UNIVERSITÄT KASSEL
Fachgebiet Elektrotechnik/Informatik
Fachbereich Programmiersprachen-/methodik

An Evaluation of Threading Systems

Diplomarbeit

Betreuer: Prof. Dr. Claudia Leopold
Dipl.-Inf. Michael Süß

Kassel, 31. März 2006

von
Tobias Gunkel
Geburtsdatum: 9. Dezember 1981
Studiengang: Informatik

Selbständigkeitserklärung

§13(7) der Prüfungsordnung vom 30.04.2002 für den Diplomstudiengang Informatik im Fachbereich Elektrotechnik der Universität Kassel:

"Bei der Abgabe der Bachelorarbeit hat der Kandidat/die Kandidatin schriftlich zu versichern, daß er/sie seine/ihre Arbeit - bei einer Gruppenarbeit seinen/ihren entsprechend gekennzeichneten Anteil der Arbeit - selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt hat."

Hiermit versichere ich, dass ich die vorliegende Diplomarbeit selbstständig erarbeitet und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie Zitate kenntlich gemacht habe.

Tobias Gunkel

Wolfhagen, 31.03.2006

Inhaltsverzeichnis

Abkürzungsverzeichnis

SMP	Symmetric Multi-Processing
SISD	Single Instruction, Single Data
SIMD	Single Instruction, Multiple Data
MISD	Multiple Instruction, Single Data
MIMD	Multiple Instruction, Multiple Data
IEEE	Institute of Electrical and Electronics Engineers
POSIX	Portable Operating System Interface for Unix
API	Application Programming Interface
NPTL	Native POSIX-Threads Library
NGPT	Next Generation POSIX Threading
J2SE	Java 2 Standard Edition
JVM	Java Virtual Machine
JRE	Java Runtime Environment
MSIL	Microsoft Intermediate Language
CLR	Common Language Runtime
FLS	Fiber Local Storage
TLS	Thread Local Storage
TSD	Thread-Specific Data
FIFO	First In First Out
LWP	Light Weight Process

Kapitel 1

Einleitung

Warum die Zukunft in der Parallelisierung liegt
und
Multi-Threading so wichtig ist, wie nie zuvor

1.1 Motivation

Am 19. April 1965 veröffentlichte die Zeitschrift “Electronics“ einen Artikel von Gordon E. Moore. In diesem beschäftigte sich der Autor mit der Zukunft der damals neuen Technologie integrierter Schaltungen. Moore war der erste, der beobachte, dass es eine Regelmäßigkeit in der Entwicklung der Anzahl von Komponenten auf integrierten Schaltungen gibt. Er nahm an, dass sich diese jährlich verdoppeln würde, musste dann aber einige Jahre später seine Aussage insofern korrigieren, als dass es lediglich eine Verdopplung alle zwei Jahre gäbe [Moo65]. Diese zum Erscheinungsdatum kaum beachtete Aussage ist heute, nach fast 40-jährigem Bestehen, eine der fundamentalsten in der Mikroelektronik. Auch wenn mittlerweile eine Verdoppelung alle 18 Monate angenommen wird, hat das Mooresche Gesetz die vergangene Entwicklung integrierter Schaltungen mit sehr hoher Genauigkeit vorhergesagt und gilt noch heute. Nach Aussage führender Köpfe aus der Mikroelektronikindustrie wird dieses Gesetz auch noch weitere 10 Jahre gelten.

Momentan sehen sich die Entwickler jedoch mit größeren Problemen konfrontiert. Die bisherige Vorgehensweise, die größeren Integrationsdichten für höhere Taktraten auszunutzen, lässt sich aufgrund zu hoher Abwärme der Transistoren mittlerweile nur noch mit erheblichem Aufwand durchführen. Damit die Prognosen des Mooreschen Gesetzes weiterhin eingehalten werden können, setzen die Firmen daher vermehrt auf verbesserte Architekturen. Die gegenwärtig wohl wichtigste Verbesserung dieser Art dürfte das Umschwenken von Single-Core auf Multi-Core CPUs sein. Hierunter versteht man die Integration mehrerer CPUs in einem gemeinsamen Gehäuse. So kündigte Intel [1] im Herbst 2004 an, dass bis Ende 2006 etwa 70%

[1] dessen Mitbegründer G. Moore ist

der neu produzierten Heim-PCs Dual-Core-Systeme sein werden. Durch den Einsatz der Dual-Core Prozessoren soll sich innerhalb von 4 Jahren die Prozessorleistung verzehnfachen. Mit Single-Core CPUs wäre im gleichen Zeitraum lediglich eine Verdreifachung erreichbar (vgl. [int])

Die Lebenszeit von PC-Hardware ist sehr gering. Sollte sich zeigen, dass die Prognose von Intel stimmt, so stünden bereits in wenigen Jahren den meisten PC-Anwendern Multiprozessorsysteme zur Verfügung. Es sollte jedoch erwähnt werden, dass Intel nicht die einzige Firma ist, die das Potential von Multiprozessorsystemen im Heimbereich entdeckt hat. Mittlerweile bieten fast alle Prozessorhersteller Dual- bzw. Multicore-CPUs für "Normalanwender" an. Man darf auch nicht vergessen, dass der Einsatz von Multiprozessoren in professionellen Systemen, vor allem Serversystemen, schon lange Zeit Standard ist. Mit der Erschließung des Heimbereiches, der sich lange Zeit echter Parallelisierung verschlossen hat, könnte nun aber die Parallelverarbeitung auf allen Gebieten ihren Durchbruch feiern.

Diese neuen Umstände sollten ein Umdenken bei den Software-Entwicklern mit sich bringen. In der Vergangenheit wurden kaum große Anstrengungen unternommen, um Programme für den Heimbereich durch Parallelisierung zu optimieren. Aufgrund mangelnder Hardware konnte bei einer Großzahl von Programmen meist nur ein geringer Geschwindigkeitsvorteil erzielt werden, wenn überhaupt. Ganz anders als in den Bereichen der Serveranwendungen oder wissenschaftlichen Programme, in denen schon seit langer Zeit Multiprozessorsysteme erfolgreich eingesetzt werden.

In Zukunft wird sich somit immer öfter die Frage stellen, wie man Programme effizient parallelisieren kann, um die nun vorhandenen Multiprozessoren auch auslasten zu können. Je mehr Programme parallelisiert werden müssen und je unterschiedlicher diese Programme sind, desto mehr Herausforderungen werden bei der Parallelisierung gemeistert werden müssen. Ist bei einigen Anwendungen die Parallelität inhärent, so ist sie bei anderen wiederum nur sehr schwer oder überhaupt nicht herbeizuführen. Probleme dürften auch Programme mit einem hohen Grad an Benutzerinteraktion schaffen, da Berechnungen auf Eingaben des Benutzers warten müssen. Wie effizient die Parallelisierung letztendlich funktioniert, hängt im großen Maße von den eingesetzten Mitteln zur Parallelisierung ab.

In Zukunft sollten auch Programmierer mit geringen Kenntnissen der Parallelverarbeitung ein Programm parallelisieren können. Daher müssen Programmiersysteme oder Bibliotheken, die für die Parallelisierung verwendet werden, auch den Aspekt der "Usability", also der Anwenderfreundlichkeit berücksichtigen.

Eine Gruppe von parallelen Programmiersystemen sind die sogenannten Threading-Systeme. Sie sind für Multiprozessoren geeignet und für fast alle dieser Architekturen verfügbar. Die momentane und die angekündigte zukünftige Konzentration der Prozessorhersteller auf Multiprozessorsysteme unterstreicht die Relevanz dieser Programmiersysteme. Mitte der 90er Jahre hat es einen Boom dieser Threading-Systeme gegeben. Die Zahl der Threading-Systeme hat mittlerweile unüberschaubare Ausmaße angenommen. Aus diesem Grund ist eine nähere Be-

trachtung dieser Systeme notwendig, um die Möglichkeiten, aber auch die Einschränkungen der großen Anzahl momentan zur Verfügung stehenden Threading-Systeme überblicken zu können.

1.2 Zielsetzung

Ziel dieser Arbeit ist es, einen Überblick über die Eigenschaften existierender Threading-Systeme zu geben. Dabei sollen die Systeme kategorisiert und miteinander verglichen werden. Unterschiede u.a. in den Bereichen Strukturierung, Performance, Usability, Funktionalität, Synchronisation und Kommunikation werden aufgezeigt und bewertet.

