Betriebssystem sorgt dafür, dass alle Prozesse Rechenzeit bekommen. Die Wechsel zwischen den Prozessen nennt man Kontextwechsel (engl. Context Switch). Siehe dazu auch [CONTEXTSW]. Solche Prozesswechsel sind aber aufwändig und kosten natürlich Zeit. Für Kontextwechsel ist generell der Scheduler des Betriebssystems zuständig. Dieser kann natürlich versuchen die Anzahl Kontextwechsel zu minimieren indem die Zeitscheiben für die einzelnen Prozesse vergrössert werden. Dies würde bedeuten, dass ein Prozess länger Rechenzeit bekommt. Andererseits bedeutet dies, dass ein wartender Prozess länger darauf warten muss um wieder Rechenzeit zu bekommen. Dies ist insbesondere bei zeitkritischen Anwendungen wie Multimedia kritisch.

Ein weiteres Problem bei der Aufteilung in mehrere Prozesse ist die Kommunikation und Synchronisation. Die meisten Betriebssysteme stellen so genannte Inter-Prozess-Kommunikation (IPC, Inter Process Communication) zur Verfügung. Diese ist aber abhängig vom Betriebssystem und teilweise sehr aufwändig.

Hier muss insbesondere darauf geachtet werden, dass der Geschwindigkeitsgewinn aus der parallelen Verarbeitung nicht durch Synchronisierung, Prozess-Erzeugung, Prozess-Terminierung und Inter-Prozess-Kommunikation zunichte gemacht wird.

7.2.2. Threads

Threads werden häufig auch als leichtgewichtige Prozesse oder LWP (englisch für Lightwight Processes) bezeichnet. Mit Threads wird versucht den Nachteilen der Multi-Prozess-Programmierung entgegenzutreten. Die Erzeugung eines neuen Threads ist weniger aufwändig da für einen Thread nicht ein gesamter Prozess-Kontext erstellt werden muss. Ein Thread läuft innerhalb des erzeugenden Prozess-Kontextes. Aus demselben Grund sind auch Kontextwechsel zwischen Threads weit weniger aufwändig. Wird ein Thread beendet so muss natürlich auch nur der Thread-Kontext entfernt werden und nicht gleich der ganze Prozess-Kontext.

Da Threads ihren Adressraum mit Prozessen teilen vereinfacht sich auch die Kommunikation zwischen ihnen (Inter-Thread-Kommunikation). Hier muss nicht auf Betriebssystem-Funktionen zur Inter-Prozess-Kommunikation zurückgegriffen werden. Die Kommunikation kann über gemeinsame Variabeln im selben Adressraum geschehen.

Andererseits stellt uns der gemeinsame Zugriff auf Prozessressourcen natürlich wieder vor weitere Probleme. Der Zugriff auf gemeinsame Speicherbereiche muss synchronisiert werden um sicherzustellen, dass die Daten konsistent bleiben. Einer der Schlüsselpunkte zur effizienten Thread-Programmierung liegt darin diesen Synchronisationsaufwand im Verhältnis zur parallelen Verarbeitung möglichst gering zu halten.

7.2.3. Verteilung

Eine weitere Möglichkeit Aufgaben parallel abzuarbeiten besteht in der Verteilung auf mehrere Systeme. Dies entspricht der schon in Kapitel 4 angesprochenen horizontalen Skalierung und soll hier nicht näher betrachtet werden.

Allerdings sind die Grenzen hier fliessend. Beispielsweise spricht man selbst bei grossen Rechnersystemen in einem Gehäuse von Knoten (engl. Nodes) wie bei einem Cluster wenn die Hardware intern entsprechend aufgebaut ist. Verwaltet ein System den Speicher nach dem NUMA/ccNUMA Prinzip (siehe dazu Kapitel 5.2.1) so verhält es sich im Grunde wie ein sehr schnell gekoppeltes verteiltes System.

7.3. Frameworks, Standards und Libraries

7.3.1. POSIX-Threads

Wenn man von POSIX Threads spricht, so ist allgemein die Plattformunabhängige Definition der POSIX Thread Schnittstelle gemeint. Diese spezifiziert „nur“ die Schnittstelle für die Thread-Behandlung. Nicht aber deren Implementierung. Dies hat insbesondere den Vorteil, dass POSIX Threads auf vielen unterschiedlichen Betriebssystemen verfügbar sind. Diese Tatsache erlaubt dem Programmierer Threads einzusetzen ohne auf Plattformspezifische APIs zurückgreifen zu müssen. Da die Thread-Behandlung innerhalb eines komplexen Programms sehr eng mit dem Programmcode verknüpft ist würde eine Plattformabhängige Programmierung in den meisten Anwendungen sehr viel Aufwand verursachen. POSIX Threads abstrahieren diese Komplexität. Natürlich muss die POSIX Thread Library Plattformabhängig implementiert werden. Diese Adaptierung muss aber nur einmal gemacht werden. Im Optimalfall bietet das Betriebssystem direkte Unterstützung für POSIX Threads. In diesem Fall müsste keine Abbildung der Thread-Behandlung auf Betriebssystemfunktionen durch die POSIX-Thread Bibliothek stattfinden.

Nachfolgend werden einige der wichtigsten Funktionen der POSIX Thread Schnittstele kurz erläutert. Die Beispiele stammen dabei aus dem sehr guten POSIX Thread Tutorial von Mark Hays (siehe auch [POSIXTUTOR]) Hierbei handelt es sich nicht um eine abschliessende Dokumentation sondern um einen Überblick über die POSIX Schnittstelle.

Um einen Thread zu erzeugen wird die folgende Funktion verwendet:

```
pthread_create(&tid, &attr, function, &parameters)
```

Listing 11 POSIX Thread erzeugen

Tabelle 25 pthread_create() Parameter

Argument	Beschreibung
`tid`	Pointer auf eine Datenstruktur vom Typ pthread_t. Wird allgemein als Thread ID (TID) bezeichnet und entspricht dem Thread-Handle um den Thread später kontrollieren zu können.
`attr`	Thread-Attribute. Hierbei handelt es sich um Attribute welche die Eigenschaften des Threads direct beeinflussen.
`function`	Name der auszuführenden Funktion. Der Thread wird diese Funktion nach der Erzeugung aufrufen.
`parameters`	Pointer auf eine Datenstruktur, die als Parameter an die Funktion übergeben wird.

Nach dem Start des Threads wird das Hauptprogramm unter Umständen noch weitere Aufgaben erledigen. Sehr häufig wird es dann aber auf die Beendigung des/der Threads warten. Dies kann mit folgender Funktion getan werden:

```
pthread_join(tid, &return)
```

Listing 12 Warten auf Thread-Ende

Tabelle 26 pthread_join() Parameter

Argument	Beschreibung
`tid`	Pointer auf eine Datenstruktur von Typ pthread_t. Dies entspricht dem Thread Handle wie bereits bei pthread_create().
`return`	Pointer an dem die Rückgabewerte der Thread-Funktion abgelegt werden sollen (void Pointer).

Durch die Verwendung mehrerer Threads entsteht natürlich das Problem der gleichzeitigen Modifikation von globalen, gemeinsamen Daten. Insbesondere erleichtern Threads ja gerade den Zugriff auf gemeinsame Daten. Als Beispiel sei die Situation genannt wo zwei Threads eine Variable auslesen, 1 addieren und wieder speichern. Täten sie das streng sequenziell, dann würde die Variable am Ende der Modifikation um 2 grösser sein als am Anfang. Lesen aber beide (quasi-) gleichzeitig den aktuellen Wert aus und inkrementieren ihn unabhängig voneinander, dann „gewinnt" schlussendlich derjenige, der die Variable als letztes speichert/überschreibt. Um dies zu verhindern bietet die POSIX Thread Schnittstelle so genannte Mutexe an. Mutex steht für „Mutual Exclusion" und bezeichnet einen Mechanismus, bei dem nur ein Thread in einen kritischen Bereich eintreten kann. Befindet sich bereits ein Thread in diesem Bereich, so müssen weitere Threads warten bis dieser den geschützten Bereich verlassen hat.

Mutex mit POSIX Threads:

```
pthread_mutex_t lock;
// code
pthread_mutex_lock(&lock);
// code
pthread_mutex_unlock(&lock);
```

Listing 13 POSIX Mutex

Mit diesem Code ist sichergestellt, dass sic him Bereich zwischen `pthread_mutex_lock()` und `pthread_mutex_unlock()` nu rein einziger Thread aufhalten kann. Durch Mutexe lässt sich der Zugriff auf gemeinsame Ressourcen kontrollieren. Dabei können beliebig vielen Mutexe erstellt werden. Allerdings ist darauf zu achten, dass jeder Datenzugriff synchronisiert wird. Ignoriert ein Thread den Mutex (den er ja manuell verwenden muss) so kann dies wieder zu denselben Problemen führen. In der Objektorientierten Programmierung hilft hierbei das Konzept der Datenkapselung indem der direkte Zugriff auf die Daten verhindert wird und über dafür bestimmte get und set Methoden realisiert wird. Diese können dann die Synchronisation an zentraler Stelle übernehmen. Ob eine Klasse intern synchronisiert ist und somit konsistente Daten garantiert wird oft mit dem Attribut „Thread safe" gekennzeichnet. Eine Klasse, die Thread safe ist muss den parallelen Zugriff regeln und in jedem Fall gültige Daten garantieren.

Hierbei ist insbesondere auf die Gefahr von Deadlocks zu achten (siehe dazu auch Kapitel 7.1).

Angenommen Thread A besitzt den Mutex-Lock für Datenfeld 1 und Thread B besitzt den Mutex-Lock für Datenfel 2.

Nun versucht Thread B den Mutex-Lock für Datenfeld 1 auch noch zu bekommen ohne den Mutex-Lock für Datenfeld 2 abzugeben. Jetzt muss Thread B auf die Lock-Freigabe von Datenfeld 1 warten.

Wenn jetzt Thread A aus irgendeinem Grund den Lock für Datenfeld 1 nicht freigibt und/oder seinerseits versucht den Lock für Datenfeld 2 zu bekommen, dann befinden sich beide Threads in einem Deadlock-Zustand aus dem sie nicht mehr herauskommen. Es besteht eine Zirkuläre Abhängigkeit.

Warten auf Bedingungen:

```
pthread_mutex_locak(&mutex);
while (!predicate) {
   pthread_cond_wait(&condvar, &mutex);
}
pthread_mutex_unlock(&mutex);
```

Listing 14 POSIX Mutex - warten auf Bedingungen

Falls in diesem Code-Teil die Variable predicate den Wert false hat, dann wird der Thread mittels `pthread_cond_wait()` schlafen gelegt. Wichtig ist es dabei zu wissen, dass der Mutex-Lock beim warten abgegeben und erst beim aufwecken wieder zugewiesen wird. Um ihn wieder aufzuwecken kann folgendes Code-Fragment verwendet werden.

```
pthread_mutex_lock(&mutex);
predicate=1;
pthread_cond_broadcast(&condvar);
pthread_mutex_unlock(&mutex);
```

Listing 15 POSIX Thread - condiditonal wait

Das Aufwecksignal wird über die Variable `condvar` gesendet.

Weiterführende Informationen:

- Mark Hays, POSIX Thread Tutorial: [POSIXTUTOR]

7.3.2. OpenMP

Bei OpenMP handelt es sich wie bei POSIX auch um eine API Spezifikation. Allerdings auf einer ganz anderen Ebene. OpenMP definiert eine Schnittstelle für so genannte Compiler-Direktiven. Das Ziel dabei ist es, dass der Programmierer sich nicht selber um die Erzeugung, Synchronisierung und Terminierung von Threads kümmern muss. All diese Aufgaben werden automatisch vom Compiler übernommen. Zu diesem Zweck definiert OpemMP spezielle Compiler-Direktiven. Diese konzentrieren sich insbesondere auf die Parallelisierung von Schleifen welche häufig in mehrere parallel laufende Schleifen aufgeteilt werden können. Dieser Verfahren wird auch Data partitioning genannt.

Ein grosser Vorteil von OpenMP ist, dass die Compiler-Direktiven von Compilern die kein OpenMP unterstützen, einfach ignoriert werden. Dies erlaubt dem Programmierer eine portable Implementierung. Nachfolgend eine Liste der Compiler, die bekanntermassen OpenMP unterstützen:

Tabelle 27 Compiler mit OpenMP Unterstützung

Compiler	Beschreibung
Microsoft Visual C++	Unterstützung erst ab Visual Studio 2005. Ausserdem unterstützt Visual Studio 2005 Express Edition kein OpenMP weil die benötigten Libraries fehlen.
Intel	Die aktuellen Intel Compiler unterstützen OpenMP und sind zu Evaluierungszwecken kostenlos bei Intel erhältlich. Siehe dazu auch [INTELC].
GCC	GCC wird erst ab Version 4.2 OpenMP unterstützen. Es dürfte aber nicht mehr lange dauern bis Version 4.2 freigegeben wird. Snapshots können bereits auf der offiziellen Homepage bezogen werden. Siehe dazu auch [GCC].

Einer der grossen Vorteile von OpenMP liegt darin, dass bestehender (sequenzieller) Code ohne manuelle Thread-Behandlung parallelisiert werden kann. Dies kann anhand eines kurzen Beispieles aus [2] verdeutlicht werden.

Original Code:

```
double w=1.0 / (double) n;
double sum = 0, x;
for (int i=0; i<=n; i++) {
   x = w * ((double)i - 0.5);
   sum += 4 / (1 + x * x );
}
pi = w * sum;
printf(“pi = %13lf\n”, pi);
```

Listing 16 OpenMP, parallelisierbarer Code

Dieser Code kann nun mit OpenMP Anweisungen parallelisiert werden:

```
double w = 1.0 / (double) n;
double sum = 0, x, f_x;
#pragma omp parallel for private(x, f_x) shared(w, sum)
for (int i=0; i<=n; i++) {
   x = w * ((double)i - 0.5);
   f_x = 4 / (1 + x * x );
#pragma omp critical
   sum += f_x;
}
pi = w * sum;
```

Listing 17 OpenMP, parallelisierter Code

Auf einem 4-Prozessor System würde die for-Schleife jetzt standardmässig auf 4 Threads aufgeteilt und parallel bearbeitet. Die `#pragma omp critical` Anweisung ist notwendig weil alle Threads hier

synchronisiert werden müssen um die Konsistenz der Variable `sum` zu gewährleisten. OpenMP bietet hierzu auch alternativ die Pragma Anweisung `atomic` an. Hier wäre `atomic` effizienter, lässt sich aber nur auf einzelne Anweisungen und nicht auf Code-Blöcke anwenden.

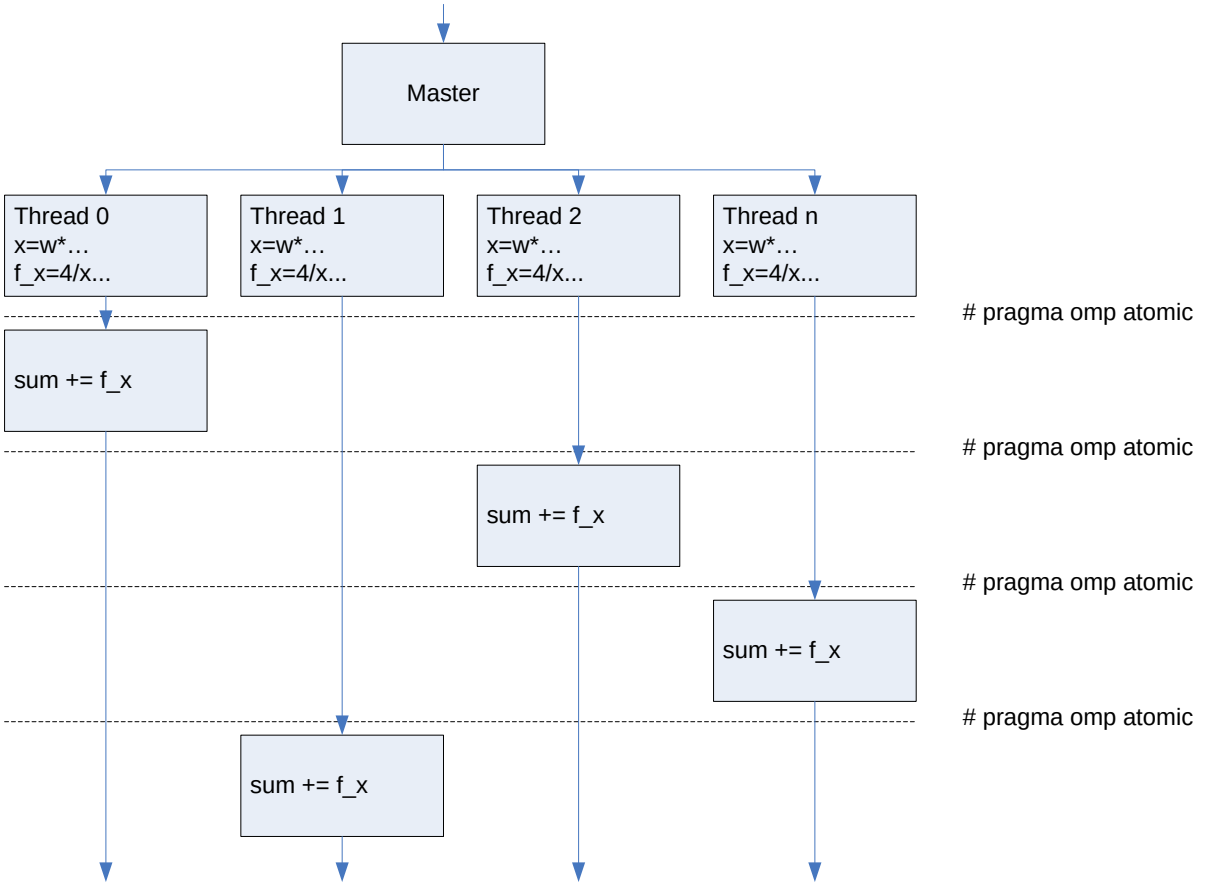


Abbildung 24 Parallele Verarbeitung der Beispiel-Schleife mit OpenMP

Lässt man das Beispiel so laufen, dann werden auch die Gefahren der OpenMP Programmierung deutlich. Das parallel ablaufende Beispiel läuft hier mit zwei oder mehr Threads tatsächlich massiv langsamer ab. Dies liegt daran, dass die Synchronisation des kritischen Bereiches (Summierung) im Vergleich zur Berechnung viel zu aufwändig ist und mehr Rechenzeit in Anspruch nimmt als durch die parallele Verarbeitung gewonnen wird.