1.3 Aufbau

Das folgende Kapitel stellt die Grundlagen der parallelen Programmierung vor, um dem Leser einen Einblick in dieses Gebiet zu verschaffen und das Verständnis des Thread-Begriffes zu erleichtern. Im dritten Kapitel werden Programmiersprachen und Threading-Systeme vorgestellt, deren Eigenschaften dargestellt und verglichen werden. Im Hauptteil dieser Arbeit werden Konzepte von Threading-Systemen untersucht und die Implementierungen dieser Konzepte in den ausgewählten Threading-Systemen betrachtet, verglichen und evaluiert. Im Anschluss soll eine Schlussfolgerung aus den Betrachtungen gezogen werden.

Kapitel 2

Grundlagen

Damit der Begriff des “Thread“ verstanden werden kann, sind Vorkenntnisse bezüglich der Bedeutung von Nebenläufigkeit, Parallelität und Tasks nötig. Diese soll der nun folgende Abschnitt liefern.

2.1 Nebenläufigkeit und Tasks

Jedes Programm, das auf einem Rechner ausgeführt werden kann, besteht aus einer Folge von Befehlen in Maschinensprache. Diese können der Reihe nach ausgeführt oder durch Sprünge zu einer anderen Stelle im Code in beliebiger Reihenfolge ausgeführt werden. Register verwalten dabei den Status der aktuellen Berechnung. So zeigt der Inhalt eines speziellen Registers, der Programmzähler (engl.: Program-Counter) oder kurz PC, auf den Befehl im Maschinencode des Programmes, der als nächstes ausgeführt werden soll. Weitere Register beinhalten z.B. das Ergebnis der letzten Berechnung, die Adresse des obersten Elements auf dem Stack oder Offset-Adressen für bestimmte Datensegmente im Speicher.

Ein Rechner mit nur einer CPU, der wie fast alle der heutigen Computer auf der “von-Neumann“-Architektur basiert, lädt nun in jedem Rechenzyklus den Befehl, auf dessen Adresse der PC zeigt und führt ihn aus. Dabei wird der PC auf die Adresse des nächsten Befehls gesetzt und die anderen Register gegebenenfalls mit neuen Werten beschrieben.

Führt man ein Programm auf einem Multiprozessorsystem zur gleichen Zeit mehrmals aus, so werden die Prozesse je nach Anzahl der Prozessoren parallel abgearbeitet. Steht nur eine Recheneinheit zur Verfügung, kann nur ein Befehl in jedem Rechenzyklus ausgeführt werden. Es muss in gewissen Abständen zwischen der Ausführung des ersten und zweiten Prozesses hin- und hergeschaltet werden. Da man hier nicht von Parallelität sprechen kann, nennt man dies Pseudo-Parallelität. Besitzt der Rechner hingegen mehrere CPUs, so können die Prozesse parallel abgearbeitet werden, wobei jede CPU einen Befehl einer der beiden Prozesse ausführt. Aber auch hier muss zwischen den Prozessen umgeschaltet werden, wenn mehr Prozesse als CPUs existieren. Das Umschalten nennt man auch ***Kontext-Wechsel*** (engl.: context-switch).

Wann ein Kontext-Wechsel erfolgen soll, wird von einem sogenannten Scheduler, der Teil des Betriebssystems ist, entschieden. Nach welchen Richtlinien der Scheduler dabei vorgeht, ist durch seinen ***Scheduling-Algorithmus*** spezifiziert. Bei jedem Kontext-Wechsel müssen die Registerinhalte des aktuellen Prozesses wiederhergestellt werden, da sich die Prozesse zu einem Zeitpunkt an verschiedenen Stellen im Maschinencode aufhalten und damit unterschiedliche Berechnungen ausführen. Ansonsten würden die Prozesse ihre Registerinhalte gegenseitig überschreiben.

Wie man im Laufe dieses Kapitels sehen wird, gibt es nicht nur Prozesse, die auf einer CPU abgearbeitet werden können. Eine auf der CPU ausführbare Einheit aus zusammenhängendem Maschinencode mit ihren Registern soll in den folgenden Abschnitten als ***Task*** bezeichnet werden. Tasks werden auch häufig Handlungsfäden, im englischen Thread, genannt. Diese an sich treffende Bezeichnung führt aber zu Verwirrungen, da auch die in dieser Arbeit betrachteten Threads, die spezielle Formen von Tasks sind, den gleichen Namen tragen. Tasks, die zur gleichen Zeit, also parallel, auf einer oder mehreren CPU ausgeführt werden, betrachtet man als ***nebenläufig***.

Zur Verdeutlichung ein kleines Beispiel. In einem besseren Textverarbeitungsprogramm werden meist mehrere Tasks zusammen abgearbeitet. So kann der Benutzer Text eintippen, während im Hintergrund der bereits getippte Text formatiert wird. Da diese beiden Tasks, bzw. Handlungsabläufe parallel ablaufen, wird der Programmierer sie als zwei verschiedene Prozesse oder Threads implementieren. Diese sind dann nebenläufig.

Anhand dieser ersten Betrachtung scheinen sich Prozesse und Threads zu gleichen. Es gibt jedoch wichtige Unterschiede zwischen diesen Arten von Tasks.

2.2 Vergleich von Prozessen und Threads

Nun stellt sich vor allem die Frage, worin die Unterschiede zwischen Prozessen und Threads bestehen. Was sind die Vor- und Nachteile und welche dieser Task-Arten sollte man für welche Aufgabenstellung verwenden?

2.2.1 Kernel-Mode und User-Mode

Da Prozesse und in den meisten Fällen auch Threads eng an das darunterliegende Betriebssystem gekoppelt sind, tauchen in diesem Zusammenhang oftmals die Begriffe Kernel-Mode und User-Mode auf. Das grobe Verständnis dieser Begriffen ist wichtig, um den Unterschied zwischen Prozessen und Threads zu verstehen.

In frühen Betriebssystemen war es möglich, Programme zu schreiben, mit denen auf die Datenstrukturen bzw. den Adressraum des Betriebssystems oder anderer Programme zugegriffen werden konnte. Dadurch konnten Programmierfehler oder bösartige Programme ohne weiteres

das gesamte Betriebssystem zum Absturz bringen. Mittlerweile ist dies nicht mehr möglich, da die Befehle eines Programmes in einem von zwei mehr oder minder privilegierten Modi ausgeführt werden. Einer der beiden Modi ist der User-Mode. In ihm darf nur auf Datenstrukturen des Programmes selbst zugegriffen werden. Man kann ihn als eine Art Sandkasten betrachten, in dem ein Programm anstellen kann, was es will. Ein Zugriff auf Daten, die vom Betriebssystem verwaltet werden, ist hier nicht gestattet. Mittels Systemaufrufen ist es jedoch möglich, dem Betriebssystem einen Auftrag zu geben, eine Datenstruktur zu ändern oder sonstige Dinge zu tun, die im User-Mode nicht möglich sind. Dabei wird die Kontrolle dem Programm entzogen und an das Betriebssystem übergeben. Das Betriebssystem wechselt dann in den sogennanten Kernel-Mode, in dem es vollen Zugriff auf alle Datenstrukturen hat. Dies schließt natürlich die Daten des aufrufenden Programmes ein. Die Ergebnisse des Aufrufs können dann in eine Datenstruktur des Programmes geschrieben oder als Rückgabewert übergeben werden. Nach erfolgreicher oder fehlgeschlagener Ausführung des Systemaufrufs wird die Kontrolle wieder an das Programm abgegeben, das daraufhin im User-Mode weiterarbeiten kann.

Wichtig ist hierbei vor allem, dass der Wechsel vom User- in den Kernel-Mode und wieder zurück in den User-Mode Zeit braucht, da auch hier, wie bei einem Prozesswechsel, ein Kontext-Wechsel vollzogen werden muss. Es müssen Register gesichert und weitere Aktionen durchgeführt werden, bevor das Betriebssystem die Kontrolle übernehmen kann. Aus diesem Grund sollte man Systemaufrufe wenn möglich meiden und stattdessen Bibliotheksfunktionen verwenden, die im User-Mode ausgeführt werden können.

2.2.2 Prozesse

Ein Prozess ist ein Programm in Ausführung. Jeder Prozess besitzt eigene Dateideskriptoren, Signale, Stack, Adressraum und Register und somit auch einen eigenen Programmzähler PC. Ein Prozess kann nach seiner Erzeugung einen Handlungsstrang verfolgen, d.h. einen Task realisieren. Die einfachste Möglichkeit Nebenläufigkeit zu erhalten ist es, mehrere Prozesse eines oder verschiedener Programme gleichzeitig laufen zu lassen. Lässt man diese Prozesse auf einem Mehrprozessorsystemen ablaufen, werden diese je nach Anzahl der vorhandenen Prozessoren parallel abgearbeitet. Selbst auf Systemen mit nur einer CPU kann dies eine Beschleunigung bringen. Nämlich dann, wenn die Programme sehr viele langwierige I/O-Operationen ausführen. Würde lediglich ein Prozess gestartet und dieser auf Beendigung einer I/O-Anfrage warten, würde der Prozessor brach liegen. Bei mehreren Prozessen kann ein Kontextwechsel auf einen lauffähigen Prozess erfolgen. Dieser kann dann die CPU verwenden, während der blockierte Prozess wartet.

Ein Beispiel, wie ein Multiprozessorsystem von Prozessen profitieren könnte, wäre z.B. der gleichzeitige Betrieb eines Office-Programmes, eines Virenscanners und einer Firewall. So könnte das Office-Programme auf der ersten CPU ausgeführt werden, während die Instruktionen des Virenscanners von CPU2 und die Firewall von CPU3 ausgeführt werden. Lästige “Hänger“ des Office-Programmes, wie sie auf einem Einprozessorsystem durch die beiden Hintergrund-

dienste hervorgerufen werden könnten, sollten hier nicht mehr auftreten.

Prozesse werden auch “schwergewichtige Prozesse“ genannt. Das Wort “schwergewichtig“ soll verdeutlichen, dass die Erzeugung von Prozessen und der Kontextwechsel zwischen Prozessen mit hohem Verwaltungsaufwand verbunden sind. Die Datenstrukturen von Prozessen werden komplett vom Betriebssystem im Kernel-Mode verwaltet und können lediglich über Systembefehle erzeugt und verändert werden. Damit ist für jede Prozessumschaltung ein langwieriger Wechsel in den Kernel-Mode notwendig.

Die Kommunikation zwischen zwei Prozessen ist auch nicht ohne weiteres möglich, da Prozesse keinen gemeinsamen Adressraum besitzen. Daher können sie hierüber auch keine Daten austauschen. Für die Kommunikation und Synchronisation muss auf die sogenannte Interprozess-Kommunikation (IPC) zurückgegriffen werden, mit denen unter anderen auch ein gemeinsamer Adressraum realisiert werden kann. Die Verwendung von IPC ist jedoch meist sehr aufwändig und ineffizient.

2.2.3 Threads

Der große Nachteil von Prozessen bei der parallelen Programmierung ist, dass sie

1. keinen gemeinsamen Adressraum besitzen
2. vom Betriebssystem verwaltet werden und langsame Kontextwechsel nötig sind
3. die Kommunikation der Threads untereinander sehr schwierig ist

Der Einsatz von Threads löst viele der oben genannten Probleme. Sie sind ebenfalls nebenläufige “Handlungsstränge“, d.h. Tasks, jedoch innerhalb eines Prozesses. Jeder Prozess besitzt bei seiner Erzeugung bereits automatisch einen Thread, den Haupt-Thread, der die Ausführung des Programmes übernimmt. Im Gegensatz zu Prozessen besitzen Threads, die innerhalb des gleichen Prozesses erzeugt wurden, einen gemeinsamen Adressraum, nämlich den des Prozesses. Jeder Thread besitzt einen eigenen Stack und eigene Register und somit auch einen eigenen Programmzähler. Durch den gemeinsamen Adressraum ist eine Kommunikation der Threads untereinander sehr leicht zu implementieren. Ein weiterer Vorteil von Threads gegenüber Prozessen ist bei einigen (aber nicht allen) Threading-Systemen, dass sich nicht das Betriebssystem, sondern der Prozess selbst um die Verwaltung der Threads kümmert. Das Betriebssystem hat also keine Kenntnis davon, dass die Threads überhaupt existieren. Threads können dann im User-Mode erzeugt werden, wodurch die zeitintensiven Sprünge in den Kernel-Mode entfallen. Dieser Vorteil ist aber auch gleichzeitig ein Nachteil, da das für das Scheduling zuständige Betriebssystem die Threads als einen Prozess ansieht und alle Threads blockiert, wenn auch nur einer davon eine I/O-Operation startet und daraufhin blockiert.

2.2.4 Kernel-Level vs. User-Level Threads

Es gibt zwei unterschiedliche Arten von Threads, die unterschieden werden müssen: Kernel-Level und User-Level Threads. Anhand der Namen kann man bereits erkennen, dass die Unterscheidung dieser Thread-Arten mit Kernel- und User-Mode zusammenhängen. Tatsächlich ist es so, dass diese Bezeichnungen Auskunft darüber geben, in welchem Bereich die Threads implementiert sind, bzw. wo die Datenstrukturen zur Verwaltung der Threads abgelegt sind. Mag sich diese Unterscheidung im Moment trivial anhören, so sind die Auswirkungen dieser Unterschiede jedoch gravierend.

Als Kernel-Level Threads bezeichnet man zum einen Threads, die vom Kernel selbst unterstützt werden und zum anderen Bibliotheken, die Gebrauch von diesen Threads machen. Die Bibliotheken sollen hier als Kernel-Level Thread-Bibliotheken und die eigentlichen Threads auf Betriebssystemebene als Kernel-Level Threads bezeichnet werden. Kernel-Level Threads werden über Systemaufrufe vom Betriebssystem erzeugt, verwaltet und geschedult. Die Datenstrukturen sind nur im Kernel-Mode einsehbar und änderbar. Bis Anfang der 1990er Jahre besaß kaum ein Betriebssystem Threading-Funktionalität (vgl. [LB98, S. 2]. Dies hat sich im Laufe der 1990er Jahre schlagartig geändert, so dass bereits 1997 die bekanntesten Betriebssysteme Threads im Betriebssystemkern implementierten.

User-Level Threads hingegen werden durch Programmbibliotheken realisiert. Dabei gibt es jedoch verschiedene Ansätze.

Will man Multithreading auf einer Plattform verwenden, deren Betriebssystem keine Threads unterstützt, hat man keine andere Wahl, als diese selbst zu implementieren oder eine Programmbibliotheken zu verwenden, die dies bereits tut. Eine solche Bibliothek müsste alle Aspekte des Multi-Threadings, von der Erzeugung bis zur Synchronisierung der Threads, komplett selbst implementieren. Das heißt auch, dass die Bibliothek einen Scheduler zur Verfügung stellen muss, der entscheidet, welcher der User-Threads als nächstes und für wie lange laufen darf. Einer der Vorteile einer solchen Bibliothek ist, dass sie systemunabhängig und somit leicht portierbar ist. Da kein Systemaufruf zur Verwaltung der Threads ausgeführt werden müssen und somit kein Wechsel in den Kernel-Mode nötig ist, sind Kontextwechsel zwischen den Threads und Änderungen der Thread-Eigenschaften schneller zu vollziehen. Diese Lösung hat aber auch sehr große Nachteile. Zum einen hat der Betriebssystemkern keine Kenntnis von der Existenz der User-Level-Threads. Er und vor allem sein Scheduler sieht sie als einen Prozess bzw. einen Kernel-Level-Thread. Somit wird einem Prozess mit User-Level-Threads vom Scheduler nur das Zeitquantum zugebilligt, das auch Prozesse mit nur einem Thread bekommen. Noch unangenehmer ist es, dass ein durch eine E/A-Operation blockierter Thread nicht nur sich, sondern den gesamten Prozess blockiert. Der Prozess kann also nicht mehr weiterrechnen. Erst wenn die Daten der E/A-Operation zur Verfügung stehen, kann der Prozess sich in die Warteschlange des Schedulers einreihen, um darauf zu warten vom Scheduler ausgewählt zu werden. Ohne Kenntnis der einzelnen Threads von Seiten des Betriebssystems ist es auch nicht möglich, mehrere CPUs gleichzeitig zu verwenden. Es findet also keine Parallelität zwischen den Threads statt,

sondern lediglich Pseudo-Parallelität.

Ein anderer Ansatz von User-Level Thread-Bibliotheken ist es, einfach nur als eine Art Wrapper für die bereits existierenden Kernel-Level Threads zu fungieren. Dies ist sinnvoll, da die Systemaufrufe zur Erzeugung von Kernel-Level Threads oft sehr kompliziert sind. Zum einen sollen die Systemaufrufe durch eine Vielzahl von Parametern möglichst allen Bedürfnissen bei der Erzeugung und Verwaltung von Threads gerecht werden. Zum anderen ist oftmals Wissen über die Interna des Betriebssystems gefragt, das ein normaler Programmierer oft nicht besitzt. Zudem sind Systemaufrufe nicht portabel, weshalb ein direkter Zugriff hierauf vermieden werden sollte. Eine User-Level-Bibliothek kann nun Funktionen zur Verfügung stellen, die sehr viel einfacher verwendet werden können. Die Standardisierung dieser Funktionen, so dass sie für Kernel-Level Threads verschiedener Systeme implementiert werden können, ist ein weiteres wichtiges Ziel von User-Level Threading-Bibliotheken.