Um solche Effekte zu minimieren bietet OpenMP weitere Direktiven. Beispielsweise kann die Summierung durch OpenMP durchgeführt werden. Dies hat zur Folge, dass die Synchronisierung nur einmal (am Ende der Berechnungen) und nicht mehr bei jedem Schleifendurchlauf zu erfolgen hat:

```
double w = 1.0 / (double) n;
double sum = 0, x;
#pragma omp parallel for private(x) shared(w) reduction (+:sum)
for (int i=0; i<=n; i++) {
   x = w * ((double)i - 0.5);
   sum = 4 / (1 + x * x );
}
pi = w * sum;
```

Listing 18 OpenMP, reduction

Hier arbeiten die parallelen Threads jeweils mit einer lokalen `sum` Variablen. Am Ende der Berechnungen wird diese von OpenMP durch Addierung „reduziert“. Während der gesamten Berechnungsdauer ist keine Synchronisation der Threads notwendig weshalb auch nahezu das volle Potential der Parallelität ausgenutzt werden kann.

Eine wichtige Eigenschaft von OpenMP ist, dass standardmässig genau so viele Threads erzeugt werden wie Prozessoren zur Verfügung stehen. Dies lässt sich aber nachträglich (auch zu Testzwecken) über Umgebungsvariabeln beeinflussen. Wichtig ist dabei auch, dass die Threads über die gesamte Laufzeit des Programms bestehen bleiben. Dadurch wird der Aufwand die Threads laufend zu erzeugen und wieder zu entfernen umgangen was insgesamt der Effizienz zu Gute kommt.

Weiterführende Informationen:

- OpenMP, Homepage: [OPENMP]
- Wikipedia, OpenMP: [OPENMPWP]
- Sun, OpenMP Unterstützung: [OPENMPSUN]
- GNU, GCC 4.2 mit OpenMP Unterstützung: [GCC]
- Intel, Compilers: [INTELC]
- Oliver Lau, c't Ausgabe 15/2006, Seite 218ff: [2]

7.3.3. Thread Building Blocks (TBB)

Die Inten Thread Building Blocks (TBB) sind vom Prinzip her verwandt mit OpenMP. Die Implementierung geschieht hier allerdings in Form einer C++ Bibliothek. Auch TBB wird zur Parallelisierung von Schleifen verwendet. Hierzu ebenfalls ein kleines Beispiel:

```
#include „tbb/blocked_range.h“
#include “tbb/parallel_for.h”
void PrintArray(int[] &v, size_t n, size_t blocksize) {
   Printer whattodo(v);
   parallel_for(blocked_range<size_t>(0, n, blocksize), whattodo);
}
```

Listing 19 TBB, ein kleines Beispiel

Bei `size_t` kann es sich hierbei um `integer`, `long`, Pointer oder Iteratoren handeln. Die parallele `for` Schleife ruft hier für die Elemente 0 bis n das Objekt `whattodo` auf. Dazu muss das Objekt den Funktionsoperator „`()`“ überladen. Dieser wird dann durch jeden Thread mit einem anderen Blockbereich aufgerufen. Hier die Klasse Printer (Objekt: `whattodo`):

```
class Printer {
   int * const m_v;
public:
   Printer(int v[]) : m_v(v) {}
   void operator() (const blocked_range<size_t>& r) const
   { for (size_t i = r.begin(); i != r.end(); ++i)
         cout << m_v[i];
   }
};
```

Listing 20 TBB, Funktionsoperator überladen

Analog zu OpenMP entsteht natürlich auch hier das Problem der Synchronisation beim Zugriff auf gemeinsame Variabeln. OpenMP bietet dazu die `reduction` Klausel. TBB bietet zu diesem Zweck die `parallel_reduce` Schleife:

```
void SumUpArray(int v[], size_t n, size_t blocksize) {
   Summarizer s(v);
   parallel_reduce(blocked_range<size_t>(0, n, blocksize), s);
   cout << s.sum();
}
```

Listing 21 TBB, parallel_reduce

Wie man sieht unterscheidet sich der Aufruf nur im übergebenen Objekt. Dieses ist wie folgt aufzubauen:

```
class Summarizer {
   int * const m_v;
   int m_sum;
public:
   Summarizer(int v[]) : m_v(v), m_sum(0) {}
   void operator() (const blocked_range<size_t>& r) {
         for (size_t i = r.begin(); i != r.end(); ++i) {
               m_sum += m_v[i]
         }
   Summarizer(Summarizer& x, split) : m_v(x.m_v), m_sum(0) {}
   void join(const Summarizer& other) {
         m_sum += other.m_sum;
   }
   int sum(void) {
         retunr m_sum;
   }
```

```
};
```

Listing 22 TBB, Beispiel: Summarizer

Hier wird von `parallel_reduce` der zweite Konstruktor (Splitting-Konstruktor) aufgerufen. Der zweite Parameter `split` dient dabei nur zur Unterscheidung von einem Copy-Konstruktor. Das Summarizer Objekt wird also von `parallel_reduce` mehrfach erzeugt. Durch die Übergabe einer Referenz auf das originale Objekt kann die Member-Variable `m_v` übernommen werden (Wertebereich). Die Membervariable `m_sum` muss natürlich lokal beleiben. Am Schluss der Operation wird die Methode `join()` aufgerufen. Dort können die Werte dann aufsummiert werden. Praktischerweise könnte man die Methode `join()` noch erweitern um beispielsweise den Maximal- oder Minimalwert noch zu erhalten.

Die TBB bieten ausserdem noch erweiterte Werkzeuge, die teilweise etwas über die Möglichkeiten von OpenMP hinausgehen. Beispielsweise `parallel_while`, `pipeline` oder die zweidimensionale Segmentierung. Einen Überblick darüber vermittelt die Quelle [3] aus der auch die oben aufgeführten Codebeispiele stammen. Interessierte finden natürlich direkt bei Intel ([INTELTBB]) weitere Informationen.

Ein kleiner Wehrmutstropfen liegt darin, dass die TBB Bibliothek nicht frei verfügbar ist. Lediglich eine Linux-Version für Nichtkommerzielle Zwecke liegt zum Download auf der Intel Webseite.

Weiterführende Informationen:

- Oliver Lau, c't Ausgabe 21/2006, Seite 234ff, Thread-Baukasten/TBB: [3]
- Intel, Thread Building Blocks 1.0 for Windows, Linux and Mac OS: [INTELTBB]

7.3.4. MPI

Das Message Passing Interface (MPI) ist zwar nicht direkt eine Technologie zur Parallelisierung eines Programmcodes kümmert sich aber um eines der wichtigsten Probleme der parallelen und verteilten Programmierung. Wir haben gesehen, dass die Kommunikation von Prozessen untereinander häufig ein grosses Problem darstellt. Durch die Threadbasierende Programmierung kann dieses Problem etwas entschärft werden da die Kommunikation über den gemeinsamen Prozesskontext laufen kann. Wir die Anwendung aber auf mehrere Prozesse verteilt oder gar auf mehreren Rechnern ausgeführt so sollten diese möglichst einfach und direkt miteinander kommunizieren können.

Die MPI-Schnittstelle erlaubt es den Programmen untereinander direkte Nachrichten auszutauschen. Dies kann sowohl lokal über den Hauptspeicher als auch über Rechnergrenzen hinweg beispielsweise über TCP/IP geschehen.

Da wir uns hier hauptsächlich mit der Thread-Programmierung beschäftigen und uns auf lokale Systeme beschränken wird MPI für uns nicht relevant sein. Für grössere und verteilte Anwendungen kann es aber durchaus hilfreich sein.

Weiterführende Informationen:

- Wikipedia, Message Passing Interface: [MPI]
- Alexander Greiml, Universität Trier, Message Passing Interface (MPI): [MPI-TRIER]

7.4. Zusammenfassung und Fazit

Dieses Kapitel hat einen Überblick über die Programmierung von parallel ablaufenden Programmen gegeben. Insbesondere die dadurch entstehenden Probleme der Kommunikation und der Synchronisierung sowie Methoden zu deren Lösung wurden vorgestellt. Die vorgestellten Techniken wie POSIX Threads, OpenMP, TBB und MPI werden für uns nur am Rande wichtig sein, da dies keine Techniken auf Java-Ebene darstellen. Diese Techniken sind aber sehr wohl für die Java Virtual Machine (siehe Kapitel 8) wichtig. Für Programmierer einer JVM können die vorgestellten Technologien durchaus wichtig sein. Wie viele davon bei der Programmierung einer Java-Applikation wichtig sein werden wird sich im Verlauf der Arbeit zeigen.

7.5. Auswirkungen auf die Aufgabenstellung

Dieses Kapitel bietet einen Überblick über Parallelisierungs-Techniken auf Applikations-Ebene. Einige der vorgestellten Technologien könnten für uns auf Java-Ebene wichtig werden:

Tabelle 28 Technologien mit direktem Einfluss auf die Arbeit

Technologie	Beschreibung
POSIX Threads	Auf Java-Ebene werden zwar nicht direkt Posix-Threads verwaltet aber Java bietet eine ähnliche Schnittstelle über die Java-API. Ob die JVM die Thread-Verwaltung über die POSIX-Schnittstelle abwickelt oder direkte Betriebssystemspezifische Routinen verwendet werden wir möglicherweise noch erfahren.
OpenMP	Auch OpenMP ist prinzipiell auf C/C++ und Fortran limitiert (siehe Kapitel 7.3.2). Wir konnten aber das Projekt JOMP (siehe [PROCEXP]) finden welches zum Ziel hat dieselbe Funktionalität für Java-Anwendungen zur Verfügung zu stellen. Deshalb könnte diese Technologie für unsere Arbeit relevant sein.

Tabelle 29 Technologien mit indirektem Einfluss auf die Arbeit

Technologie	Beschreibung
TBB	Die Intel Thread Building Blocks besteht aus einer reinen C/C++ Bibliothek, deshalb ist diese Technologie für uns mit Fokus auf Java-Implementierung nicht relevant. Im entfernten Sinne könnten einige in der Java API vorhandenen Klassen ähnliche Funktionalitäten übernehmen.
MPI	Das Message Passing Interface ist für uns nicht weiter von Interesse, da wir nicht die vertikale Skalierung auf mehreren Systemen (Cluster) untersuchen sondern die Skalierung auf einem einzigen Host. Zwar kann MPI auch zur Inter-Prozess-Kommunikation genutzt werden, angesichts moderner Thread-Unterstützung macht es aber eher wenig Sinn mehrere Prozesse zu verwenden. Zur Kommunikation zwischen Threads (geteilter Adressraum) wird MPI nicht benötigt.

8. Java Virtual Machine (JVM)

Für Java-Basierende Applikationen spielt die Java Virtual Machine (JVM) eine zentrale Rolle. Alle Java-Applikationen laufen dabei in der Virtuellen Maschine ab. Dies bedeutet natürlich, dass die Virtuelle Maschine eine weitere Ebene zwischen dem Betriebssystem und der eigentlichen Java-Anwendung darstellt. Für Java-Anwendungen ist es daher auch nicht möglich einfach auf Funktionen des Betriebssystems zugreifen zu können. Diese müssen sich somit mit der von der Virtuellen Maschine zur Verfügung gestellten API zufrieden geben.

Die Java-VM dafür sorgen, dass die Applikation mit optimaler Geschwindigkeit ablaufen kann. Ausserdem bietet die VM gegenüber der Applikation eine Schnittstelle (API) um parallele Programmierung zu ermöglichen beziehungsweise diese möglichst einfach zu gestalten.

In diesem Kapitel solle daher die Java API zur parallelen Programmierung betrachtet werden. Ausserdem wird untersucht in wie fern die API Optimierungen auf Betriebssystem und Hardware zulässt. Da anzunehmen ist, dass die Plattformunabhängige Java-API keine tief greifenden und Betriebssystemabhängigen Konfigurationen bietet soll auch die JVM selbst genauer betrachtet werden. Diese selbst ist Plattform-abhängig und soll für eine optimale Ausführung der Anwendung sorgen. Es ist anzunehmen, dass die JVM entweder durch Konfiguration oder in Form speziell optimierter Varianten an Betriebssystem und Hardware angepasst werden kann.

Die folgende Grafik zeigt die Architektur der Sun Java Plattform in der Version 5.

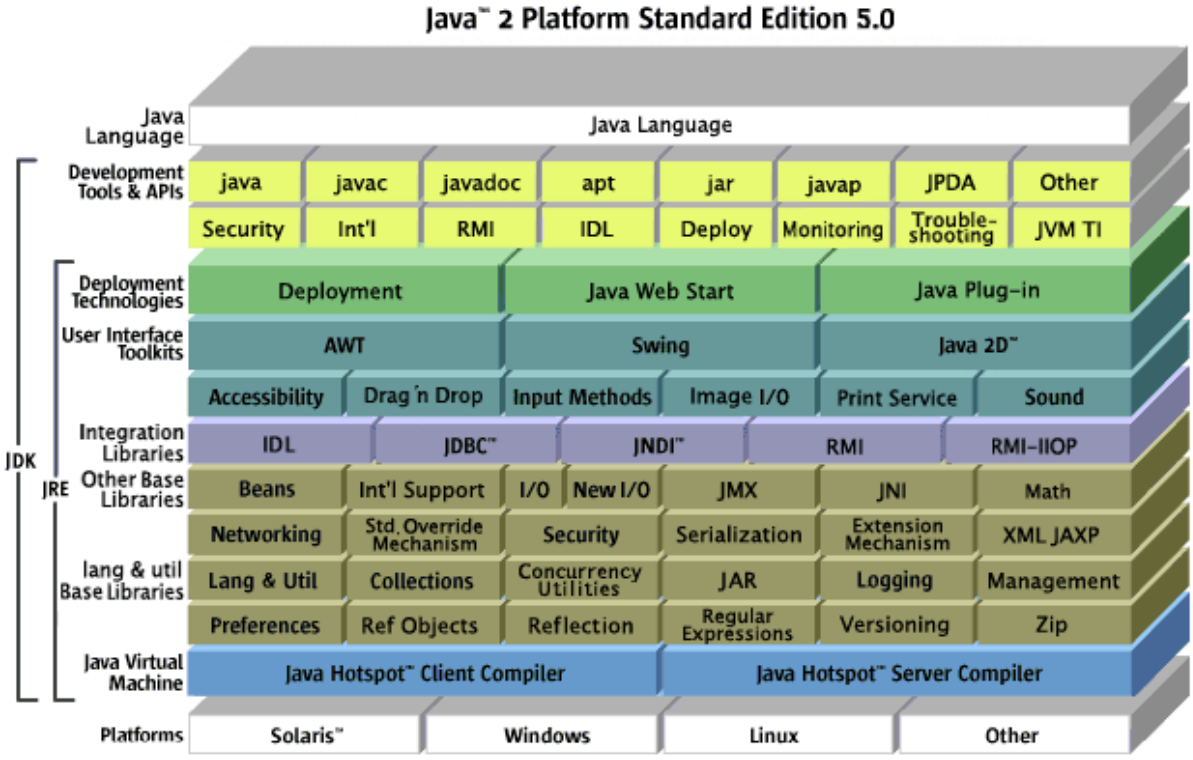


Abbildung 25 Sun Java VM Architektur

Wie gut zu erkennen ist stellt die eigentliche Virtuelle Maschine (Java Virtual Machine) eine Ebene zwischen dem Betriebssystem und der eigentlichen Java-Sprache dar. Dazwischen liegt die Java API. Diese bietet dem Programmierer einen vordefinierten Sprachumfang und damit eine Schnittstelle zur einfacheren Programmierung. Die API beinhaltet beispielsweise vordefinierte Containerklassen wie Vektoren und Maps. Diese müssen also vom Entwickler nicht mehr implementiert werden.

Weiterführende Informationen:

- Sun, Java Langugage Specification: [JLS]
- Sun, Java Virtual Machine Specification: [JVMS]

- Sun, Java API Reference: [JAPIREF]

8.1. Die Java API

Bereits in der Java Language Specification (siehe auch [JLS]) ist der Umgang mit Threads und die Behandlung von Nebenläufigkeit exakt spezifiziert. Somit ist die parallele Programmierung im Gegensatz zu vielen anderen Sprachen ein integraler Bestandteil der Sprachdefinition.

Wie bereits erwähnt stellt die Java API die Schnittstelle zwischen Anwendung und Java Virtual Machine (JVM) dar. Diese Schnittstelle beinhaltet einige wichtige Klassen, welche die parallele Programmierung ermöglichen und vereinfachen. Diese Schnittstellen sollen im Folgenden kurz beschrieben werden. Weiterführende Informationen zur Java API sind auf der Java Homepage ([JAPIREF]) zu finden.

8.1.1. Threads

Wie gesagt bietet Java eine in der Sprache selbst verankerte Thread-Unterstützung. Streng genommen gibt es innerhalb von Java gar keine Prozesse. Die Java Virtual Machine ist der einzige, sichtbare Prozess gegenüber dem Betriebssystem. Die statische main() Methode stellt dabei den Eintrittspunkt des Programmes dar. Diese wird aber bereits von einem JVM-internen Thread mit der Bezeichnung „main“ aufgerufen. Somit ist auch der Haupt-Ausführungsstrang einer Java-Applikation nichts weiter als ein Thread.

Die zur Thread-Verwaltung und parallelen Verarbeitung verwendeten Mechanismen wie Monitore sind in [JLS] beschrieben. Der Thread Lebenszyklus sieht wie folgt aus:

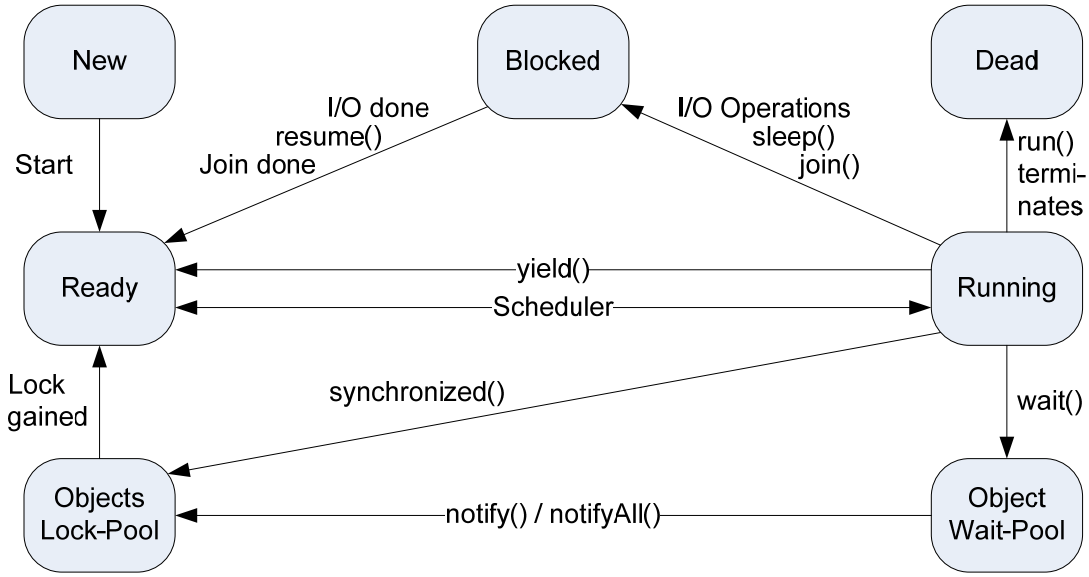


Abbildung 26 Thread Lebenszyklus, (Quelle: [1])

Jedes Objekt in Java besitzt einen Lock- und einen Wait-Pool. Dadurch sind die wichtigsten Synchronisationsprobleme bereits ohne Zusatzaufwand lösbar:

- Konkurrierenden Zugriff verhindern: Kann mittels Lock-Pool (Zeitgleich erhält nur ein Thread einen Lock) realisiert werden.
- Warten auf ein Ereignis: Kann mittels Wait-Pool realisiert werden.

Hierbei sind einige wichtige Bedingungen zu beachten:

- Sowohl `wait()` als auch `notify()` bzw. `notifyAll()` dürfen nur aufgerufen werden, wenn der Thread im Besitz des Locks für dieses Objekt ist.

- Ein Thread, der `wait()` aufruft gibt implizit den Lock ab und kommt somit bei erhaltenem `notify()` automatisch in den Lock-Pool um den Lock wieder neu zu erhalten. Ein mit `notify()` „aufgeweckter“ Thread läuft erst weiter, wenn er den Lock wieder bekommen hat.
- Ein Thread der aufgrund einer I/O Operation, `sleep()` oder `join()` blockiert wird gibt den Lock nicht ab. Dies ist wichtig zu wissen weil dieser unter Umständen dann für längere Zeit blockiert bleibt.

Mehr zur Synchronisierung in Kapitel 0.

Es gibt prinzipiell zwei Möglichkeiten einen Thread unter Java zu erzeugen. Einerseits kann von der Klasse `java.lang.Thread` abgeleitet werden und andererseits kann eine beliebige Klasse die Runnable-Schnittstelle implementieren (`java.lang.Runnable`). Da Java keine Mehrfachvererbung unterstützt und Klassen manchmal schon von anderen Klassen abgeleitet sind bleibt häufig nur die zweite Möglichkeit. In beiden Fällen muss aber nur die `run()` Methode implementiert werden. Diese wird beim Start des Threads ausgeführt. Wie in Abbildung 26 zu sehen ist endet ein Thread mit dem Ende der `run()` Methode.

Hier ein Beispiel einer Thread-Klasse, die von `java.lang.Thread` abgeleitet ist:

```
package ch.skybeam.examples;

public class MyThread extends Thread {
    // fields, methods...

    @Override
    public void run() {
        super.run();
        // do some stuff
    }
}
```

Listing 23 Java, Threaderzeugung durch Ableitung

Ein konkreter Thread kann danach mittels folgendem Code (z.B. in der `main()` Methode) erzeugt werden:

```
Thread t = new MyThread();
t.start();
```

Listing 24 Java, Thread starten (Thread Klasse)

Analog dazu eine Klasse, welche die Runnable-Schnittstelle implementiert:

```
package ch.skybeam.examples;

public class MyRunnable implements Runnable {
    // fields, methods....

    public void run() {
        // do some stuff
    }
}
```

Listing 25 Java, Thread mittels Runnable Interface

Die Erzeugung des Threads geschieht analog dazu:

```
Thread t = new Thread(new MyRunnable());
t.start();
```

Listing 26 Java, Thread starten (Runnable Interface)

Hier wird lediglich der Basisklasse Thread das zuvor definierte Runnable-Objekt übergeben. Das erzeugte Thread-Objekt verhält sich danach gleich wie im Beispiel zuvor.

Wichtig: Beim Start von Threads ist darauf zu achten nicht die `run()` Methode auszuführen sondern die `start()` Methode der Klasse `java.lang.Thread`. Wird die `run()` Methode ausgeführt, so

verhält sich der Aufruf wie ein Methodenaufruf und wird im Kontext des aufrufenden Threads ausgeführt (z.B. im „main“ Thread).

Es folgt eine Auflistung der wichtigsten Methoden der Thread Klasse (`java.lang.Thread`):

Tabelle 30 Wichtige Methoden von java.lang.Thread

Methode	Beschreibung
`start()`	Diese Methode aktiviert den Thread. Das führt zum Aufruf der run() Methode in der Thread-Klasse bzw. der im Konstruktor angegebenen Runnable-Klasse.
`getPriority()` `setPriority()`	Mit diesen Methoden kann die Thread-Priorität abgefragt bzw. beeinflusst werden.
`isDaemon()` `setDaemon()`	Erlaubt die Abfrage des Daemon-Status eines Threads. Standadmässig sind alle Java Threads so genannte User Threads. Die JVM beendet sich erst, wenn alle User-Threads beendet sind. Wenn die main() Methode beendet ist überprüft die JVM ob noch User-Threads abgearbeitet werden und wartet falls nötig auf deren Terminierung. Daemon-Threads werden dabei ignoriert. Soll also ein Thread einfach bis zur Beendung der JVM weiterlaufen so empfiehlt sich die Definition als Daemon. Dieser wird dann bei der Terminierung der JVM automatisch beendet.
`getState()`	Seit Java 1.5 ist es möglich den aktuellen Status eines Threads zu erfragen. Dies ist insbesondere zu Debug-Zwecken sinnvoll oder zur Überwachung des Systemstatus.
`interrupt()` `isInterrupted()`	Den Abbruch eines Threads mittels der interrupt() Methode ist die bevorzugte Art der vorzeitigen Thread-Terminierung. Vorsicht: Wird interrupt() auf einem Thread aufgerufen, der sich im „blocked“ Zustand befindet so wird eine InterruptedException geworfen. Wird diese nicht abgefangen terminiert der Thread natürlich. Die Korrekte Art eines Interrupted-Handling wäre die aktive Prüfung von isInterrupted() durch den Thread und eine entsprechende Terminierung der run() Methode.
`join()`	Bewirkt, dass der aktuelle Thread auf die Terminierung des Threads auf dem join() aufgerufen wird wartet.
`yield()`	Mittels dieser Methode lässt sich die Rechenzeit freiwillig abgeben. Der aufrufende Thread wird automatisch wieder in den Status „ready“ versetzt. Dies erlaubt die Abgabe von Rechenzeit zugunsten anderer Threads.

Warnung: Eine Methode namens `stop()` existiert zwar aber sollte nicht mehr verwendet werden da diese Methode keine Möglichkeit zur sauberen Terminierung (Aufräumen von Datenstrukturen, Freigabe/Schliessung von Dateien und Handlern, beenden von Transaktionen) vorsieht.

Mehrere Threads können auch in einer so genannten Thread Gruppe zusammengefasst werden. Dazu wird zunächst eine leere Gruppe erzeugt und bei der Erzeugung der Threads als Argument übergeben:

```
ThreadGroup tg = new ThreadGroup("Thread-Gruppe");
new Thread(tg, new MyRunnable()).start();
```

Listing 27 Java, Threadgruppen

Nun können einige Operationen direkt auf der Gruppe anstatt auf den einzelnen Threads ausgeführt werden. Beispielsweise lässt sich durch

```
tg.interrupt();
```

Listing 28 Java, Threadgruppen (Interrupt)

das Interrupt-Signal an alle Threads in der Gruppe senden. Eine Methode um auf die Terminierung aller Threads in der Gruppe zu warten existiert allerdings nicht. Um dies zu erreichen muss also die

Liste der Threads aus der Gruppe herausgeholt werden um die `join()` Methode jedes einzelnen Threads aufzurufen.

8.1.2. Collections

Wo parallel verarbeitet wird finden praktisch immer auch Zugriffe auf gemeinsame Ressourcen oder gemeinsame Datenfelder statt. Java bietet bereits eine Reihe von so genannten Collections. Collections sind Container-Klassen um Daten in einer bestimmten Struktur abzulegen. Speziell bei der parallelen Programmierung ist es wichtig, dass solche Container Thread-Safe sind. Dies bedeutet, dass selbst bei parallelen Zugriffen auf die Daten kein undefinierter oder ungewollter Zustand eintreten kann. Die meisten der bis zu Java 1.4.x vorhandenen Collection-Klassen sind nicht synchronisiert und somit nicht Thread-Safe. Bei Java 5 kamen hier einige sehr wichtige neue Klassen hinzu. Die wichtigsten sollen hier kurz vorgestellt werden:

Tabelle 31 Neue Concurrent-Collections in Java 5 (java.util.concurrent Package)

Klasse	Beschreibung
`ConcurrentHashMap`	Wurde als Ersatz für die Hashtable Klasse entwickelt und erlaubt massiven parallelen Zugriff. Lesende Zugriffe blockieren nie und für schreibende Zugriffe lässt sich die Locking-Strategie beeinflussen bzw. optimieren.
`CopyOnWriteArray*`	Diese Klassen erstellen bei jeder Modifikation eine Kopie des Arrays. Iterationen auf dem Array können also auf dem unveränderten Array zu Ende geführt werden. Ist sehr gut für nur-lese Strukturen mit seltenen Änderungen geeignet.
`Queue`	Queue Klassen implementieren eine FIFO (First In First Out) Queue. Ist die Queue voll, dann blockiert ein put() Aufruf nicht sondern liefert einen Fehler. Ist die Queue leer, dann blockiert auch take() nicht sondern liefert eine null-Referenz.
`BlockingQueue`	Diese Klassen implementieren ebenfalls eine FIFO (First In First Out) Queue. Zusätzlich bliockieren hier die put() und take() Methoden bei voller bzw. leerer Liste bis ein Element herausgenommen bzw. entfernt wird.

8.1.3. Weitere hilfreiche Klassen

8.1.3.1. ReentrantLock

Die Klasse `java.util.concurrent.locks.ReentrantLock` bietet einen erweiterten Locking-Mechanismus im Vergleich zur Synchronisation mit dem `synchronized` Statement. Die Klasse bietet einige Methoden, die Funktionen offerieren, welche nicht von `synchronized` übernommen werden könne. Hier eine Liste der wichtigsten:

Tabelle 32 Wichtige ReentrantLock Methoden

Methode	Funktion
`isLocked()`	Erlaubt die Abfrage, ob der Lock momentan bereits vergeben ist.
`lock()`	Versucht den Lock zu bekommen. Der Aufruf hat denselben Effekt wie das Einteten in einen synchronized Block.
`lockInterruptibly()`	Erlaubt den nachträglichen Abbruch während der Thread auf den Lock warten muss. Dies ist mit synchronized nicht möglich.
`tryLock()`	Versucht den Lock zu bekommen. Falls dieser bereits vergeben ist blockiert die Methode nicht sondern meldet lediglich, dass der Versuch erfolglos war.

`tryLock(timeout)` Versucht den Lock zu bekommen, wartet aber nur maximal bis zum Ablauf der Zeitüberschreitung und meldet dann einen erfolglosen Versuch zurück. Dies ist mit synchronized ebenfalls nicht möglich.

Die Verwendung kann anhand des folgenden Codebeispiels verdeutlicht werden:

```
boolean success = lock.tryLock();
if(success) {
    try {
        System.out.print(“Reading value: " + sync.getA());
    } finally {
        lock.unlock();
    }
} else {
    System.out.print(" No lock acquired :-(");
}
```

Listing 29 Java, ReentrantLock (tryLock)

Analog dazu das Lesen mit Timeout:

```
boolean success = false;
try {
    success = lock.tryLock(500, TimeUnit.MILLISECONDS);
} catch (InterruptedException e1) {
    // interrupted during lock-try
}
if (success) {
    try {
        System.out.print("Reading value: " + sync.getA());
    } finally {
        lock.unlock();
    }
} else {
    System.out.print(" No lock acquired :-(");
}
```

Listing 30 Java, ReentrantLock (tryLock mit Timeout)

In beiden Fällen ist es wichtig, dass der gesamte Code zwischen der Lock-Anfrage und der Lock-Freigabe in einem `try` Block steht und die Lock-Freigabe im zugehörigen `finally` Block. Dies stellt sicher, dass der Lock auf jeden Fall wieder freigegeben wird. Bei `synchronized` Blöcken ist dies nicht nötig, da bei einer Exception der Block automatisch verlassen und der Lock freigegeben wird. Bei der manuellen Lock-Behandlung wird ein vergessen gegangener Lock aber nicht automatisch wieder freigegeben und bleibt bestehen.