Für die Zuweisung von User-Level Threads auf die Kernel-Level Threads gibt es verschiedene Möglichkeiten, die ***Threading-Modelle*** genannt werden:

1:1 Threading Modell Hierbei wird für jeden User-Level Thread eines Prozesses ein Kernel-Level Thread erzeugt. Jeder User-Level Thread ist fest mit einem der Kernel-Level Threads verbunden. Dies bietet den Vorteil, dass das Betriebssystem die Threads verwalten und schedulen kann. Jeder Thread erhält das volle Zeitquantum vom Scheduler und braucht sich dieses nicht mit anderen Threads zu teilen. Sollte ein Thread z.B. durch eine E/A-Operation blockieren, können die anderen Threads weiterhin ausgeführt werden und blockieren nicht. Der Nachteil ist der gleiche, der bereits für Kernel-Level Threads galt. Durch Kontext-Wechsel in den Kernel-Mode ist der Wechsel zwischen den Threads und deren Verwaltung langsam.

1:N Threading Modell In diesem Modell wird nur ein Kernel-Level Thread für den gesamten Prozess angefordert. Alle User-Level Threads müssen sich diesen Thread teilen und vom Scheduler der User-Level Bibliothek auf diesen Thread geschedult werden. Der Vorteil ist, dass nach der Erzeugung des einen Kernel-Level Threads keine weiteren Systemaufrufe für die Verwaltung der Threads notwendig sind. Hierum kümmert sich die Bibliothek selbst. Dadurch, dass lediglich im User-Mode gearbeitet wird und Kontext-Wechsel entfallen, arbeitet diese Lösung sehr schnell. Der Nachteil ist, dass der Scheduler nur den Kernel-Level Thread sieht und sich somit alle User-Level Threads das Zeitquantum des Kernel-Level Threads teilen müssen. Auch der Effekt, dass ein Thread im Blockierungsfall auch alle anderen Threads blockiert, kommt hier zum tragen. Der bedeutendste Nachteil aber ist, dass nur eine CPU gleichzeitig verwendet werden kann, wodurch die Threads nur pseudo-parallel ausgeführt werden können.

M:N Threading Modell Dieses Modell ist eine Mischform aus 1:1 und 1:N Modell. Es werden mehrere Kernel-Level Threads angefordert, die dann mittels eines eigenen Schedulers auf die vorhandenen User-Level Threads verteilt werden müssen. Normalerweise ist die Anzahl

der Kernel-Level Threads geringer, als die der User-Level Threads. Die überschüssigen Kernel-Level Threads würden sonst brachliegen. Um ein größeres Zeitquantum zugewiesen zu bekommen, wäre es aber auch möglich dies durch eine Verwendung überzähliger Kernel-Level Threads zu erreichen. Ein Blockieren aller Threads kann ausgeschlossen werden, wenn immer ein Kernel-Level Thread mehr erzeugt wird als die Anzahl von User-Level Threads, die zur gleichen Zeit blockieren können. Die Laufzeit dieses Modells liegt zwischen den 1:1 und 1:N Varianten, da hier in den Kernel-Mode gewechselt werden muss. Die Laufzeit wird aber auch dadurch beeinträchtigt, dass alles doppelt verwaltet werden muss. So sind unter anderem zwei Scheduler notwendig: für die Kernel-Level Threads und für die User-Level Threads. Durch den dadurch erzeugten Overhead wird die Ausführung gebremst.

Welches der Modelle nun das beste ist, kann nicht eindeutig gesagt werden. Die Frage, ob eine Bibliothek das 1:1 oder M:N Threading-Modell verwenden soll, entfacht regelmäßig Glaubenskriege unter den Anhänger der jeweiligen Modelle. Selbst die Firma RedHat konnte sich bei der Entwicklung der neuen NPTL Pthread-Bibliothek nicht auf eine dieser Varianten einigen. Zuerst hieß es, dass nur ein (recht kompliziertes) M:N Modell in Frage käme. Diese Ansicht wurde jedoch kurz darauf wieder verworfen, um sich doch für das 1:1 Modell zu entscheiden. (vgl. [Dre05]). Was man jedoch sagen kann, ist, dass die 1:N Variante in den meisten Fällen nicht empfehlenswert ist, da mehrere CPUs nicht gleichzeitig verwendet werden können. Sie ist jedoch notwendig, wenn nicht auf Kernel-Level Threads zugegriffen werden kann.

2.3 Parallele Programmiersysteme

Neben der Möglichkeit direkt mit Prozessen und Threads zu arbeiten, gibt es auch die Möglichkeit auf Programmiersysteme zurückzugreifen, die auf den Mitteln des Betriebsystems oder anderen Bibliotheken aufbauen. Sie bieten dem Programmierer meist mehr Komfort und Portabilität und können den Umgang mit Prozessen bzw. Threads oder die Parallelisierung sehr vereinfachen.

2.3.1 Prozess-basierte Programmiersysteme

Prozesse können ohne großen Aufwand zu betreiben nur auf einem Rechner gestartet werden. Steht einem jedoch ein Cluster oder ein anderes System mit mehreren CPUs zur Verfügung, die keinen gemeinsamen Speicher besitzen, so ist es sinnvoll mehrere Prozesse parallel auf diesen Rechnern laufen zu lassen, um ein bestimmtes Problem zu lösen. Hier ist explizit die Rede von Prozessen, da sich Threads nicht eignen, wenn kein gemeinsamer Speicher vorhanden ist. Das bedeutet natürlich nicht, dass die Prozesse eines Rechners nicht auch Threads verwenden können. Es existieren parallele Programmiersysteme, wie z.B. MPI und PVM, die Prozesse parallel auf mehreren Rechner starten können. Neben der Erzeugung der Prozesse auf den einzelnen

Rechnern ist vor allem die Synchronisation und Kommunikation der Prozesse eine wichtige Aufgabe solcher Systeme.

2.3.2 Thread-basierte Programmiersysteme

Auch wenn Threading-Systeme durch die Bereitstellung von Threads viele Vorteile bieten, haben sie immer noch einen großen Nachteil. Sie sind zumeist plattformabhängig (Betriebssystem, Programmiersprache, etc.) oder existierende Standards (z.B. POSIX) werden gar nicht oder nur halbherzig auf einer Plattform implementiert. Um Anwendungen so portabel wie möglich zu gestalten, wurden “Frameworks“, wie z.B. OpenMP (eigentlich ein aus Compilerdirektiven und Bibliotheksfunktionen bestehender Standard) oder Cilk geschaffen, die zum einen die Programmierung sehr vereinfachen (einige z.B. auch, indem sie mit Design Patterns arbeiten) aber auch auf eine große Anzahl von Plattformen portierbar sind. In vielen Fällen bauen diese Programmiersysteme auf bereits existierende Thread-Systeme wie die Pthreads auf.

2.4 Bewertungkriterien paralleler Programmierumgebungen

Eine Bewertung von Umgebungen zur parallelen Entwicklung kann beispielsweise anhand folgender Kriterien vorgenommen werden:

1. Performance:
    - Welche Geschwindigkeitszuwächse sind zu erzielen?
    - Können kurze Programmabschnitte oder Abschnitte mit geringer Parallelität effektiv beschleunigt werden?
    - Wie hoch sind die Kommunikationskosten im Vergleich zur eigentlichen Berechnung?
2. Sicherheit:
    - Wie stark muss sich der Programmierer darum kümmern, dass die parallelen Aktionen reibungslos ablaufen. Sie sich z.B. nicht gegenseitig blockieren oder zu Race-Conditions führen?
3. Funktionsumfang:
    - Welche Funktionalitäten zur Parallelisierung werden geboten?
    - Gibt es Funktionen, um Standardprobleme bei der Synchronisation, Kommunikation bzw. Interaktion insgesamt zu lösen?

4. Usability:

   - Ist die gebotene Funktionalität einfach oder sogar intuitiv anwendbar oder werden fundierte Kenntnisse benötigt, um die Umgebung verwenden zu können?
   - Wieviel Wissen über Parallelverarbeitung ist notwendig, um eine gute Parallelisierung zu erreichen?
   - Wie aufwändig ist die Parallelisierung?

2.5 Bewertung der Güte von Parallelität

Hat man ein paralleles Programm geschrieben, so stellt sich natürlich die Frage, ob sich die Parallelisierung überhaupt gelohnt hat. Es ist nicht unbedingt gesagt, das ein paralleles Programm schneller sein muss als ein sequentielles Programm. Oftmals ist es sogar der Fall, dass parallele Programme langsamer sind als ihre sequentielle Variante.

Es wäre also sinnvoll Parallelität beurteilen oder vergleichen zu können. In dieser Hinsicht ist vor allem interessant, wie gut das Programm auf einem Mehrprozessorrechner skaliert, d.h. wie viel schneller das Programm läuft, wenn man es auf mehreren Prozessoren gleichzeitig ausführt. Dies ist eines der wichtigsten Kriterien bei der parallelen Programmierung und wird Speedup genannt. Meist lassen sich jedoch nicht die Programme in ihrer Gesamtheit, sondern nur deren zentrale Algorithmen vergleichen. Dies liegt daran, dass Programme oft auf bestimmte Systemereignisse warten, was bei jeder erneuten Ausführung zu unterschiedlichen Ausführungszeiten führen kann. Die gemessenen Zeiten sind dann nicht vergleichbar. Natürlich müssen die zu vergleichenden Algorithmen auch vergleichbar sein. Zum Beispiel wäre es Unsinn, zwei Algorithmen zu vergleichen, bei denen der eine Algorithmus hundert Schleifendurchgänge durchführt und der andere tausend. Man muss daher bei den zu vergleichenden Algorithmen immer mit der gleichen Problemgröße arbeiten. In dem zuvor genannten Beispiel könnte man jeweils 1000 Schleifendurchgänge betrachten. Auch wenn sich diese Anmerkung trivial anhört, so merkt man spätestens bei auf Zufallszahlen beruhenden Algorithmen, dass eine Wiederholung mit der gleichen Problemgröße und somit gleichen Bedingungen nur bedingt möglich ist.

Man sollte ebenfalls beachten, welche Zeiten man misst. Für die Ermittlung von Speedups muss die sogenannte Wall-clock Zeit gemessen werden. Dies ist die reale Zeit, die seit dem Start der Zeitmessung vergangen ist. Würde man hingegen die CPU-Zeit messen, also die Zeit, die zur Berechnung von der CPU benötigt wird, so würde man bestenfalls die gleiche Zeit messen, die auch das sequentielle Programme benötigt.

Aus den vorherigen Betrachtungen sollte das Verständnis der Formel für den Speedup leicht fallen. Sei $t_s(n)$ die Zeit, die für das sequentielle (und nicht für das mit nur einem Prozessor ausgeführte parallele) Programm und einer Problemgröße n auf einer CPU benötigt wird. Sei $t_p(n)$ die Zeit, die für das parallele Programm mit gleicher Problemgröße auf p CPUs gebraucht

wird, so ist:

$$s_p(n) = \frac{t_p(n)}{t_s(n)}$$

der Speedup des Programmes bei Verwendung von p Prozessoren.
Gilt für alle betrachteten Prozessoranzahlen stets

$$s_p(n) = p,$$

so spricht man von einem ***linearen Speedup***. Die Ausführungsgeschwindigkeit des Programmes nimmt bei jeder zusätzlichen CPU im gleichen Maße zu. Dieses Verhalten ist das Ziel jedes Programmierers, der ein Programm parallelisieren will. Gilt sogar:

$$s_p(n) > p,$$

so spricht man gar von einem ***superlinearen Speedup***. Auch wenn dies auf den ersten Blick unmöglich scheint, so gibt es durchaus Fälle, in denen ein solcher Speedup gemessen werden kann. Der Grund dafür ist meist, dass jeder Prozessor seinen eigenen Cache besitzt und alle Caches der Prozessoren zusammen größer sind, als der eines einzigen Prozessors. Wenn nun die Prozessoren immer auf ihren lokalen Daten arbeiten und sich wenig mit den Daten in den Caches der anderen Prozessoren abgleichen müssen, so braucht der Prozessor kaum auf den langsamen Hauptspeicher zugreifen. Er kann die Daten einfach aus dem lokalen Cache holen. Arbeitet der Algorithmus mit sehr großen Datenmengen, so kann es sein, dass der Cache bei Ausführung auf einem Prozessor schnell überläuft. Bei mehreren CPUs kann der große Gesamt-Cache aufgrund der Lokalität viele Daten zwischenspeichern. Das Programm dann deutlich schneller laufen, da kaum auf den Hauptspeicher zugegriffen werden muss.

Linearer und superlinearer Speedup sind meist nur theoretisch erreichbar. Bei vielen Problemen wird man gar keinen Speedup erreichen. Die Ausführungszeit des Programmes entspricht dann der der sequentiellen Variante. In vielen Fällen ist das parallele Programm sogar langsamer. Dies liegt unter anderem daran, dass jeder Algorithmus einen konstanten Anteil von sequentiellen Programmcode besitzt, der nicht parallelisiert werden kann. Dies betrifft vor allem Programmabschnitte, in denen Tasks synchronisiert werden müssen. Diesen Sachverhalt beschreibt das ***Amdahlsche Gesetz***.

Sei f der prozentuale Anteil des nicht parallelisierbaren Programmcodes, also 0 <= f <= 1 und 1-f der Anteil des parallelisierbaren Codes. Dann ist die Laufzeit des sequentiellen Codes $t_s * f$ und die Laufzeit des parallelisierbaren Codes im besten Fall $t_s/p * (1 - f)$. Für den Speedup gilt somit:

$$s_p(n) = \frac{t_s(n)}{t_s(n) * f + \frac{t_s(n)}{p} * (1 - f)} = \frac{1}{f + \frac{1}{p} * (1 - f)} \leq \frac{1}{f}$$

Diese Formel besagt, dass selbst bei einer idealen Parallelisierung des parallelen Teils und der Verwendung unendlich vieler Prozessoren der maximal erzielbare Speedup beschränkt ist. Die Beschränkung ist abhängig von dem nicht parallelisierbaren Anteil im Programm.

2.6 Hardwarearchitekturen

Heutzutage gibt es eine Vielzahl von Parallelrechner-Architekturen, die eine Ausführung paralleler Tasks unterstützen. Die wohl bekannteste Klassifizierung dieser Architekturen stammt von M.J. Flynn [Fly72]. Dieser unterscheidet die folgenden vier Kategorien:

SISD (Single Instruction, Single Data) Hierbei handelt es sich um den klassischen von-Neumann Single-CPU Rechner. Jede CPU-Instruktion wird in sequentieller Ausführung auf einem eigenen Datensatz ausgeführt. Echte Parallelität kann mit dieser Architektur nicht erreicht werden. Mit ihnen können mehrere Tasks nur ***pseudo-parallel*** ausgeführt werden. Das heißt, dass der Prozessor so schnell zwischen den Tasks umschaltet, dass für den Anwender der Eindruck ensteht, die Tasks liefen parallel ab.

SIMD (Single Instruction, Multiple Data) Hierbei führen mehrere CPUs die gleiche Instruktion aus, jede der CPUs jedoch auf ihren eigenen Daten. In diese Klasse fallen z.B. Vektor- oder Matrix-Rechner.

MISD (Multiple Instruction, Single Data) Mehrere CPUs wenden eine Instruktion auf jeweils ihre eigenen Daten an und speichern das jeweilige Datum wieder zurück. Dies ist nur eine theoretisch mögliche Klasse. Eine praktische Relevanz hat sie jedoch nicht.

MIMD (Multiple Instruction, Multiple Data) Hierunter fallen die klassischen Parallelrechner. Jede der vorhandenen CPUs wendet hierbei eine individuelle Instruktion auf den eigenen Datensatz an. Hierbei wird noch einmal in Bezug auf die Lokalisierung der Speicher der einzelnen CPUs unterschieden. So besitzen SMP (Symmetric Multi-Processor) Rechner CPUs keinen lokalen Speicher und greifen über einen Bus lediglich auf gemeinsamen Speicher zu. Der Zugriff ist somit für alle CPUs gleich (symmetrisch). Zu dieser Kategorie zählen z.B. Multiprozessorsysteme, wie sie meist in Webservern verwendet werden. Auch die jüngste Entwicklung im Bereich der Multiprozessoren, die Multi-Core bzw. Dual-Core CPUs (im Prinzip mehrere bzw. zwei in ein gemeinsames Gehäuse integrierte CPUs)

2.7 Probleme der Parallelisierung

Je nach zu parallelisierendem Programm wird der Programmierer auf verschiedene Probleme treffen, die sich aus der Parallelität und dem gleichzeitigen Agieren der Tasks ergeben.