Der Konstruktor der Klasse `ReentrantLock` bietet einen Parameter um die Fairness einzuschalten (boolean Parameter). Fairness garantiert, dass die Threads in der Reihenfolge ihrer Anfragen den Lock bekommen. Dies mag auf den ersten Blick verlockend klingen hat aber einen massiven Performance-Einbruch zur Folge (siehe [4] S. 284). Der Geschwindigkeitsverlust (besonders bei vielen parallelen Anfragen) liegt in der zusätzlich nötigen Synchronisation der Threads und dem damit verbundenen Aufwand. Da die meisten Algorithmen nicht auf Fairness angewiesen sind sollte diese Option nur in begründeten Einzelfällen Verwendung finden.

8.1.3.2. Atomic*

Java 5 bietet im `java.util.concurrent.atomic` Package neue Klassen für die atomare Behandlung der grundlegenden Datentypen. Diese Klassen arbeiten nach dem nicht-blockierenden CAS Prinzip (siehe Kapitel 8.2.4) und arbeiten daher sehr effizient bei niedriger bis mittlerer lock contention.

Es folgt eine Liste der wichtigsten Methoden am Beispiel der AtomicInteger Klasse:

Tabelle 33 Wichtige Methoden der AtomicInteger Klasse

Methode	Beschreibung
`addAndGet()`	Diese Methode addiert in einem Atomaren Vorgang den angegebenen Wert.
`compareAndSet()`	Lässt den Benutzer die CAS-Funktion direkt ausführen. Damit lassen sich CAS-Algorithmen implementieren (siehe Kapitel 8.2.4).
`decrementAndGet()` `incrementAndGet()`	Entspricht der Integer-Operation „`--i`“, bzw. „`++i`“ wird aber atomar ausgeführt.
`getAndDecrement()` `getAndIncrement()`	Entspricht der Integer-Operation „`i--`“ bzw. „`i++`“ wird aber atomar ausgeführt.
`getAndAdd()`	Addiert den angegebenen Wert und gibt das Resultat zurück.
`getAndSet()`	Setzt den angegebenen wert und gibt den alten Wert zurück.

8.2. Synchronisierung

In Java sind Elemente zur Synchronisation direkter Bestandteil des Sprachumfangs. Die Synchronisierung ist die wohl heikelste Angelegenheit bei der parallelen Programmierung. Überall wo ein gemeinsamer Zugriff stattfindet müssen diese Zugriffe synchronisiert werden um konsistente Daten zu gewährleisten. Die einfachste Form der Synchronisation ist ein Mutex (siehe Kapitel 8.2.1). Der Begriff Mutex wird häufig als Synonym für Lock verwendet. Ein Mutex hat die Eigenschaft, dass nur eine einzige Anfrage erfolgreich ist. Alle weiteren Anfragen den Mutex/Lock zu erhalten führen zur Blockierung des aufrufenden Threads bis der Mutex wieder freigegeben wird.

Finden nur wenige/vereinzelte Zugriffe auf diese Methode statt, so fällt die Synchronisierung kaum ins Gewicht da der aufrufende Thread in der Regel sofort den Lock bekommen kann. Bei vielen parallelen Zugriffen führt dies natürlich zu einem Engpass (da nur ein Thread zeitgleich passieren darf). Der Grad der konkurrierenden Zugriffe wird ‚lock contention' genannt. Eine hohe ‚lock contention' bedeutet, dass viele Threads gleichzeitig versuchen in den kritischen Bereich einzutreten. Da dies immer nur einem zur gleichen Zeit gelingen kann müssen alle anderen Threads warten. Dies kann zu ungewollter Blockierung von Applikationsteilen führen. Ist die ‚lock contention' niedrig, so reduziert sich der Aufwand im Optimalfall auf das setzen des Locks (ein einziger Thread fordert den Lock an).

Es ist also wünschenswert die lock contention so niedrig wie möglich zu halten. Dazu gibt es drei wirksame Wege dies zu erreichen (aus [4] S. 233):

- Reduzierung der Dauer während der ein Lock gehalten wird.
- Reduzierung der Anfragehäufigkeit eines Locks.
- Exclusive Locking Techniken durch andere Koordinationsmechanismen mit besserer Konkurrierungsfähigkeit ersetzen.

Der erste Punkt zielt darauf ab den Durchsatz zu erhöhen indem ein Lock möglichst schnell wieder abgegeben wird. Damit kann der nächste Thread schneller in den geschützten Bereich eintreten.

Der zweite Punkt zielt auf Zwei Eigenschaften. Einerseits ist die Wahrscheinlichkeit eines Engpasses und gleichzeitigen Zugriffs höher, wenn mehr Anfragen auf denselben Lock stattfinden und andererseits ist mit jeder Anfrage der Aufwand zur Prüfung des aktuellen Lock-Status verbunden. Im Falle eines bereits gesperrten Locks kommt noch der Aufwand der Blockierung und dem erneuten Versuch hinzu.

Für beide Punkte werden in den Kapiteln 8.2.1.1 und 8.2.1.2 die Methoden zur Realisierung vorgestellt. Kapitel 8.3 enthält eine praktische Analyse verschiedener Locking-Strategien.

Der dritte Punkt ist etwas komplexer zu realisieren aber es gibt Techniken um den gemeinsamen Zugriff auch ohne Locks und der damit verbundenen Blockierung zu synchronisieren. Mehr dazu im Kapitel 8.2.4.

Weiterführende Informationen:

- Brian Goetz, Java Concurrency in Practice, ISBN 0-321-34960-1: [4]

8.2.1. Mutex

Die Bezeichnung Mutex ist eine Abkürzung für den englischen Ausdruck „Mutal Exclusion“ und bezeichnet den wechselseitigen Ausschluss im Zusammenhang mit gleichzeitigem Zugriff. Ein Mutex erlaubt immer nur einem einzigen Thread den Eintritt in einen geschützten Bereich. Alle nachfolgenden Anfragen führen zur Blockierung des anfragenden Threads. Erst wenn die Sperrung wieder aufgehoben wird (durch Freigabe des Mutex durch den besitzenden Thread) kann der nächste Thread in den Bereich eintreten.

Folgendes Code-Fragment verdeutlicht die Funktion:

```
public void sharedMethod() {
    lock.acquire();
    // do modification on shared data
    lock.release();
}
```

Listing 31 Java, Mutex

Dieses Konstrukt stellt sicher, das sich immer nur ein einziger Thread zeitgleich im kritischen Bereich aufhält. In Java werden solche Bereiche mit dem Schlüsselwort `synchronized` gekennzeichnet (siehe Kapitel 8.2.1.1 und 8.2.1.2).

Weiterführende Informationen:

- Wikipedia, Mutex: [MUTEX]

8.2.1.1. Blocksynchronisation

Java stellt für jedes Objekt einen Lock-Pool und einen Wait-Pool zur Verfügung (siehe auch Kapitel 8.1.1). Diese Pools werden über Monitore verwaltet. Um einen Block mit exklusivem Zugriff zu definieren wird zunächst ein Objekt benötigt welches den Lock-Pool zur Verfügung stellt. Dies kann ein beliebiges (auch ansonsten unverwendetes) Objekt sein:

```
package ch.skybeam.examples;
public class ObjectSync {
    private Object lock = new Object();

    public void methodWithLock() throws InterruptedException {
        // some code
        synchronized(lock) {
            // some more code, Thread holds lock of 'lock' object
            lock.wait(); // here the lock is freed
            // here the lock is gained again
        }
    }
}
```

Listing 32 Java, Blocksynchronisation

In diesem Code wird ein leeres Objekt mit dem Namen `lock` erzeugt. Die Methode `methodWithLock()` kann von verschiedenen Threads gleichzeitig aufgerufen werden. Sobald einer der Threads den `synchronized` Block erreicht versucht dieser den Lock auf das `lock` Objekt zu erhalten. Derjenige Thread, der den Lock bekommt darf im Block weiterlaufen. Alle anderen Threads bekommen den Lock nicht und verbleiben im Lock-Pool. Sobald der Thread mit dem Lock den `synchronized` Block verlässt wird der Lock wieder freigegeben. Damit kann ihn der nächste Thread erhalten und in den Block eintreten. Die Modellierung entspricht dem gängigen Idiom der Mutex Synchronisierung.

Würde innerhalb des `synchronized` Blocks die Methode `wait()` des `lock` Objektes aufgerufen, so würde der Thread in den Wait-Pool fallen und der Lock abgegeben. Somit könnte der nächste Thread den Lock bekommen und in den Block eintreten.

Es können beliebig viele Threads im Lock- sowie im Wait-Pool eines Objektes liegen. Die Methode `notify()` des Lock-Objektes holt dabei ein Objekt aus dem Wait-Pool heraus. Die Methode `notifyAll()` tut dasselbe, holt aber alle wartenden Threads aus dem Wait-Pool. Danach kann aber nur derjenige Thread direkt weiterlaufen, der anschliessend den Lock wieder bekommt.

Hier muss dafür gesorgt werden, dass alle Threads das `notify()` Signal bekommen. Ansonsten droht Starvation (siehe Kapitel 7.1).

Da jedes Objekt einen Wait- und einen Lock-Pool hat kann der oben stehende Code auch folgendermassen geschrieben werden:

```
package ch.skybeam.examples;
public class ObjectSync {
    public void methodWithLock() throws InterruptedException {
        // some code
        synchronized(this) {
            // some more code, Thread holds lock of 'lock' object
            lock.wait(); // here the lock is freed
            // here the lock is gained again
        }
    }
}
```

Listing 33 Java, Blocksynchronisation mit 'this'

In Diesem Fall wird kein separates Lock-Objekt erzeugt sondern auf der `this` Referenz synchronisiert. Dies bedeutet, dass die Threads den Lock- und Wait-Pool des `ObjectSync` Objektes verwenden. Im oben stehenden Code ist dies kein Nachteil. Wenn aber an mehreren Stellen immer auf dasselbe Objekt (z.B. `this`) synchronisiert wird, so kann dies unter Umständen zu verschlechterter Performance führen weil alle `synchronized`-Blöcke gleichzeitig gesperrt werden.

Da die Klassenvariabeln auf die zugegriffen wird in Java auch häufig Objekte sind bietet sich die direkte Synchronisation mit dem Datenobjekten an:

```
package ch.skybeam.examples;
public class ObjectSync {
    private Object object1 = new Object();
    private Object object2 = new Object();

    public void methodWithLock1() throws InterruptedException {
        // some code
        synchronized (this) {
            // some more code, Thread holds lock of 'lock' object
            object1.wait(); // here the lock is freed
            // here the lock is gained again
        }
    }
    public void methodWithLock2() throws InterruptedException {
        // some code
        synchronized (this) {
            // some more code, Thread holds lock of 'lock' object
            object2.wait(); // here the lock is freed
            // here the lock is gained again
        }
    }
}
```

Listing 34 Java, Locking über Klassenvariabeln

Dieser Code erlaubt den gleichzeitigen Zugriff auf `methodWithLock1()` und `methodWithLock2()` durch unterschiedliche Threads ohne dazu zu führen, dass diese sich gegenseitig blockieren.

8.2.1.2. Methodensynchronisation

In Java können auch ganze Methoden anstatt nur einzelne Blöcke synchronisiert werden. Dabei handelt es sich wie wir gleich sehen werden um den gleichen Mechanismus. Hier ein kleines Code-Beispiel:

```
package ch.skybeam.examples;
public class MethodSync {
    public synchronized void methodWithLock1() {
        // some code
    }

    public synchronized void methodWithLock2() {
        // some code
    }
}
```

Listing 35 Java, Methodensynchronisation

Wie gut zu erkennen ist wird das Schlüsselwort `synchronized` in der Methodendeklaration verwendet. Dies bewirkt, dass die gesamte Methode synchronisiert wird. Auffällig ist dabei die Tatsache, dass kein Synchronisations-Objekt angegeben wird. Bei der Methodensynchronisation verwendet Java implizit den Lock-Pool der `this` Referenz. Im Beispiel wäre dies die Instanz der Klasse `MethodSync`. Da es bei statischen Objekten keine `this` Referenz gibt wird in diesem Fall auf das dazugehörende `class` Objekt synchronisiert.

Der oben stehende Code könnte also auch folgendermassen geschrieben werden:

```
package ch.skybeam.examples;
public class MethodSync {
    public void methodWithLock1() {
        synchronized (this) {
            // some code
        }
    }

    public void methodWithLock2() {
        synchronized (this) {
            // some code
        }
    }
}
```

Listing 36 Java, Methoden und Blocksynchronisation

Der hauptsächliche Nachteil aus der Methodensynchronisation besteht in der Tatsache, dass alle Locks auf dasselbe Objekt stattfinden. Somit werden mit dem Eintritt eines Threads in eine synchronisierte Methode automatisch alle anderen synchronisierten Methoden für die anderen Threads gesperrt. Derjenige Thread, welcher den Lock besitzt, kann aber diesen mehrmals bekommen (siehe dazu auch Kapitel 8.2.1.3.

Aus diesem Grund sollte die Methodensynchronisation nur eingesetzt werden, wenn entweder der gleichzeitige Zugriff auf alle Datenfelder gesperrt werden muss oder nur ein einziges Datenfeld existiert. Ansonsten ist die Objektsynchronisation vorzuziehen da dadurch unter Umständen ein viel kleinerer Code-Block gesperrt werden kann.

8.2.1.3. Weitere wichtige Hinweise

Java führt intern einen Zähler auf Locks und eine Referenz auf den Thread, der den Lock aktuell besitzt. Ein Objekt gilt bei der Erzeugung als ungesperrt (engl. unlocked). Erhält ein Thread den Lock so wird seine Referenz im Objekt vermerkt und der Lock-Zähler um 1 erhöht. Das Objekt gilt nun als gesperrt. (engl. locked) Gibt der Thread den Lock ab, so wird der Zähler um 1 erniedrigt. Erreicht der Zähler 0, dann wird der Lock gelöscht und das Objekt gilt wieder als ungesperrt. Dabei kann ein Thread den Lock mehrmals erhalten. Dann wird einfach der Zähler erhöht. Besitzt ein Thread den Lock bereits so wird jeder weitere Lock auf dasselbe Objekt sofort erteilt (da ja niemand anders diesen aktuell besitzen kann) und der Zähler erhöht.

Locking ist allgemein „teuer“ und kostet gleich in zweierlei Hinsicht Zeit und somit Performance. Einerseits ist der Ein- und Austritt in synchronisierte Bereiche aufwändig zu regeln und andererseits müssen andere Threads warten während einer sich im synchronisierten Bereich befindet. Deshalb sollte nur dort synchronisiert werden wo dies auch wirklich nötig ist und die Synchronisations-Blöcke sollten möglichst klein gehalten werden. Beispielsweise macht es wenig Sinn die Methodensynchronisation für sehr grosse Methoden zu verwenden wenn diese aufwändigen Berechnungen durchführen aber nur am Ende der Methode auf ein zu schützendes Datenfeld zugreifen. Wird ein solches Datenfeld aber am Anfang der Methode gelesen und darf bis zum Ende der Berechnungen nicht modifiziert werden, dann muss es leider die gesamte Berechnungsdauer über gesperrt bleiben. Man könnte jetzt in Versuchung geraten jeweils nur ganz kurze Strecken zu synchronisieren. Dies wirkt sich aber manchmal auch negativ aus, da häufige Ein- und Austritte aus synchronisierten Bereichen sich auch auf die Performance auswirken. Für einige Beispiele siehe Kapitel 8.3.

8.2.2. Unterbrechbare Locks

Die gezeigte Lock-Behandlung mit `synchronized` Blöcken ist recht statisch und auch unflexibel. Ein Lock wird zu Beginn des `synchronized` Blockes angefragt und gehalten bis zum Ende des Blockes. Bei einem unerwarteten Programmabbruch (Exception) wird der Block verlassen und der Lock automatisch wieder freigegeben. In Manchen Fällen möchte man aber den Lock abhängig von einer Bedingung oder gar in einer aufgerufenen Methode freigeben. Es gibt auch Situationen bei denen der Lock angefragt werden soll ohne ewig darauf zu warten. Die Synchronisierung mit `synchronized` bietet Keine Möglichkeit der Definition einer Zeitüberschreitung bei der Lock-Anforderung. Ausserdem bietet `synchronized` keine Möglichkeit einen wartenden Thread zu unterbrechen.