Synchronisierung Wechselwirkungen der Tasks untereinander, vor allem Zugriffe auf gemeinsame Daten müssen vom Programmierer koordiniert werden, um Programmfehler zu vermeiden. In vielen Fällen führen die Tasks Teilberechnungen durch, die sehr stark voneinander

abhängig sind. So könnte es sein, dass ein Task erst mit der Berechnung seiner Teilaufgabe beginnen darf, sobald ein anderer Task seine Berechnung beendet hat. Dieses Warten auf ein andere Task muss im Allgemeinen vom Programmierer erzwungen werden, ansonsten können Race-Conditions auftreten (siehe nächsten Abschnitt). Auch die Terminierung einer Berechnung und die Kommunikation der Tasks untereinander müssen durch Synchronisation geregelt werden.

Race-Conditions Die Ausführung eines parallelen Programmes darf nicht von der Geschwindigkeit der CPU oder der Tasks abhängen. Sollte das Programm unterschiedliche Ergebnisse liefern, weil bei einem Durchlauf ein Task einen bestimmten Programmabschnitt schneller erreicht haben sollte als bei einem vorherigen Durchlauf, so spricht man von einer Race-Condition. Insbesondere betrifft dies die Ausführungsgeschwindigkeit der Tasks untereinander. Race-Conditions können schnell bei fehlender Synchronisation und Zugriff auf gemeinsamen Speicher auftreten. Vorwiegend in Situationen, in denen beide Tasks eine gleiche Stelle im Speicher beschreiben, treten Race-Conditions. Ein oft genanntes Beispiel ist die gleichzeitige Kontoabbuchungen von einem Konto durch zwei Tasks. Ein Konto enthält 50 €. Der erste Task will hiervon 3 € abziehen, der zweite 2 €. Der erste Task beginnt und liest den aktuellen Stand von 50 €. Der Task wird unterbrochen und Task zwei liest ebenfalls den Stand von 50 €. Er zieht 2 € ab, so dass nun 48 € auf dem Konto liegen und hat seine Ausführung beendet. Der erste Task wird wieder aktiv und zieht die 3 € von den zuvor gelesenen 50 € ab. Der zuvor geschriebene Kontostand von 48 € wird mit dem angeblich neuem von 47 € überschrieben. Die Bank hat durch ihr schlecht programmiertes Abbuchungsprogramm 2 € Verlust gemacht. Nicht gerade viel für eine Bank. Doch was wäre, wenn es um Beträge in Millionenhöhe geht?

Ein großes Problem bei Race-Conditions ist, dass sie nicht immer auftreten, sondern nur nach ein paar Durchläufen des Programmes, evtl. sogar jahrelang nicht. Solche Fehler sind nur schwer reproduzierbar. Es kann auch nur sehr schwer getestet werden, ob eine Änderung am Programm den Fehler wirklich behoben hat. Debugging ist daher nur sehr umständlich oder gar nicht möglich. Man sollte daher auf saubere Programmierung und Synchronisierung achten, um Race-Conditions im vorhinein ausschließen zu können.

Um Race-Conditions zu vermeiden, müssen folgende Punkte beachtet werden:

1. Zugriffe auf gemeinsamen Speicher müssen synchronisiert werden
2. Es dürfen keine Annahmen gemacht werden, wie lange die CPU oder ein Task für bestimmte Abschnitte des Programmes benötigt.

Deadlocks Deadlock ist die englische Bezeichnung für Verklemmung und bezeichnet eine solche von Tasks. Eine Verklemmung kommt dann Zustande, wenn ein Task A auf eine Bedingung wartet, die nur von einem anderen Task B erfüllt werden kann. Wartet Task B wiederum auf eine Bedingung, die nur von Task A erfüllt werden kann, so können beide Tasks unend-

lich lange aufeinander warten, bis sie weiter rechnen können – Sie sind verklemmt. Das Programm muss daraufhin neu gestartet werden. Es muss aber nicht sein, dass die beiden Tasks direkt aufeinander warten. Es kann auch ein Zyklus von Warteabhängigkeiten vorhanden sein, an dem mehrere Tasks beteiligt sind, z.B. wenn A auf B, B auf C und C wiederum auf A warten muss. Dementsprechend komplex kann die Erkennung von Deadlocks (z.B. mittels Banker-Algorithmus) sein. Deshalb wird meist ganz auf eine Deadlock-Erkennung verzichtet. Der Programmierer sollte daher besonders darauf achten, Vorbedingungen für Deadlocks gar nicht erst entstehen zu lassen. Hiermit beschäftigt sich vor allem die Literatur über Betriebssysteme im Rahmen der Prozessverwaltung.

Lastenbalancierung Im Bereich der Parallelprogrammierung ist man immer bestrebt, die Prozessoren von Multiprozessorsystemen möglichst effizient zu nutzen. Hierzu muss die Arbeit gleichmäßig auf die Prozessoren aufgeteilt werden, was man mit Lastenbalancierung (engl.: Load-Balancing) bezeichnet. Betrachten wir zum Beispiel ein Programm, dass aus mehreren unterschiedlich großen unabhängigen Teilaufgaben (Tasks) besteht. Hat man zwei Prozessoren zur Verfügung und gibt einem Thread die großen Teilaufgaben und dem anderen die gleiche Anzahl kleinerer Teilaufgaben, so wird der Thread mit den kleineren Tasks im Normalfall als erstes fertig sein. Der andere Thread rechnet weiterhin an seinen Aufgaben und könnte gut Hilfe gebrauchen. Es wird lediglich eine CPU verwendet, die andere liegt brach; Die Folge einer schlechten Lastenbalacierung. Eine bessere Lastenbalancierung durch intelligentere Verteilung der Teilaufgaben auf die Threads hätte zur Folge, dass beide Prozessoren gleichmäßig ausgelastet sind. Dadurch ist der Zeitbedarf für die Bewältigung der Aufgaben mit guter Lastenbalancierung sehr viel geringer als in dem vorigen Fall mit schlechten Lastenbalanzierung.

2.8 Kritische Abschnitte

Bei der parallelen Programmierung ergibt sich das Problem, dass mehrere Prozesse oder Threads *zur gleichen Zeit* auf gemeinsame Ressourcen zugreifen können. Greift ein Thread auf eine gemeinsame Ressource (meistens Variablen) zu, so geschieht das im Allgemeinen nicht atomisch[1], d.h. es kann ein Thread inmitten seiner Ausführung unterbrochen werden und ein anderer Thread die Kontrolle über die CPU übernehmen. Dieser könnte nun ebenfalls versuchen, die Ressource zu verändern, eine typische Race-Condition. Durch Synchronisierung der Prozesse und Threads muss verhindert werden, dass Race-Conditions oder Deadlocks entstehen. Bei parallelen Prozessen müssen meist “nur“ E/A-Anfragen für gemeinsam verwendete Geräte synchronisiert werden. Komplizierter wird es bereits, wenn mehrere Prozesse einen gemeinsamen Speicherbereich vom Betriebssystem anfordern, da auch Zugriffe hierauf synchronisert werden müssen. Nun ist es aber bei parallelen Threads (innerhalb eines Prozesses) immer so, dass sie auf einem gemeinsamen Adressraum arbeiten, womit auch Zugriffe auf gemeinsame

[1] In der Literatur wird anstatt des Ausdrucks “atomisch“ meist “atomar“ gebraucht. Beide Worte sollen jedoch das gleiche ausdrücken.

Daten synchronisiert werden müssen. Das heißt, dass Zugriffe auf gemeinsame Variablen oder, allgemeiner formuliert, Speicherstellen vom Programmierer nur dann erlaubt werden dürfen, wenn nicht gleichzeitig ein anderer Thread den Wert der Speicherstelle verändert. So dürfen niemals zwei Threads gemeinsam eine Speicherstelle beschreiben. Auch das Lesen einer Speicherstelle, während sie von einem anderen Thread beschrieben wird, ist nicht erlaubt. Es könnte andernfalls der gelesene Wert während dem Lesevorgang verändert werden und Race-Condition eintreten. Lediglich parallele Leseoperationen auf gemeinsamen Speicher sind erlaubt. Bereiche in der Programmausführung, in denen nur ein oder eine bestimmte Anzahl von Threads gleichzeitig aktiv sein dürfen, um Race-Conditions zu vermeiden, nennt man ***kritischer Abschnitt***. Z.B. können in einem solchen Abschnitt Zugriffe auf gemeinsame Variable geschehen.