Um diese Flexibilität zu bieten enthält die Java-API ab Version 5 die ReentrantLock Klasse (siehe Kapitel 8.1.3.1). Diese erlauben den Abbruch eines Threads, der auf einen Lock wartet. Ausserdem ist es damit möglich einen Zeitüberschreitung bei der Lock-Anfrage zu definieren sowie auch die Anfrage automatisch abbrechen zu lassen, wenn der Lock nicht sofort erteilt werden kann.

Weiterführende Informationen:

- Brian Goetz, Java Concurrency in Practice, ISBN 0-321-34960-1: [4]

8.2.3. Lock Granularität

Unter dem Begriff der Granularität wird verstanden wie fein gegliedert die Locks verwendet werden. Im Extremfall könnte in der gesamten Applikation ein einziges Lock-Objekt verwendet werden. Dies hätte aber den Nachteil, dass bei jedem synchronisierten Zugriff der alleinige Lock benötigt wird und gleichzeitig auch alle anderen kritischen Stellen gesperrt würden. Der andere Extremfall wäre, dass jedes Datenfeld durch seinen eigenen Lock geschützt würde. Dies hätte aber zum Nachteil, dass für viele Operationen mehrere Locks notwendig wären was die Wahrscheinlichkeit für Deadlocks erhöht. Ausserdem würde solch extrem fein Granuliertes Locking zu viel Overhead bei Lock-Anfragen und Lock-Freigaben führen.

Die optimale Granularität liegt irgendwo dazwischen und ist von Anwendungsfall zu Anwendungsfall unterschiedlich. Es gibt aber einige Techniken mit denen man die Granularität verfeinern kann falls dies nötig ist. Mehr dazu in den folgenden Abschnitten.

Weiterführende Informationen:

- Brian Goetz, Java Concurrency in Practice, ISBN 0-321-34960-1: [4]

8.2.3.1. Lock Splitting

Lock Splitting ist häufig die einfachste Art die Lock-Granularität zu verfeinern. Hierbei wird für zwei Objekte die dasselbe Lock-Objekt verwenden ein eigenes Lock-Objekt erzeugt. Durch die Aufteilung von einem auf zwei Locks kann im Optimalfall die lock contention für beide Objekte halbiert werden. Das Resultat sind weniger blockierende Zugriffe und einen dadurch gesteigerten Durchsatz.

Natürlich funktioniert Lock Splitting nur, wenn eine Operation nicht den Lock auf beide Objekte benötigt, ansonsten müsste dies Operation nach dem Split beide Locks anfordern. Wie bereits erwähnt kann Lock Splitting das Risiko für Deadlocks erhöhen.

8.2.3.2. Lock Striping und Lock Partitioning

Wie erwähnt stösst Lock Splitting an die Grenzen wenn entweder mehrere Objekte in eine Operation involviert sind oder ein einzelnes Datenfeld bereits durch seinen eigenen Lock geschützt ist. Beispielsweise kann auf den ersten Blick ein Zugriff auf ein Array nicht weiter unterteilt werden und muss durch ein einzelnes Lock-Objekt geschützt sein.

Einen Ausweg bietet Lock-Striping. Hierbei wird das Objekt selbst noch in weitere Lock-Bereiche unterteilt. Beispielsweise können Arrays in manchen Fällen in gleichgrosse Blöcke aufgeteilt werden. Im Extremfall erlaubt dies ein Locking auf Element-Ebene (wenn für jedes Element ein eigener Lock zur Verfügung steht.

Meistens ist Lock-Striping aber schwierig umzusetzen. Insbesondere verlangen eventuell einige Operationen wie Hash-Berechnungen den Lock auf alle Elemente. Diese Operationen würden dann sehr „teuer“ in der Ausführung und würden möglicherweise den Programmfluss empfindlich stören.

Der Betriff Lock-Partitioning wird häufig synonym verwendet und bezeichnet ebenfalls die Aufteilung eines Lock-Bereiches in mehrere Scheiben/Blöcke. Bei zweidimensionalen Datenstrukturen wird Lock-Partitioning auch häufig für die Aufteilung in Blöcke verwendet.

8.2.4. Compare and Swap / Compare and Set (CAS)

Alle bis jetzt kennen gelernten Methoden zur Synchronisierung basieren auf dem Prinzip des Ausschlusses und der Sperrung eines gewissen Bereiches durch einen Lock. Es gibt aber auch Methoden zur Modifikation gemeinsamer Daten ohne diese zu sperren. Dazu benötigt man so genannte atomare Funktionen. Atomare Funktionen haben die Eigenschaft, dass sie in einer Operation ohne Unterbrechung durchgeführt werden. Konkret heisst das, dass ein Thread, der eine Atomare Operation ausführt während dieser nicht unterbrochen werden kann. Somit ist auch keine Modifikation des Wertes während der Operation möglich.

Um eine solche Operation möglichst effizient umsetzen zu können benötigt man Hardwareunterstützung. Eine dieser Operationen ist die Compare-and-Swap Funktion. Sie wird sowohl von der IA32 wie auch von der Sparc-Architektur unterstützt. Die Funktion besitzt folgenden `Syntax: compareAndSwap(V, A, B)`. Hierbei ist V die Adresse des gespeicherten Wertes, A der erwartete Wert und B der neue Wert. Die Funktion vergleicht nun den Wert A mit dem tatsächlich gespeicherten Wert an der Adresse V. Sind die Werte V und A identisch, so wird V durch B ersetzt. Sind die Werte nicht identisch, dann wird nichts ausgeführt. Im Erfolgsfall gibt die Funktion den neuen Wert zurück, im Fehlerfall den alten.

In leicht abgewandelter Form mit den Namen Compare-and-Set findet das Konzept auch Anwendung in Java. Java verwendet intern dieselben Methoden aber erst mit Java 5 wurden die Methoden über die API zugänglich gemacht (siehe auch Kapitel 8.1.3.2).

CAS-Algorithmen sind nicht blockierend und eliminieren daher auch die Gefahr von Lock-Basierenden Deadlocks. Ausserdem entfällt die Synchronisation des Zugriffes. Ein typischer CAS-Algorithmus sieht wie folgt aus:

1. Lesen eines Originalwertes aus dem Speicher.
2. Berechnung des neuen Wertes basierend auf dem gelesenen Wert.
3. Zurückschreiben des neuen Wertes mittels CAS. Dabei können folgende Fälle eintreten:
    - Der Originalwert wurde nicht verändert und der neue Wert somit geschrieben. Der Algorithmus ist somit beendet.
    - Der Wert wurde zwischenzeitlich durch einen anderen Thread verändert. In diesem Fall wird wieder bei Schritt 1 weitergemacht.

Durch diesen Algorithmus sind konsistente Daten garantiert. Der Nachteil besteht in der Tatsache, dass bei stark konkurrierendem Zugriff der Algorithmus natürlich häufiger durchlaufen werden muss. Je länger der Algorithmus (Schritt 2) ist, desto wahrscheinlicher ist es, dass der Wert in der Zwischenzeit verändert wurde.

Die Vorteile überwiegen aber in den meisten Fällen. Insbesondere bei niedriger oder mittlerer Last (was einer typischen Applikationsauslastung entspricht) sind CAS-Algorithmen meist deutlich effizienter als Locks. In [4] S. 328 hat sich gezeigt, dass nur bei extremer Last ein `ReentrantLock` minimal schneller war als ein `AtomicInteger` (der auf CAS basiert, siehe dazu Kapitel 8.1.3.2). Bei niedriger und mittlerer lock contention war ein `AtomicInteger` deutlich schneller (ungefähr Faktor 2) und lag damit im Mittelfeld zwischen unsynchronisiertem und mit Lock gesichertem Zugriff.

Ein Riesiger Vorteil bei CAS-Algorithmen ist wie gesagt die Minimierung des Deadlock-Risikos. Selbst bei extremer Belastung ist mit CAS-Algorithmen sichergestellt, dass bei jedem Durchgang ein Thread weiterkommt.

Nachteilig wirkt sich insbesondere die schwierige Handhabung und Entwicklung von CAS-Algorithmen aus. Besonders beim Zugriff auf mehr als nur ein Datenelement muss die Konsistenz aller Datenfelder sichergestellt werden.

Weiterführende Informationen:

- Brian Goetz, Java Concurrency in Practice, ISBN 0-321-34960-1: [4]

8.3. Implementierung in Java

In diesem Kapitel werden anhand eines praktischen Beispieles einige Vor- und Nachteile der Synchronisierung sowie einige Fallstricke für Programmierer beleuchtet. Die Beispiele konzentrieren sich auf eine Analyse der Performance und sollen ein Gefühl für die Wichtigkeit korrekter Synchronisierung vermitteln.

8.3.1. Methodensynchronisation

```
package ch.skybeam.examples;
public class MethodSync {
    private Object A;
    private Object B;

    public synchronized void methodWithLock1() {
        // some code accessing A only
    }

    public synchronized void methodWithLock2() {
        // some code accessing B only
    }
}
```

Listing 37 Java, Methodensynchronisation (grobes Locking)

Dieser Code würde beim Aufruf von `methodWithLock1()` implizit das Objekt `MethodSync` für einen Thread sperren. Somit wären auch Zugriffe auf `methodWithLock2()` gesperrt obwohl diese nicht auf dieselben Datenfelder zugreifen.

Besser wäre:

```
package ch.skybeam.examples;
public class MethodSync {
    private Object A;
    private Object B;

    public void methodWithLock1() {
        synchronized(A) {
            // some code accessing A only
        }
    }

    public void methodWithLock2() {
        synchronized(B) {
            // some code accessing B only
        }
    }
}
```

Listing 38 Java, verfeinertes Locking)

Bei diesem Beispiel werden die Methoden unabhängig voneinander aufgerufen. Somit würden sich zwei Threads, welche eine Referenz auf dasselbe Objekt haben nicht blockieren sofern sie nicht dieselbe Methode aufrufen.

8.3.2. Überlange Synchronisierung

```
package ch.skybeam.examples;
public class MethodSync {
    private int A;
    public synchronized void methodWithLock1() {
        // lots of slow code
        // even more slow code

        // incrementing shared counter
        A++;
    }
}
```

Listing 39 Java, überlange Synchronisierung

Hier wird das Objekt (`this` Referenz) für die gesamte Berechnungsdauer gesperrt. Dies kann unter Umständen sehr lange dauern. Aufgrund der Methodensynchronisation könnte auch keine weitere `synchronized` Methode dieser Klasse durch einen anderen Thread aufgerufen werden. Dies kann je nach Design der Anwendung zur Blockierung anderer Applikationsteile führen.

Besser wäre hier:

```
package ch.skybeam.examples;
public class MethodSync {
    private int A;
    public void methodWithLock1() {
        // lots of slow code
        // even more slow code

        // incrementing shared counter
        synchronized(this) {
            A++;
        }
    }
}
```

Listing 40 Java, Synchronisiertung verkürzen

Hierbei wird nur der Zugriff auf die gemeinsame Variable `A` synchronisiert. Zu beachten ist auch, dass die Methode `methodWithLock1()` hier nicht mehr synchronisiert ist.

Das Funktioniert natürlich nicht, wenn die Variable `A` in der Berechnung verwendet würde und während der gesamten Berechnung nicht verändert werden darf:

```
package ch.skybeam.examples;
public class MethodSync {
    private int A;
    public synchronized void methodWithLock1() {
        int tmp = A;
        // lots of slow code
        // even more slow code
        int result = ...;

        // incrementing shared counter
        A = result;
    }
}
```

Listing 41 Java, Synchronisierung verkürzen 2

Hier könnte allenfalls eine Synchronisierung auf ein anderes Objekt als die `this` Referenz geprüft werden um nicht gleich auch alle anderen `synchronized` Methoden dieses Objektes zu sperren.

8.3.3. Extrem häufiges Locking/Unlocking

Folgende Klasse bietet einen sicheren (synchronisierten) Zugriff auf einen Integer-Wert:

```
package ch.skybeam.examples;
public class MethodSync {
    private int A = 0;

    public synchronized int getA() {
        return A;
    }

    public synchronized void setA(int newA) {
        this.A = newA;
    }
}
```

Listing 42 Java, häufiges Locking/Unlocking

Hier macht die Methodensynchronisation sogar Sinn, da nur auf ein Datenfeld zugegriffen wird und dieses über dasselbe Objekt (`this`) gesperrt werden kann.

Nun wird über den folgenden Code auf das Element zugegriffen:

```
MethodSync perfTest = new MethodSync();
long startTime = System.nanoTime();

for(int i=0; i<100000000; i++) {
    perfTest.setA(perfTest.getA()+1);
}

long timeSpent = System.nanoTime() - startTime;
System.out.println("Value of perfTest: " + perfTest.getA());
System.out.println("Time spent: " + timeSpent/1000000 + "ms");
```

Listing 43 Java, Loking in einer Schleife

Wird dieser Code ausgeführt, so wird bei jedem Schleifendurchgang zwei Mal der Lock auf das `perfTest` Objekt angefordert. Sowohl `setA()` als auch `getA()` sind synchronisiert und verlangen daher den Lock. Hier geschieht beides 100 Millionen Mal.

Die Ausgabe sieht auf meinem System wie folgt aus:

```
Value of perfTest: 100000000
Time spent: 2559ms
```

Listing 44 Ausgabe

Versuchen wir mal folgenden Code für den Objektzugriff:

```
MethodSync perfTest = new MethodSync();
long startTime = System.nanoTime();
synchronized (perfTest) {
    for(int i=0; i<100000000; i++) {
        perfTest.setA(perfTest.getA()+1);
    }
}
long timeSpent = System.nanoTime() - startTime;
System.out.println("Value of perfTest: " + perfTest.getA());
System.out.println("Time spent: " + timeSpent/1000000 + "ms");
```

Listing 45 Java, Synchronisierung ausserhalb der Schleife

Hier wird der Lock für das `perfTest` Objekt schon vor der Schleife angefordert. Es ist also anzunehmen, dass der Aufwand des Lockings gegen Null geht. Erstaunlicherweise sieht die Ausgabe wie folgt aus:

```
Value of perfTest: 100000000
Time spent: 2386ms
```

Also keine ausserhalb der Messgenauigkeit liegende Veränderung. Durch das Ergebnis ist man geneigt zu glauben, dass Locking offenbar doch nicht so viel Zeit in Anspruch nimmt. Doch Vorsicht: Wir arbeiten hier nur mit einem Thread. Java muss den Lock eines Objektes erst wirklich abgeben und neu anfordern wenn ein Kontextwechsel stattfindet. Offenbar wird hier von der Java VM optimiert. Um das zu beweisen modifizieren wir den Zugriff leicht und erzeugen einen dummy-Thread. Dessen einzige Aufgabe ist es hie und da mal auf das Objekt zuzugreifen und Java dadurch zu zwingen den Lock abzugeben:

```
MethodSync perfTest = new MethodSync();
Thread dummy = new DummyThread(perfTest);
dummy.start();
long startTime = System.nanoTime();
// synchronized (perfTest) {
    for(int i=0; i<100000000; i++) {
        perfTest.setA(perfTest.getA()+1);
    }
// }
long timeSpent = System.nanoTime() - startTime;
System.out.println("Value of perfTest: " + perfTest.getA());
System.out.println("Time spent: " + timeSpent/1000000 + "ms");
```

Listing 46 Java, Locking in einer Schleife

Wie zu sehen ist wurde die Synchronisierung vor der Schleife hier auskommentiert. Dafür wird ein Thread erzeugt. Dessen Code sieht folgendermassen aus:

```
package ch.skybeam.examples;
public class DummyThread extends Thread {
    private MethodSync sync;

    public DummyThread(MethodSync s) {
        this.sync = s;
        this.setDaemon(true);
    }

    public void run() {
        int count = 0;
        while (true) {
            System.out.print("Thread tick " + ++count);
            System.out.print(" reads value: " + sync.getA());
            System.out.println("");
            try {
                Thread.sleep(500);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
```

Listing 47 Java, Dummy-Thread zur Simulation von 'lock contention'

Wie gut zu erkennen ist bekommt dieser Thread eine Referenz auf das synchronisierte Objekt. Ansonsten tut der Thread nichts ausser 500ms warten um dann einmal lesend auf das Objekt zuzugreifen. Da dieses vorher ungefähr 2.5 Sekunden lief sollte dies nur ungefähr 5 Mal passieren. Den zusätzlichen Aufwand durch den Kontextwechsel selbst und den Lesezugriff (nur ca. 5 Mal) vernachlässigen wir hier. Der Thread wird im Übrigen als Daemon konfiguriert. Damit beendet sich die Endlosschlaufe automatisch wenn das Programm zu Ende ist. Ohne diese Konfiguration würde sich das Programm am Ende nicht beenden.