Kapitel 3

Threading-Systeme im Überblick

Mittlerweile gibt es eine unüberschaubare Anzahl von Threading-Systemen. Darunter bekannte wie Win32-Threads oder Pthreads und weniger bekannte wie Balder-Threads, Nano-Threads oder die Threads der GLib. Alle diese Systeme und deren Besonderheiten zu erfassen würde den Rahmen dieser Arbeit sprengen. Viele Threading-Systeme bieten Multithreading nur als zusätzliches Feature an, das jedoch aufgrund fehlender Funktionalität nur eingeschränkt oder evtl. gar nicht einsetzbar ist. Andere Threading-Systeme wiederum sind lediglich Wrapper um bereits existierende Threading-Systeme und delegieren Aufrufe einfach nur an die darunter liegende Bibliothek. Wieder andere Threading-Systeme besitzen zwar eine eigene Implementierung, sind jedoch von der Handhabung so eng an andere Threading-Systeme angelehnt, dass ein großer Unterschied nicht mehr erkennbar ist. Dies betrifft vor allem die Erzeugung von Threads und die Synchronisation von Threads, die in vielen Threading-Systemen ähnlich gehandhabt wird.

Es gibt jedoch einige Systeme, die Pionierarbeit im Bereich von Multi-Threading geleistet haben und neue Mechanismen etabliert haben, die in einigen Fällen sogar in weitere Threading-Systeme übernommen wurden. In diesem Kapitel sollen nun die Threading-Systeme vorgestellt werden, auf die zum Vergleich der Threading-Mechanismen in späteren Kapiteln des öfteren als Beispiele aufgeführt werden. Bei der Auswahl wurde Wert darauf gelegt, dass es sich um Systeme handelt, die möglichst viele Mechanismen anbieten oder Aspekte der Multi-Threading Programmierung völlig anders lösen, als andere Systeme. Außerdem sollte der aktuelle Stand der Threading-Systeme repräsentiert werden, was sich darin niederschlägt, dass fast nur Systeme betrachtet werden, die in den letzten beiden Jahrzehnten eingeführt wurden. Pioniersysteme, wie sie in diesem Abschnitt bereits erwähnt wurden, werden in einigen Fällen zwar kurz angesprochen aber nicht vertiefend behandelt. Zum einen deshalb, weil es sich meist um Prototypen handelt oder der Quelltext nicht offen verfügbar ist. Zum anderen weil diese Systeme auf modernen Rechnern einfach nicht mehr zum Laufen gebracht werden können und somit die Praxistauglichkeit der Systeme fehlt.

In den späteren Kapiteln werden aber auch Konzepte von Threading-Systeme betrachtet, die in diesem Kapitel nicht vorgestellt werden. Dabei handelt es meist um solche Systeme,

die in einem gewissen Bereich des Multithreading eine Sonderstellung einnehmen, ansonsten jedoch keine weiteren bedeutenden Unterschiede aufweisen oder in Hinsicht auf Multi-Threading Programmierung bedeutungslos sind. So hat Ada ein interessantes Rendezvous-Synchronisationsprinzip, ist aber eigentlich kein Threading-System im herkömmlichen Sinn, da nur Tasks betrachtet werden, die entweder als Prozesse oder Threads gestartet werden können.

3.1 POSIX-Threads

Die POSIX-Threads oder kurz Pthreads sind eines der am weitesten verbreiteten Threading-Systemen. Bei den Pthreads an sich handelt es sich jedoch nicht um eine konkrete Implementierung eines Threading-Systems, sondern um einen Standard für eine solche. Genau genommen sind Pthreads kein eigenständiger Standard, sondern Teil des POSIX[1] (Portable Operating System Interface for Unix) Standards, daher auch der Name. Der POSIX-Standard wurde durch die IEEE (Institute of Electrical and Electronics Engineers) mit dem Ziel einer einheitlichen Programmierschnittstelle (API) für Unix-Systeme spezifiziert. Diese kann man sich als eine Bibliothek mit genormten Interface vorstellen, bei der sich alle Funktionen, die durch das Interface definiert werden genau so verhalten, wie es der Standard vorsieht. So ist z.B. definiert, welche Funktionen (darunter Mathematik-Routinen und sogar Systemaufrufe) es geben muss und unter welchem Namen und mit welchen Parametern die Funktionen aufgerufen werden. Diese Bibliothek wird dann entweder mit dem jeweiligen Betriebssystem mitgeliefert oder nachträglich installiert. Verwendet ein Programm anstatt vom Betriebssystem abhängender Funktionen nur noch die systemunabhängigen POSIX-Funktionen, kann das Programm, ohne umgeschrieben werden zu müssen, in der Theorie auf allen anderen POSIX-konformen Betriebssystemen laufen. Im POSIX-Standard wird nicht nur spezifiziert, welche Funktionsaufrufe es geben muss, sondern auch was sie bewirken müssen. Einige Funktionen sind optional und müssen nicht von der Bibliothek implementiert werden. Andere sind in ihren Auswirkungen nicht eindeutig beschrieben. Das hat zur Folge, dass die Implementierung mehr Interpretationsfreiraum haben, worunter jedoch die Portabilität leidet. Die Portabilität wird durch die große Anzahl von Standards und Versionen weiter erschwert. Dies wird im nächsten Abschnitt etwas deutlicher (siehe auch 4.1). Im POSIX-Standard werden die Schnittstellen als C-Funktionsköpfe definiert, allerdings kann man mit einem geeignetem Interface auch mit Fortran, Ada oder sogar mit Perl und Java POSIX-Bibliotheken verwenden. POSIX ist, auch wenn der Name es vermuten ließe, nicht auf UNIX beschränkt. Selbst Windows NT [wik06c] ist zu einem gewissen Grad POSIX-kompatibel. Bei WindowsXP wird die POSIX-Bibliothek jedoch nicht mehr standardmäßig in die Distribution eingebunden. Man muss sich hierzu erst eine von Microsoft angebotene Unix-Laufzeitumgebung namens Interix beschaffen. Diese umfasst sogar eine Minimalversion der

[1] Die offizielle POSIX-FAQ zur korrekten Aussprache von POSIX: “It is expected to be pronounced pahz-icks, as in positive, not poh-six, or other variations. The pronunciation has been published in an attempt to promulgate a standardized way of referring to a standard operating system interface.“ [aus06]

Pthreads. Es gibt jedoch auch OpenSource Projekte, die POSIX-Funktionalität für NT-Systeme bieten.

Im Jahr 1988 wurde die erste POSIX-Version als IEEE Standard 1003.1-1988 verarbschiedet. Heutzutage versteht man unter POSIX jedoch die Gesamtheit der IEEE 1003 Standards, die momentan die Standards IEEE 1003.1 bis 1003.26 umfassen. Im Rahmen der Pthreads ist eigentlich nur der Standard 1003.1 interessant. Erst im Jahr 1995 wurde Threading-Funktionalität mit Standard IEEE 1003.1c in POSIX aufgenommen. Spricht man von Pthreads, so meint man diesen Teil des POSIX-Standards. Somit kann man IEEE 1003.1 als den eigentlichen Pthreads-Standard oder zumindest seine Grundlage ansehen. Daneben gibt es jedoch noch weitere Teile des POSIX-Standards, die man zu den Pthreads zählen kann. So z.B. IEEE 1003.1b, IEEE 1003.1d und IEEE 1003.1j für Echtzeit-Erweiterungen, von denen einige auch unabhängig von den Pthreads verwendet werden können. Hierzu gehören z.B. die in der parallelen Programmierung sehr häufig eingesetzten Semaphore (IEEE 1003.1b), Mutexe mit Timeout (IEEE 1003.1d) oder Barrieren und Spinlocks (IEEE 1003.1j). Zusätzlich zu der bereits sehr verwirrenden großen Anzahl von Standards werden diese des öfteren überarbeitet oder um neue Funktionalität ergänzt. Die Version des Standards wird durch das Anhängen der Jahreszahl der Überarbeitung ersichtlich. So ist IEEE 1003.1-1988 die initiale Version von 1988 und IEEE 1003.1-2004 die aktuelle Version von 2004. Viele Bibliotheken geben zwar an, den POSIX-Standard einzuhalten, tun dies jedoch nicht wirklich oder nur teilweise. Daher sollte man darauf achten, dass eine Implementierung POSIX-konform (“POSIX conformance“ genannt) ist, also den Standard in seiner Gesamtheit unterstützt. Werden nur Teile des Standards unterstützt, wird dies als “POSIX compliance“ bezeichnet (vgl. [Doe]). Am besten ist es, wenn die Bibliothek von einer authorisierten Stelle als POSIX-konform zertifiziert wurde. Mit einem kleinen Testprogramm [pos06] kann man die Konformität einer Implementierung testen. Allerdings sollte man das Ergebnis eher mit Vorsicht genießen.