Das Ergebnis überrascht dann aber mit erstaunlich langer Laufzeit. Hier die Ausgabe (gekürzt):

```
Thread tick 1 reads value: 584218
Thread tick 2 reads value: 21971701
```

```
Thread tick 3 reads value: 44552450
Thread tick 4 reads value: 45991802
[…]
Thread tick 39 reads value: 98295964
Thread tick 40 reads value: 99761596
Value of perfTest: 100000000
Time spent: 19862ms
```

Listing 48 Ausgabe

Das Programm braucht also plötzlich etwa 8 Mal länger als erwartet. Was ist passiert? Am zusätzlichen Aufwand für den Thread liegt es nicht. Wir kommentieren mal eine der Ausgabezeilen aus:

```
// System.out.print(" reads value: " + sync.getA());
```

Listing 49 Java, Lock-verursachende Zeile entfernen

Und schon sieht das Ergebnis wieder wie vorher aus:

```
Thread tick 1
Thread tick 2
Thread tick 3
Thread tick 4
Thread tick 5
Value of perfTest: 100000000
Time spent: 2330ms
```

Listing 50 Ausgabe

Wir haben also vorhin die VM wirklich dazu gezwungen den Lock häufiger abzugeben und dadurch entstand die längere Laufzeit. Das Ergebnis verschlechtert sich noch weiter, wenn nicht alle 500ms sondern in kürzeren Abständen zugegriffen wird. Doch selbst bei 40 konkurrierenden Zugriffen wie oben entstand ein Performance-Einbruch von Faktor 8. Es ist anzunehmen, dass die VM so intelligent ist den Lock so lange zu behalten wie der Thread mit der `for` Schleife im Status „running“ ist.

Was kann man dagegen tun?

Eine Versuche wäre es wert den Lock während der gesamten `for`-Schleife zu behalten indem man ihn manuell (durch `synchronized`) anfordert:

```
MethodSync perfTest = new MethodSync();
Thread dummy = new DummyThread(perfTest);
dummy.start();
long startTime = System.nanoTime();
synchronized (perfTest) {
    for(int i=0; i<100000000; i++) {
        perfTest.setA(perfTest.getA()+1);
    }
}
long timeSpent = System.nanoTime() - startTime;
System.out.println("Value of perfTest: " + perfTest.getA());
System.out.println("Time spent: " + timeSpent/1000000 + "ms");
```

Listing 51 Java, Synchronisation ausserhalb der Schleife

Leider führt dies auch nicht zum erwünschten Ergebnis:

```
Threat tick 1 reads value: 100000000
Value of perfTest: 100000000
Time spent: 19740ms
```

Listing 52 Ausgabe

Der einzig sichtbare Effekt ist, dass der Thread über den Gesamten Schleifen-Vorgang angehalten wird weil er nie den Lock für das `perfTest` Objekt erhält. Das hält die JVM aber nicht davon ab hie und da einen Kontextwechsel vorzunehmen und zu prüfen, ob der Lock nun zu haben sei (was natürlich Zeit kostet).

Dieser Code wäre aber insgesamt sicherer, weil es auch möglich wäre, dass zwischen den Methodenaufrufen `perfTest.getA()` und `perfTest.setA()` ein Kontextwechsel stattfindet und der nun ablaufende Thread dazwischen A aus der klasse `MethodSync` verändert. In diesem Fall würde das anschliessende `perfTest.setA()` den Wert schlicht wieder ersetzen. Durch die Synchronisierung wird dies verhindert da während der gesamten Verarbeitung der Lock an `perfTest` gehalten wird.

Eine Möglichkeit zur Performance-Steigerung wäre hier unter Umständen der unsynchronisierte Zugriff durch den Thread. Dies beinhaltet aber weitere Gefahren. Siehe dazu den nächsten Abschnitt.

8.3.4. Teilweise unsynchronisierter Zugriff

Angenommen unser Thread aus der vorherigen Aufgabe ist nicht auf die Synchronisierung angewiesen sondern will lediglich einen gültigen Wert erhalten (egal ob dieser gerade verändert wird). Dazu implementieren wir eine weitere Methode in der Klasse `MethodSync`:

```
package ch.skybeam.examples;
public class MethodSync {
    private int A = 0;

    public synchronized int getA() {
            return A;
    }

    public synchronized void setA(int newA) {
            this.A = newA;
    }

    public int getAsyncA() {
        return A;
    }
}
```

Listing 53 Java, teilweise unsynchronisierter Zugriff

Die Methode `getAsyncA()` gibt den aktuellen Wert von A zurück ohne einen Lock zu verwenden.

Dazu passend wird der Thread-Code aktualisiert:

```
package ch.skybeam.examples;
public class DummyThread extends Thread {
    private MethodSync sync;

    public DummyThread(MethodSync s) {
        this.sync = s;
        this.setDaemon(true);
    }

    public void run() {
        int count = 0;
        while (true) {
            System.out.print("Thread tick " + ++count);
            System.out.print(" reads value: " + sync.getAsyncA());
            System.out.println("");
            try {
                Thread.sleep(100);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

Listing 54 Java, Dummy Thread liesst ohne Locking

Das Hauptprogramm bleibt unverändert:

```
MethodSync perfTest = new MethodSync();
Thread dummy = new DummyThread(perfTest);
dummy.start();
long startTime = System.nanoTime();
synchronized(perfTest) {
    for(int i=0; i<100000000; i++) {
```

```
        perfTest.setA(perfTest.getA()+1);
    }
}
long timeSpent = System.nanoTime() - startTime;
System.out.println("Value of perfTest: " + perfTest.getA());
System.out.println("Time spent: " + timeSpent/1000000 + "ms");
```

Listing 55 Java, Unverändertes Hauptprogramm

Die Ausgabe sieht wie folgt aus (gekürzt):

```
Thread tick 1 reads value: 1373453
Thread tick 2 reads value: 6793724
[…]
Thread tick 19 reads value: 96867749
Value of perfTest: 100000000
Time spent: 2337ms
```

Listing 56 Ausgabe

Es sieht also so aus als würde der konkurrierende Zugriff nicht mehr bremsend ins Gewicht fallen und das obwohl der Thread jetzt alle 100ms eine Abfrage macht und nicht mehr nur alle 500ms.

Hier ist aber Vorsicht geboten. In diesem Beispiel sind wir davon ausgegangen, dass die neue Methode `getAsyncA()` zwar nicht synchronisiert ist aber trotzdem immer einen gültigen Wert liefert. Dies trifft nicht immer zu. In der VM Spezifikation steht nicht, dass 64-bit Werte atomar aktualisiert werden müssen. Für 32-bit Werte trifft dies zu, da alle heutigen Prozessoren atomare 32-bit Operatoren verwenden. Bei der Verwendung von 64-bi Datentypen (`long`, `double`) kann es vorkommen, dass die ersten 32-bit der Variable vom neuen und die zweiten 32-bit der Variable vom alten Wert stammen was möglicherweise einem total ungültigen Wert entspricht. Um dies zu verhindern bietet Java das Schlüsselwort `volatile`. Also `volatile` gekennzeichnete Datentypen werden quasi Atomar aktualisiert und nicht in lokalen Caches gehalten. Somit wird sichergestellt, dass bei jedem Lesezugriff ein gültiger und aktueller (nicht im Cache vorhandener) Wert geschrieben wird. Leider kostet die natürlich auch wieder etwas Performance (aber nicht so viel wie die Synchronisierung):

```
package ch.skybeam.examples;
public class MethodSync {
    private volatile int A = 0;

    public synchronized int getA() {
            return A;
    }

    public synchronized void setA(int newA) {
            this.A = newA;
    }

    public int getAsyncA() {
        return A;
    }
}
```

Listing 57 Java, volatile Schlüsselwort

Die Ausgabe sieht dann wie folgt aus:

```
Thread tick 1 reads value: 1175145
[…]
Thread tick 19 reads value: 96067839
Value of perfTest: 100000000
Time spent: 2351ms
```

Listing 58 Ausgabe

Das Beispiel zeigt uns eindrucksvoll, dass die Synchronisierung mit Bedacht eingesetzt werden muss. Es sollte generell nur da synchronisiert werden wo es auch wirklich nötig ist. In unserem Beispiel ist es nicht zwingend notwendig, dass der parallel laufende Thread synchronisiert Zugreifen muss.

Allerdings ist es in diesem Beispiel auch nicht nötig den Zugriff auf das `MethodSync` Objekt zu synchronisieren da nur unser „main“ Thread schreiben darauf zugreift. Deshalb wäre hier sogar ein unsynchronisierter Zugriff denkbar (siehe nächster Abschnitt).

8.3.5. Vollständig unsynchronisierter Zugriff

Im Beispiel von oben wurde der Zugriff aus dem Thread unsynchronisiert behandelt. Allerdings ist hier eigentlich gar keine Synchronisation mehr notwendig weil sowieso nur ein einziger Thread (der „main“ Thread) auf das Objekt zugreift. Somit entfernen wir die Synchronisation mal komplett:

```
package ch.skybeam.examples;
public class MethodSync {
    private volatile int A = 0;

    public int getA() {
            return A;
    }

    public void setA(int newA) {
            this.A = newA;
    }
}
```

Listing 59 Java, unsynchronisierter Zugriff

Der Thread kann nun ebenfalls wieder die nun unsynchronisierte Methode `getA()` verwenden.

```
package ch.skybeam.examples;
public class DummyThread extends Thread {
    private MethodSync sync;

    public DummyThread(MethodSync s) {
        this.sync = s;
        this.setDaemon(true);
    }

    public void run() {
        int count = 0;
        while (true) {
            System.out.print("Thread tick " + ++count);
            System.out.print(" reads value: " + sync.getA());
            System.out.println("");
            try {
                Thread.sleep(100);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

Listing 60 Java, Dummy Thread verwendet unsynchronisierte Methoden

Die Ausführungs-Methode sieht nun wie folgt aus:

```
MethodSync perfTest = new MethodSync();
Thread dummy = new DummyThread(perfTest);
dummy.start();
long startTime = System.nanoTime();
    for(int i=0; i<100000000; i++) {
        perfTest.setA(perfTest.getA()+1);
    }
long timeSpent = System.nanoTime() - startTime;
System.out.println("Value of perfTest: " + perfTest.getA());
System.out.println("Time spent: " + timeSpent/1000000 + "ms");
```

Listing 61 Java, Haupt-Thread

Die Ausgabe bescheinigt uns einen gesunkenen Overhead:

```
Thread tick 1 reads value: 10275874
Thread tick 2 reads value: 38403126
Thread tick 3 reads value: 68075974
Thread tick 4 reads value: 91267779
Value of perfTest: 100000000
Time spent: 439ms
```

Listing 62 Ausgabe

Der Nachteil dieser Methode liegt jetzt darin, dass keine Methode des Objektes `perfTest` synchronisiert ist. In unserem Code ist dies aber gar nicht notwendig, da gar niemand Zugriff auf diese Referenz erhalten kann. Nur der Thread bekommt die Referenz mitgegeben, dessen Code kenne ich aber und kann somit sicherstellen, dass keine konkurrierende Modifikation entstehen kann.

Beachtlich ist hierbei, dass durch die Entfernung der Synchronisation nochmals eine um Faktor 5 höhere Geschwindigkeit erzielt werden konnte.

Angenommen es gäbe mehrere Codestellen (beispielsweise noch eine innerhalb des Threads), die schreibend auf das `perfTest` Objekt zugreifen müssen, dann könnten alle Stellen durch eine manuelle Objektsynchronisation Thread-Safe gemacht werden. Beispielsweise in unserem Hauptprogramm:

```
MethodSync perfTest = new MethodSync();
Thread dummy = new DummyThread(perfTest);
dummy.start();
long startTime = System.nanoTime();
synchronized(perfTest) {
    for(int i=0; i<100000000; i++) {
        perfTest.setA(perfTest.getA()+1);
    }
}
long timeSpent = System.nanoTime() - startTime;
System.out.println("Value of perfTest: " + perfTest.getA());
System.out.println("Time spent: " + timeSpent/1000000 + "ms");
```

Listing 63 Java, manuelle Synchronisation aller relevanten Stellen

Der lesende Zugriff im Thread-Code braucht nicht unbedingt synchronisiert zu werden. Insbesondere haben wir die Variable schon mit `volatile` gekennzeichnet. Deswegen wir auf jeden Fall ein gültiger Wert ausgelesen.

Die Ausgabe sieht nun wie folgt aus:

```
Thread tick 1 reads value: 6815157
Thread tick 2 reads value: 32033000
Thread tick 3 reads value: 57297691
Thread tick 4 reads value: 82176413
Value of perfTest: 100000000
Time spent: 492ms
Thread tick 5 reads value: 100000000
```

Listing 64 Ausgabe

Und ist somit nur unwesentlich langsamer.

Wie oben erwähnt wird der lesende Zugriff innerhalb des Threads nicht synchronisiert. Tun wir dies Trotzdem, dann fällt auf, dass der Thread nur ein einziges Mal durch die Schleife läuft und entweder 0 oder 100000000 ausgibt. Dies liegt daran, dass die gesamte Schleife über der Lock für das `perfTest` Objekt gehalten wird und der Thread somit gar nie den Lock bekommen könnte. Ein Kompromiss würde daher die Synchronisation im innern der `for` Schleife darstellen. Dies würde dem Thread pro Schleifendurchgang einmal die Möglichkeit geben den Lock zu bekommen:

```
MethodSync perfTest = new MethodSync();
Thread dummy = new DummyThread(perfTest);
dummy.start();
long startTime = System.nanoTime();
for(int i=0; i<100000000; i++) {
```

```
    synchronized(perfTest) {
        perfTest.setA(perfTest.getA()+1);
    }
}
long timeSpent = System.nanoTime() - startTime;
System.out.println("Value of perfTest: " + perfTest.getA());
System.out.println("Time spent: " + timeSpent/1000000 + "ms");
```

Listing 65 Java, Lock abgeben

Der Thread kann folgendermassen angepasst werden:

```
package ch.skybeam.examples;
public class DummyThread extends Thread {
    private MethodSync sync;

    public DummyThread(MethodSync s) {
        this.sync = s;
        this.setDaemon(true);
    }

    public void run() {
        int count = 0;
        while (true) {
            System.out.print("Thread tick " + ++count);
            synchronized(sync) {
                System.out.print(" reads value: " + sync.getA());
            }
            System.out.println("");
            try {
                Thread.sleep(100);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

Listing 66 Java, synchronisierter Java Thread

Somit muss der Thread für den lesenden Zugriff jetzt auch den Lock bekommen. Leider hat diese Änderung bereits deutliche Auswirkungen auf die Performance:

```
Thread tick 1 reads value: 2351280
Thread tick 2 reads value: 14515341
Thread tick 3 reads value: 26834097
Thread tick 4 reads value: 38733872
Thread tick 5 reads value: 51882091
Thread tick 6 reads value: 65153897
Thread tick 7 reads value: 78421122
Thread tick 8 reads value: 90772255
Value of perfTest: 100000000
Time spent: 1006ms
Thread tick 9 reads value: 100000000
```

Listing 67 Ausgabe

Der Grund liegt hier in der höheren Anzahl von Lock-Wechseln. Der main Thread gibt den Lock maximal einmal pro Schleifendurchgang ab was dem Thread ermöglicht diesen zu bekommen.

Abschliessend kann gesagt werden, dass Synchronisierung nur da eingesetzt werden soll wo es wirklich nötig ist. In unserem Beispiel verlässt unser `perfTest` Objekt unseren eigenen Code nie und kann auch nicht von aussen modifiziert werden. Somit können wir schon über den Code sicherstellen, dass kein konkurrierender Zugriff stattfindet und die Synchronisation der Datenfelder kann entfallen.

Wird das Objekt allerdings veröffentlicht so kann nicht mehr sichergestellt werden, dass kein Fremd-Code konkurrierend darauf zugreift.

8.4. JVM Optimierung

Die Java-VM stell das Bindeglied zwischen der Programmiersprache Java und dem Betriebssystem dar. Da Java Programme plattformunabhängig sind laufen diese auf jeder Plattform auf der eine Java VM verfügbar ist. Diese wiederum muss natürlich die Funktionalitäten der Sprache Java auf die vom Betriebssystem zur Verfügung gestellten Funktionalitäten abbilden. Da unterschiedliche Betriebssysteme und unterschiedliche Hardware-Architekturen sich teilweise stark unterscheiden oder gar gegenteilige Konzepte verfolgen ist dies eine nicht-triviale Aufgabe.

Als Beispiel sei eine JVM Implementierung auf einem Betriebssystem ohne Kernel-Level Threads genannt. Der Java-Programmierer muss in seinem Programm trotzdem den vollen Sprachumfang inklusive Threads nutzen können. Es ist nun die Aufgabe der JVM die vom Programmierer genutzte Funktionalität möglichst effizient an die zur Verfügung gestellte Hardware und das Betriebssystem anzupassen. Wenn das Betriebssystem keine Threads unterstützt kann die JVM diese Aufgabe übernehmen und die Threads quasi in Software emulieren. Dazu implementiert die JVM einen Scheduler und teilt die zur Verfügung stehende Rechenzeit im Zeitscheiben-Verfahren (oder mittels einer anderen Scheduling-Strategie) auf die Threads auf. Dies ist natürlich sehr ineffizient da die JVM hier nur diejenige Zeit an die Threads verteilen kann, die dem Prozess der virtuellen Maschine vom Betriebssystem zugewiesen wird. Ausserdem ist eine Verteilung auf mehrere Prozessoren somit unmöglich.

Das Beispiel zeigt deutlich, dass die Performance einer Java-Applikation von mehr abhängt als der effizienten Programmierung der Java-Routinen. Vielmehr wird ein effizientes Zusammenspiel von Java-Applikation, Java Virtual Machine (JVM), Betriebssystem und Hardware benötigt. Ein Entwickler einer JVM ist natürlich darauf bedacht die Möglichkeiten einer Plattform möglichst effizient auszuschöpfen. Die Optimierung der JVM ist eine Aufgabe, die selbst Sun Microsystems (als Schöpfer von Java) noch nicht abschliessend gelöst hat. Seit Java 1.0 in den frühen 90er Jahren vorgestellt wurde hat Java in Sachen Performance einen gewaltigen Sprung nach vorne gemacht. Trotzdem haftet Java noch das Image an extrem langsam und Ressourcenhungrig zu sein.

Dieses Kapitel gibt einen Überblick über einige der wichtigsten Optimierungen der Java VM und wo man eventuell noch selber etwas „drehen“ kann.

8.4.1. Just In Time (JIT) Compiler

Java ist prinzipiell eine interpretierte Sprache. Der Sourcecode wird durch den Java-Compiler in den so genannten Bytecode überführt. Dabei handelt es sich im Sinne der Plattformunabhängigkeit nicht um Hardwareabhängigen Binärcode sondern um eine Art Zwischenstufe. Die Java-Runtime interpretiert dann diesen Bytecode um das Programm auszuführen.

Um dies zu beschleunigen wird häufig genutzter Code zur Laufzeit im Hintergrund Kompiliert. Bei erneuten Aufrufen derselben Methode/Klasse wird dann auf den Hardwarenahen, kompilierten Code zurückgegriffen.

Diese Methode bietet sogar Vorteile gegenüber statisch kompiliertem Code wie C/C++. Je nach Verwendung der Klassen kann die kompilierte Variante mit Optimierungen re-kompiliert werden um eine bessere Leistung zu erzielen.

Die Sun HotSpot VM erlaubt die Ausgabe des Kompilierungs-Status während der Ausführung durch die Angabe folgender Kommandozeilenoption beim Start:

```
-XX:+PrintCompilation
```

Listing 68 Java, JIT PrintCompilation

Durch die Angabe der Folgenden Option kann der JIT Compiler deaktiviert werden. In diesem Fall läuft die VM dann im Interpreted-Mode:

```
-Xint
```

Listing 69 Java, JIT deaktivieren

8.4.2. Thread-Modelle

Wie bereits erwähnt ist die JVM zuständig für die Verwaltung der Threads und deren Abbildung auf Betriebssystemebene. Für Solaris ist bei Sun Microsystems eine Dokumentation des Thread-Modelles in der Dokumentation zu finden (siehe [HOTSPOTTHR]). Leider ist dieselbe Information für Windows nicht verfügbar. Die Dokumentation zeigt auch, dass hier viele Versuche zur Optimierung gemacht wurden. Unter Solaris 8 gibt es in Verbindung mit der Java HotSpot Runtime in der Version 1.4 ganze 4 Methoden der Thread-Abbildung:

- Many-to-Many, thread based synchronisation
- Many-to-Many, LWP based synchronisation
- One-to-One via Bound threads
- One-to-One via Alternate Threads library

Offensichtlich hat sich die letzte Variante durchgesetzt. Unter Solaris 9 ist nämlich nur noch die „One-to-One via alternate Threads library“ Methode verfügbar.

8.4.3. Garbage Collection

Beim Programmieren mit Java muss sich der Entwickler nicht selber um die Allokation und die Freigabe von Speicher kümmern. Dies wird vollumfänglich vom Garbage Collector (GC) übernommen. Vereinfacht gesagt überprüft Java zur Laufzeit in unregelmässigen Abständen (abhängig beispielsweise vom freien Speicher, der aktuellen Systemlast usw.) welche Objekte entfernt werden können um Speicher freizugeben. Hierbei arbeitet Java mit einem Referenzzähler. Objekte, die nirgends mehr referenziert sind können auch nicht mehr verwendet werden und können somit entfernt werden. Bis zur Version 1.3.1 unterstützte Java keine parallele Garbage Collection. Dies führte zu massiven Performance-Einbrüchen während die Garbage Collection durchgeführt wird.

Um die Performance zu verbessern wurden Methoden wie die Segmentierung des Speichers eingeführt. Da die meisten Objekte nur eine kurze Lebensdauer haben werden diese in einem für „junge“ Objekte reservierten Speicherbereich abgelegt (genannt „Eden“). Dort findet auch eine häufigere Garbage Collection statt. Überleben die Objekte lange genug kommen diese in den „Tenured“ Bereich. Im „Permanenten“ Bereich werden allgemeine Datenstrukturen, Klassen- und Methodenbeschreibungen abgelegt.

In Java 5 kann zwischen verschiedenen Garbage Collectoren gewählt werden. Welcher der beste von ihnen ist hängt von Verschiedenen Faktoren ab und kann häufig nur durch erweiterte Tests bestimmt werden.

Die folgende Option aktiviert beispielsweise einen inkrementellen GC. Dieser erledigt den Grossteil seiner Arbeit ohne die Applikation zu beeinflussen (parallel):

```
-Xincgc
Oder (selber Effekt):
-XX:+UseConcMarkSweepGC
```

Listing 70 Java, Incremental Garbage Collection

Um Mehr Informationen über die Vorgänge innerhalb der JVM zu erhalten können folgende Optionen verwendet werden:

```
-XX:+PrintGCDetails
-XX:+PrintGCTimeStamps
```

Listing 71 Java, Debug Garbage Collection

Der folgende Parameter definiert die gewünschte Garbage Collection Laufzeit im Verhältnis zur Anwendungs-Laufzeit:

```
-XX:GCTimeRatio=<nnn>
```

Listing 72 Java, Garbage Collection Ratio

Wobei die Angabe prozentual erfolgt nach der Formel (1/(1+<nnn>)). Für nnn=19 resultiert also eine Zuweisung von 5% für die Garbage Collection. Standardmässig ist 1% eingestellt.

Nur am Rande mit der Garbage Collection verbunden sind die Einstellungen für die Heap-Grösse. Standardmässig gelten eine Minimalgrösse von 3.5MB und eine Maximalgrösse von 64MB für den Heap. Insbesondere die Maximalgrösse reicht bei Server-Anwendungen schnell nicht mehr aus und sollte erweitert werden. Ein Grösserer Wert bedeutet hier nicht, dass die VM mehr Speicher belegen wird aber dass sie das kann falls nötig:

```
-Xms
-Xmx
Biespielsweise:
-Xms64M
-Xmx512M
```

Listing 73 Java, Heap Grösse

Das Beispiel würde die Minimalgrösse auf 64MB und die Maximalgrösse auf 512MB begrenzen.

Zwei weitere Parameter beeinflussen die Allozierung bzw. De-Allozierung von Speicher durch die VM:

```
-XX:MinHeapFreeRatio=40
-XX:MaxHeapFreeRatio=70
```

Listing 74 Java, Heap Grössenverhältnis

Die VM versucht mit diesen Zahlen den Anteil des freien Speichers zwischen 40% und 70% zu halten. Dies bedeutet, dass bei steigendem Speicherbedarf der Anwendung der Anteil unter 40% sinkt und dadurch neuer Speicher alloziert wird. Umgekehrt wird der Heap wieder verkleinert wenn der Anteil freien Speichers über 70% wächst. Diese Grössenänderungen sind natürlich auch aufwändig. Insbesondere bei grossen Anwendungen reicht die Initialgrösse von 3.5MB nicht und es muss häufig nachalloziert werden. Um dies zu vermindern ist es ratsam bei Anwendungen mit hohem Speicherbedarf den `-Xms` Parameter zu verwenden und möglicherweise gleich mit `-Xmx` ein höheres Oberlimit zu setzen.

Im Zusammenhang mit der Garbage Collection spielt auch ein weiterer Aspekt eine wichtige Rolle: Die Hardware-Architektur. Auf NUMA/ccNUMA Systemen ist es von (eventuell entscheidendem) Vorteil wenn die Objekte im lokalen Speicher des ausführenden Prozessors liegen (siehe Kapitel 5.2.1). Java bietet zumindest im Moment auf API-Ebene keine Möglichkeit auf die Speicherverwaltung Einfluss zu nehmen. Dies würde angesichts der Plattformunabhängigkeit auch kaum Sinn machen. Aus unserer Sicht muss längerfristig die Java VM selber dafür sorgen, dass die Objekte im lokalen Speicher des Prozessors liegen. Betriebssysteme wie MS Windows bieten dazu bereits diverse Affinitäts-Optionen um Threads an Prozessoren oder Prozessor-Gruppen zu binden. Wir gehen davon aus, dass wahrscheinlich der Garbage Collector auf lange Sicht die Aufgabe der Speicher-Relozierung übernehmen wird. Dazu bräuchte der Garbage Collector nicht mal mehr viele zusätzliche Informationen. Im Moment wertet dieser aus ob ein Objekt einen Referenzzähler ungleich null hat um zu entscheiden, ob das Objekt noch benötigt wird. Wüsste der GC jetzt welcher Thread hauptsächlich auf das Objekt zugreift so könnte er dieses in den Lokalen Speicher desjenigen Prozessors verschieben auf dem dieser Thread ausgeführt wird.

In die Selbe Richtung geht das Dokument von Mustafa M. Tikir mit dem Titel „NUMA-Aware Java Heaps for Server Applications“. Offenbar wurden hier bereits Messungen und Modifikationen in diesem Bereich gemacht. Der Vorschlag aus dem Dokument ist es den Heap nicht nur in junge und alte Objekte aufzuteilen sondern diese wiederum in mehrere Prozessor-Lokale Bereiche. Da die meisten Objekte bereits jung wieder sterben macht dort eine Verschiebung kaum Sinn. Viel mehr Sinn macht es diese gleich im Richtigen Heap-Bereich zu erzeugen (lokal zum erzeugenden Thread). Überlebt das Objekt den „Eden“ Zyklus und kommt in den Tenued-Bereich so kann der Garbage Collector von Zeit zu Zeit prüfen ob das Objekt noch im lokalen Speicher des hauptsächlich zugreifenden Threads liegt und dieses bei Bedarf verschieben. Da die Objekte dann schon länger existieren werden sie nicht mehr so häufig Verschoben was einer nur unwesentlich erhöhten Belastung der internen Bus-Systeme entspricht. Auf einem 32-CPU NUMA-System reduzierte sich die Ausführungszeit beim SPECjbb2000 um bis zu 40% was den Aufwand sicher rechtfertigen würde.

Weiterführende Informationen:

- Sun, HotSpot Garbage Colleciton Tuning with the 5.0 Java Virtual Machine: [HOTSPOTGC]

- Mustafa M. Tikir, NUMA-Aware Java Heaps for Server Applications: [JAVANUMA]

8.4.4. Weitere Parameter

Die aktuelle Java HotSpot Virtual Machine unterstützt eine Reihe weiterer Parameter um die Geschwindigkeit zu optimieren. Eine Liste der offiziell dokumentierten Parameter ist unter [HOTSPOTOPT].

Sun hat einige Optionen unter den folgenden Parametern zusammengefasst:

```
-client
-server
```

Listing 75 Java, Client/Server VM Parameter

Die Optionen lassen eine Optimierung auf Server oder Client Anwendungen zu und beeinflussen einige der oben bereits genannten Parameter. Die Client-VM ist auf einen schnellen Programmstart und wenig Speicherverbrauch hin ausgelegt. Die Server-VM dagegen ist auf maximalen Durchsatz hin optimiert. Meistens macht es keinen Sinn diese Option manuell zu setzen da für Desktop-Betriebssysteme meist nur Client-Applikationen gestartet werden und auf Servern nur Server-Anwendungen laufen.

Hinweis: Unter Windows beinhaltet nur das JDK Package die Server-VM. Die JRE Variante beinhaltet lediglich die Client-Version. Zu erkennen sind die Versionen daran, dass im `<JRE_HOME>/bin/` jeweils ein Unterverzeichnis ‚`client`' bzw. ‚`server`' liegt. Darin ist dann die jeweils angepasste `jvm.dll` zu finden.

8.4.5. Reordering

Die Java Spezifikation erlaubt explizit die Umsortierung von Programmcode wenn dadurch das Ergebnis nicht beeinflusst wird. Dies kann zu möglicherweise unerwünschten Effekten führen wenn man sich dessen nicht bewusst ist. Beispielsweise könnte Java den folgenden Code auch umsortieren:

```
int a = x + 8;
int b = 10 * y;
int c = a + b ;
```

Listing 76 Java, Reordering 1

Der Code könnte also auch folgendermassen abgearbeitet werden:

```
int b = 10 * y;
int a = x + 8;
int c = a + b ;
```

Listing 77 Java, Reordering 2

Da das Ergebnis von c nicht von der Reihenfolge der einzelnen Instruktionen abhängt ist diese Umsortierung erlaubt.

Bei der Ausführung dürfen also auch keine Annahmen getroffen werden in welcher Reihenfolge der Code abgearbeitet wird. Beispielsweise wenn ein Thread auf Daten eines Objektes zugreift und davon ausgeht, dass ein anderer Thread entweder nichts gemacht hat oder die Änderungen in einer bestimmten Reihenfolge vornimmt. Wie oben zu sehen ist könnte die Reihenfolge der Modifikationen auch umsortiert werden.

Weiterführende Informationen:

- Brian Goetz, Java Concurrency in Practice, ISBN 0-321-34960-1: [4] S. 340.

8.4.6. Lock elosion, Lock coarsening

Zwei weitere Beispiele von Optimierung können anhand des folgenden Beispieles erklärt werden:

```
public String getStoogeNames() {
    List<String> stooges = new Vector<String>();
    stooges.add("Moe");
    stooges.add("Larry");
    stooges.add("Curly");
    return stooges.toString();
}
```

Listing 78 Java, Lock elosion, Lock coarsening

Dieser simple Code beinhaltet (zu) viel Synchronisation. Führt die JVM diesen Code genau so aus wie der hier aufgelistet ist, dann wird 4 Mal der Lock für das ‚`stooges`' Objekt angefragt und wieder freigegeben. Der Grund liegt in der Synchronisierung des Vektor Objektes. Sowohl die `add()` als auch die `toString()` Methoden sind synchronisiert. Eine Intelligente JVM könnte hier eine Optimierung vornehmen und den Lock nur einmal zuweisen, die vier Instruktionen ausführen und dann den Lock wieder abgeben. Dieses Verfahren wird ‚lock elosion' genannt. Die IBM JVM beherrscht dieses Verfahren und die Sun HotSpot JVM soll es in Version 7 ebenfalls können.

Eine weitere Möglichkeit zur Optimierung liegt hier darin das Locking komplett wegzulassen. Da es sich hier um eine Methodenvariable handelt und diese nie veröffentlicht wird kann gar niemand anders auf das `stooges` Objekt zugreifen. Somit wird auch nur ein Thread gleichzeitig auf diese Objektinstanz zugreifen und eine Synchronisation kann somit entfallen.

In diesem speziellen Fall wäre sogar noch eine weitere Optimierung möglich. Die Methode getStoogeNames() wird immer denselben Rückgabewert haben. Somit wäre es möglich die Methoden bei einem erneuten Aufruf gar nicht mehr abzuarbeiten sondern direkt denselben String zurückzugeben.

Weiterführende Informationen:

- Brian Goetz, Java Concurrency in Practice, ISBN 0-321-34960-1: [4] S. 233.

8.5. Zusammenfassung und Fazit

Dieses Kapitel hat einen Überblick über die Architektur von Java. Dies beinhaltet sowohl die Abstraktion der Hardware als auch des Betriebssystems. Daher können sich Java-Entwickler auch nur auf die von der Java API zur Verfügung gestellte Funktionalität beschränken. Direktre Zugriffe auf Betriebssystem-Funktionalität oder gar auf die Hardware ist nicht direkt möglich. Dies würde auch die Plattformunabhängigkeit unterlaufen.

Somit liegt es insbesondere an der JVM die Java-Anwendungen möglichst effizient auf der vorhandenen Hardware ablaufen zu lassen. Ob dabei beispielsweise Threads nur innerhalb der VM existieren oder nach aussen an das Betriebssystem weitergereicht werden (mittels nativer Unterstützung oder Bibliotheken wie POSIX-Threads) ist dabei nicht von der Applikation beeinflussbar. Vielmehr geht es hier darum die richtige JVM auszuwählen und diese richtig zu konfigurieren. Einerseits an die Anforderungen der Applikation und andererseits an die Möglichkeiten und Eigenschaften des Betriebssystems.

Für den Applikationsentwickler gilt es natürlich trotzdem möglichst effizient zu programmieren. Insbesondere bei der Synchronisierung ist dies wie gezeigt eine sehr komplexe Aufgabe. Es ist zu vermuten, dass bei Systemen mit mehr Prozessoren/Kernen/parallelen Threads der Synchronisierungsaufwand steigt (Stichwort ‚lock contention').

8.6. Auswirkungen auf die Aufgabenstellung

Gemäss der Aufgabenstellung analysieren wir die Skalierbarkeit auf Multi-Prozessor und Multi-Core Maschinen auf Java-Ebene. Die dazu verwendete Java-API wurde in diesem Kapitel vorgestellt. Diese scheint keine tiefer greifende Kontrolle der Threads auf Betriebssystem- oder Hardware-Ebene zu bieten. Beispielsweise bietet die API von `java.lang.Thread` keine Möglichkeit die Thread-Affinität zu setzen. Somit bleibt uns allenfalls der Umweg diese über externe Programme zu beeinflussen (soweit möglich). Möglicherweise ist dies zur effizienten Skalierung aber gar nicht notwendig und kann komplett der JVM überlassen werden. Der weitere Verlauf dieser Arbeit wird zeigen in wie fern Java-Applikationen auf der geforderten Hard- und Software skalierbar ist und gegebenenfalls optimiert werden kann.

Tabelle 34 Technologien mit direktem Einfluss auf die Arbeit

Technologie	Beschreibung
Java Threading	Es ist zu zeigen in wie fern Java-Threads auf der geforderten Hard- und Software-Kombination skalierbar ist. Beispielsweise ob eine ausgesuchte JVM überhaupt Threads auf Betriebssystem-Ebene erzeugt oder diese nur „emuliert" (Stichwort ‚green Threads').
JOMP	Das JOMP (siehe [PROCEXP]) Projekt bietet eine OpenMP Schnittstelle für Java um eine semi-automatische Parallelisierung zu erreichen. Diese Technologie soll auf ihr Potential hin untersucht werden.

Tabelle 35 Technologien mit indirektem Einfluss auf die Arbeit

Technologie	Beschreibung
JVM Optimierung	JVMs bieten üblicherweise einige Konfigurationsparameter (siehe Kapitel 8.4). Diese könnten die Performance natürlich auch beeinflussen. Es ist aber nicht das Ziel dieser Arbeit die Auswirkungen jedes Parameters auf die Skalierung einer spezifischen Applikation zu untersuchen. Solche Messungen gehören in den Bereich des Feintunings bei der Konfiguration einer Anwendung für den produktiven Einsatz.

9. Glossar

In diesem Kaptitel werden die wichtigsten Begriffe kurz zusammengefasst um einen schnellen Überblick über die Thematik zu ermöglichen.

Tabelle 36 Glossar

Begriff	Beschreibung
Affinität	Bezeichnet die Zuordnung eines Prozesses/Threads zu physikalischen Recheneinheiten. Durch die Definition einer Affinitätsmaske kann gesteuert werden auf welchen Recheneinheiten die Anwendung ausgeführt werden kann. Siehe Kapitel 6.8.
API	API (Application Programming Interface) defniert eine Schnittstelle zwischen verschiedenen Software Systemen. Eine API definiert typischerweise eine Reihe von Methoden, Parametern, Datentypen und Datenfeldern. Siehe z.B. POSIX Threads API, Kapitel 7.3.1.
AMD	Advanced Micro Devices; Hersteller von Mikroprozessoren. Siehe Kapitel 5.6.3.
ASMP	Asymmetric Multi Processing (ASMP) bezeichnet die Verarbeitung mit parallel arbeitenden Einheiten wobei einzelne Einheiten Spezialaufgaben zugewiesen sind. Somit sind nicht alle Einheiten gleichberechtigt. Siehe Kapitel 5.2.
Cache-Coherence	Bezeichnet die Synchronisierung des Caches bei Systemen mit mehreren Prozessoren und verteilten Caches. Siehe Kapitel 5.2.1.
CAS	Compare and Swap bzw. Compare and Set bezeichnet eine atomare (meist hardwareunterstützte) Operation in der ein gespeicherter Wert mit dem vermuteten Wert verglichen wird. Stimmt dieser überein, so wird ein neuer Wert gesetzt. Ansonsten wird nichts getan. CAS Funktionen erlauben Lock-freie Algorithmen. Siehe Kapitel 8.2.4.
CISC	Complex Instruction Set Computing: Bezeichnet Prozessoren mit einem grossen Befehlssatz. Dieser beinhaltet häufig auch komplexe Operationen, die somit mit einem Befehl abgearbeitet werden können. Vergleiche auch mit RISC. Siehe Kapitel 5.4.
CMP	Chip Multi Processing (CMP) bezeichnet einen Chip, der in der Lage ist mehrere Prozesse gleichzeitig abzuarbeiten. Dies passiert aber auf einem Chip und nicht auf mehreren Prozessoren. Siehe Kapitel 5.2.
CMT	Chip Multi Threading (CMT) ist eine Technologie bei der ein Prozessor bei jedem Taktzyklus n Instruktionen (je eine pro n-Threads) einlesen kann. Siehe Kapitel 5.3.
Collections	Ein insbesondere mit der Programmiersprache Java geläufiger Begriff für eine Sammlung von Daten(objekten) in einer Datenstruktur. Die einfachste Form einer Collection ist ein Array. Siehe Kapitel 8.1.2.

Begriff	Beschreibung
Context-Switch	Wechsel zwischen mehreren Prozessen oder Threads. Siehe Kapitel 7.2.1.
CPU	Abkürzung für Central Processing Unit. Wird synonym für die deutsche Bezeichnung Hauptrpozessor bzw. Prozessor verwendet.
Deadlock	Ein Zustand in dem Prozesse in einer zyklischen Abhängigkeit stehen und gegenseitig auf Ressourcen warten, die nur von einem anderen Prozess freigegeben werden können. Siehe Kapitel 7.1.
Garbage Collection (GC)	Bezeichnet den Prozess der Speicherverwaltung bzw. Speicher-Räumung durch die Entfernung ungenutzter Objekte. Dies ist nötig, da in Java beispielsweise der Speicher nicht in durch Destruktoren freigegeben werden kann. Siehe Kapitel 8.4.3.
Hyper-Threading	Eine von Intel bei einigen Pentium 4 Modellen eingeführte Technologie zur verbesserten Auslastung der internen Pipeline. HyperThreading stellt gegenüber dem Betriebssystem einen zweiten (virtuellen) Prozessor zur Verfügung. Dieser ist aber physikalisch gar nicht vorhanden. Instruktionen an diesen Prozessor können die Auslastung der internen Rechen-Einheiten des Pentium 4 verbessern.
IPC	Inter-Prozess-Kommunikation: Die Kommunikation zwischen zwei Prozessen in getrenntem Kontext. Siehe Kapitel 7.2.1.
Java	Eine von Sun Microsystems forcierte Programmtechnologie. Java-Programme werden nicht wie klassische C/C++ Programme in Plattformabhängige Binaries kompiliert sondern in den so genannten Bytecode. Dieser wird dann von der Java Virtual Machine interpretiert und zur Laufzeit optimiert. Java-Programme können somit auf jeder Plattform ausgeführt werden, für die eine Java Virtual Machine existiert. Siehe Kapitel 8.
JIT	Wird meistens in Verbindung mit JIT-Compilern (Just In Time) verwendet. Dabei ist die Eigenschaft gemeint, dass der Code (bei Java der Bytecode) zur Laufzeit der Programmes kompiliert und optimiert wird. Siehe Kapitel 8.4.1.
JOMP	Java-basierende Implementierung von OpenMP-Ähnlichen Direktiven zur Parallelisierung. Zu OpenMP siehe Kapitel 7.3.2.
JVM	Die Java Virtual Machine ist ein Interpreter für Java Bytecode. Die JVM ist dabei das Bindeglied zwischen Betriebssystem und den plattformunabhängigen Java Anwendungen. Siehe Kapitel 8.
Kernel	Zentrale Teil eines Betriebssystems, der die wesentlichsten Funktionen realisiert und sich zur Laufzeit permanent im Arbeitsspeicher befindet
KLT	Kernel Level Thread, Threads die auf Betriebssystemebene implementiert werden. Sind dem OS bekannt und können auf verschiedene CPUs verteilt werden
Kontext	Thread- oder Prozesskontext repräsentiert den Zustand eines Threads oder Prozesses und ist im Thread Control Block TCB oder Process Control Block PCB gespeichert

Begriff	Beschreibung
Livelock	Ein Zustand in dem zwei oder mehr Prozesse ihren Status dauernd verändern um weiterzukommen aber trotzdem immer blockiert werden. Siehe Kapitel 7.1.
Lock elosion, Lock coarsening	Bezeichnet zwei Techniken um unnötig häufiges Locking/Unlocking zu vermeiden. Dabei werden mehrere Locking-Anfragen hintereinander zusammengefasst. Wird ein Lock gar nicht benötigt und dieser automatisch wegrationalisiert, dann nennt man das Lock coarsening. Siehe Kapitel 8.4.6.
Lock Granularität	Definiert wie feinkörnig Locks auf Datenstrukturen vergeben sind. Dies kann sehr grob (ein Lock für alle Daten) oder sehr feinkörnig (bis mehrere unterschiedliche Locks pro Datenstruktur) sein. Siehe Kapitel 8.2.3.
Lock Splitting	Bezeichnet allgemein die Möglichkeit einen Lock für mehrere Objekte in mehrere Locks (für jedes Objekt einen) aufzuteilen. Siehe Kapitel 8.2.3.1.
Lock Striping	Bezeichnet die weitere Aufteilung eines Objektes durch mehrere Locks (z.B. Array-Sektionen). Siehe Kapitel 8.2.3.2.
MPI	Das Message Passing Interface (MPI) wird zum Nachrichtenaustausch (Inter-Process-Communication, IPC) verwendet. Dabei kann MPI transparent sowohl auf einem lokalen Rechner als auch verteilt im Netzwerk verwendet werden. Siehe Kapitel 7.3.4.
Mutex Lock	Mutual Exclusion (Mutex) ist ein Programmkonstrukt welches sicherstellt, dass nur ein einziger Prozess sich innerhalb eines geschützten Bereiches aufhalten kann. Siehe Kapitel 8.2 und 8.2.1.
NetBurst	Eine von Intel eingeführte Architektur-Bezeichnung die im Wesentlichen eine lange Pipeline und dadurch eine hohe Taktrate beinhaltet. Die Architektur wurde für Pentium 4 Prozessoren entwickelt und verwendet, wird aber nicht mehr weiter verfolgt. Siehe Kapitel 5.5 und 5.6.1.
NUMA	Non-Uniform Memory Access (NUMA) bezeichnet eine Architektur in der jede Verarbeitungseinheit lokalen Speicher besitzt und durch Kommunikation mit den anderen Verarbeitungseinheiten auch deren Speicher ansprechen kann. Siehe Kapitel 5.2.1.
OpenMP	Eine Spezifikation der API zur Parallelisierung von Programmen. OpenMP definiert Compiler-Direktiven damit ein Compiler den bestehenden Code parallelisieren kann. Siehe Kapitel 7.3.2
Package	Ein bei der Programmiersprache Java geläufiger Begriff für die hierarchische Sortierung von Klassen. Ähnlich den Namensräumen (engl. Namespace) bei C++. Bei der Bezeichnung java.util.Vector handelt es sich um den voll qualifizierten Bezeichner für die Klasse Vector im Package java.util. Siehe Kapitel 8.
Pipelining	Bezeichnet die Abarbeitung einer Instruktion in vereinfachten Teilschritten. Dadurch kann die folgende Instruktion bereits eingelesen werden sobald die vorhergehende die

Begriff	Beschreibung
	nächste Stufe erreicht hat. Siehe Kapitel 5.5.
POSIX Threads	POSIX definiert eine Schnittstelle zwischen Applikation und Betriebssystem. Die Schnittstelle ist plattformunabhängig definiert und erlaubt somit die portable Programmierung. POSIX Threads bezeichnet die Behandlung von Threads mit POSIX-Schnittstellen. Siehe Kapitel 7.3.1.
Reordering	Bezeichnet eine Technik der Code-Optimierung. Hierbei darf der Compiler/Interpreter Anweisungen umsortieren um ein optimiertes Laufzeitverhalten zu erziehen. Dabei muss aber garantiert bleiben, dass das Endergebnis nicht verfälscht wird. Siehe Kapitel 8.4.5.
RISC	Reduced Instruction Set Computing: Bezeichnet Prozessoren mit einem kleinen Befehlssatz. Komplexe Befehle werden im Gegensatz zu CISC Prozessoren in mehreren Schritten ausgeführt. Befehle wie „Wert an Speicherstelle XY inkrementieren“ werden zu „Wert laden, wert Inkrementieren, Wert zurückschreiben“. Siehe Kapitel 5.4.
Scheduling	Bezeichnet die Tätigkeit des Betriebssystems beim Preemptiven Multitasking die Prozessorzeit nach einem bestimmten Algorithmus den einzelnen Ausführungseinheiten zuzuweisen (auf Ebene Thread oder Prozess). Siehe Kapitel 6.7.
Skalar	Ein Prozessor in Skalarem Design verarbeitet immer nur eine Instruktion gleichzeitig. Siehe Kapitel 5.4.
SMP	Symmetric Multi Processing (SMP) bezeichnet die Verarbeitung mit parallel arbeitenden Einheiten wobei jede Einheit gleichberechtigt behandelt wird. Siehe Kapitel 5.2.
Starvation	Starvation ist ein Zustand in dem ein Prozess auf Ressourcen oder Daten wartet und diese nie bekommt. Der Prozess kann somit nie eine Arbeit anfangen oder erledigen. Siehe Kapitel 7.1.
Superskalar	Ein Prozssor in superskalarem Design versucht mittels Dispatcher alle Recheneinheiten gleichzeitig auszulasten. Siehe Kapitel 5.4.
Super-Threading	Super-Threading ist eine Technologie bei der ein Prozessor bei jedem Taktzyklus eine Instruktion eines Threads einlesen kann. Siehe Kapitel 5.3.
Synchronisierung	Allgemeine Bezeichnung für die Überwachung von konkurrierenden Zugriffen. Siehe Kapitel 8.2.
TBB	Intel Thread Building Blocks. Eine C++ Bibliothek die Methoden zur parallelen Verarbeitung bereitstellt (Schleifenparalleisierung). Siehe Kapitel 7.3.3.
TDP	Termal Design Power. Bezeichnet die typische Leistungsabgabe von elektronischen Bauteilen. Bei der TDP handelt es sich um einen wichtigen Wert zur Dimensionierung von Kühllösungen.

Begriff	Beschreibung
	Siehe auch Kapitel 5.5 und 5.6.1.
Thread	Ein leichtgewichtiger Prozess. Ein Thread teilt den Adressraum mit dem Prozess zu dem er gehört. Dadurch werden einerseits die Kommunikation und andererseits der Kontextwechsel beschleunigt. Siehe Kapitel 7.2.2 und 8.1.1.
Thread-Safety	Thread-Safety ist ein Attribut, welches bei der parallelen Programmierung verwendet wird um zu spezifizieren, dass der parallele Zugriff auf ein Objekt selbst dann sicher ist, wenn mehrere Zugriffe gleichzeitig stattfinden. Sicherheit bedeutet in diesem Zusammenhang, dass keine unerwarteten Ereigenisse oder Zustände eintreten können. Siehe Kapitel 8.
UMA	Uniform Memory Access (UMA) bezeichnet eine Architektur in der alle Verarbeitungseinheiten über ein gemeinsames Bussystem auf den Speicher zugreifen. Siehe Kapitel 5.2.1.

10. Verzeichnisse

10.1. Tabellenverzeichnis

10.2. Abbildungsverzeichnis

10.3. Code Listings

10.4. Index