Auch wenn diese etwas eingehendere, jedoch noch längst nicht erschöpfende, Betrachtung der POSIX-Standards auf den ersten Blick im Zusammenhang mit dieser Arbeit unnötig erscheint, ist dieses Grundwissen jedoch notwendig, um in späteren Kapiteln einige Nachteile der Pthreads z.B. im Bereich der Portabiliät besser verstehen zu können. So sind verschiedene Implementierungen des POSIX-Standards, selbst wenn sie den gleichen Standard abdecken, in der Praxis meist doch nicht vollständig kompatibel. Pthread-Bibliothek ist also nicht gleich Pthread-Bibliothek. Für weitere Informationen zu POSIX soll auf [pas06] verwiesen werden. Auf der genannten Internetseite findet man eine Übersicht über alle bisherigen POSIX-Standards und Versionen. Generelle Informationen zum POSIX-Standard kann man auf [aus06] finden. Der offizielle POSIX-Standard kann kostenfrei im HTML-Format unter [uni06] eingesehen oder kostenpflichtig als PDF bezogen werden.

Nun zu den eigentlichen Pthreads. Hier gilt natürlich das gleiche, wie für den POSIX-Standard allgemein. Es gibt unzählige Pthread-Implementierungen für alle möglichen Betriebssyteme und Programmiersprachen. Auch wenn die Pthreads zum POSIX-Standard gehören, erfolgt die Implementierung meist getrennt von den anderen im Standard definierten Funk-

tionen. So gibt es auch Pthread-Bibliotheken für Systeme, die über keine POSIX-Bibliothek verfügen, wie dies bei den meisten Versionen von Windows der Fall ist. Auch wenn Windows keine eigene Pthreads-Implementierung mitliefert, hat man mit der Bibliothek Pthreads-Win32 [Pth06] die Möglichkeit dies nachzuholen. Zwar bietet Windows eigene Threading-Systeme an, diese sind aber z.B. nicht portabel auf Unix-Systeme. Verwendet man hingegen Pthreads auch unter Windows, braucht man zumindest den Threading-Anteil des Programmes nicht mehr für Unix-Systeme umzuschreiben. Die weitaus größere Bedeutung haben Pthreads jedoch auf Unix-Systemen. Hier sind die Pthreads das beliebteste Threading-System und werden standardmäßig mit der GNU C-Bibliothek (glibc) ausgeliefert. Eine Standard-Implementierung gibt es unter den Unix-Systemen jedoch nicht. So fand erst vor kurzer Zeit unter Linux mit dem Wechsel von der Kernelversion 2.4 auf 2.6 auch ein Austausch der Pthread-Implementierung statt. Die ältere Pthread-Bibliothek namens LinuxThreads wurde durch die maßgeblich von RedHat entwickelte NPTL-Bibliothek (Native POSIX-Threads Library) ersetzt, da LinuxThreads den Anforderungen eines modernen Threading-Systems nicht mehr genügte. Dazu musste sich NPTL jedoch erst gegen den Konkurrenten NGPT (Next Generation POSIX Threading) von IBM durchsetzen, der ursprünglich als Nachfolger der LinuxThreads gedacht war. Aufgrund eines effektiveren Designs (siehe auch 2.2.4 konnte sich NPTL jedoch durchsetzen und NGPT in die Bedeutungslosigkeit verdrängen. NPTL bietet im Gegensatz zu seinem Vorgänger schnellere Thread-Erzeugung und eine bessere Unterstützung des POSIX-Standards. Waren die alten LinuxThreads noch nicht POSIX-konform, so wurde dies mit NPTL zumindest angestrebt. Allerdings wurde bereits ein Bug gefunden [npt06], durch den NPTL gegen den POSIX-Standard verstößt und somit auch NPTL nicht POSIX-konform ist. Programmcode, der die alten LinuxThreads verwendet ist nicht mehr 100% kompatibel zu den neuen NPTL-Threads und umgekehrt. Der vorige Punkt ist gutes Beispiel dafür, dass die Portabilität in vielen Fällen nur theoretischer Natur ist. Pthreads sieht außerdem auch vor, dass zusätzliche implementierungsspezifische Funktionen eingeführt werden dürfen, die dann jedoch mit der Endung "_np" (non portable) versehen werden müssen. So existiert mit pthread_mutex_getowner_np() unter AIX eine Funktion zur Bestimmung des Threads, der gerade im Besitz eines bestimmten Mutex ist. Diese Funktion ist jedoch nicht portabel.

Pthreads bieten eine sehr klar strukturierte Schnittstelle und sind vom Funktionsumfang eines der vielfältigsten Threading-Systeme. So gehören Mutexe, Semaphoren, Bedingungsvariablen, Barrieren, Spin-Locks und Read-Write-Locks zu den angebotenen Funktionalitäten. Allerdings sind einige dieser Features im Standard entweder als optional gekennzeichnet oder erst in späteren Versionen spezifiziert worden. Somit müssen sie auch nicht in allen Implementierungen vorhanden sein. Vor allem für Entwickler zeitkritischer Systeme wie z.B. Embedded Systems dürften die Echtzeiterweiterungen der Pthreads interessant sein. Hiermit können Threads mit hohen Prioritäten angelegt werden, die viel besser und vor allem schneller auf Ereignisse im System reagieren können als normale Threads. Trotz des großen Funktionsumfangs wirken die Pthreads jedoch aufgeräumt, da auf unnötige Funktionen verzichtet wurde. Einige Threading-Systeme, darunter auch Win32, bieten eine Vielzahl von Versionen der ein-

zelnen Funktionen an, um unterschiedliche Ausprägungen dieser Funktionen zu ermöglichen. Durch die große Anzahl von Funktionen wird es schwer, einen Überblick über die Threading-Bibliothek zu erhalten. Pthreads bieten nur eine Variante, die für die meisten Anwendungsfälle ausreicht. Diese Funktionen bieten für den Fall, dass ein spezifischeres Verhalten gewünscht ist, die Möglichkeit Attribut-Variablen als Parameter zu übergeben. So kann bei der Erzeugung eines Threas mittels eines Attributs die Schedulingstrategie gewählt werden. Wird kein Attribut übergeben, wird der normale Scheduling-Algorithmus verwendet. Die Pthreads sind eigentlich auf die Programmiersprache C zugeschnitten. Sie können aber auch mit einer geeigneten Bibliothek in anderen Programmiersprachen wie Fortran eingesetzt werden. Es ist schwer, allgemeines über die Architektur der Pthreads zu sagen, da diese weitestgehend von der Implementierung abhängt. Pthreads können sowohl eine reine User-Level als auch eine Kernel-Level Bibliothek sein, wobei bei der letzten Möglichkeit alle Funktionsaufrufe an den Kernel weiterdelegiert werden. Bei den NPTL-Threads werden z.B. Kernel-Level Threads verwendet. Da die Programmierung der NPTL Threading-Bibliothek und des Linux-Kernel sehr eng koordiniert wird, kann vieles der Pthread-Funktionalität, wie die Thread-Erzeugung, durch Systemaufrufe realsiert werden. Auch das Threading-Modell ist unter den Implementierungen nicht einheitlich. So kann man vom 1:1 bis zum M:N Modell alles finden.

Zum Abschluss der Betrachtung der Pthreads soll ein kleines Beispiel-Programm betrachtet werden:

```
1  #include <pthread.h>

3  void* thread_start(void* arg) {
4      /* Arbeite */
5      ...
6      pthread_exit(NULL);
7  }

9  int main() {
10     pthread_t thread;

12     /* Erzeuge und starte Pthread */
13     pthread_create(&thread, NULL, thread_start, NULL);
14     /* Warte auf die Beendigung des Threads */
15     pthread_join(thread, NULL);

17     return 0;
18 }
```

Beispiel 3.1: Ein einfaches Beispiel mit Pthreads

3.2 Java-Threads

Hierbei handelt es sich nicht um eine eigenständige Bibliothek, sondern um einen Teil der Standardbibliothek (Java-API) der Java-Plattform. Die Java-Plattform selbst besteht nicht nur aus der Java-API, sondern auch aus der Java Virtual Machine (JVM) (vgl. [wik06a]). Diese JVM kann man sich als virtuellen Rechner vorstellen. Java-Programme, die in der Java-Programmiersprache programmiert wurden, werden in Maschinencode für die JVM (Java-Bytecode) und nicht in maschinenabhängigen Programmcode des darunter liegenden realen System übersetzt. Die Komponenten der Java-Plattform werden in Form der Java Runtime Environment (JRE) genannten Laufzeitumgebung angeboten. Java-Programme können dann auf allen Rechnern ausgeführt werden, die eine JRE installiert haben. Diese Aussage muss man allerdings etwas einschränken, da Java zwar abwärtskompatibel aber nicht aufwärtskompatibel ist. Programme, die für neuere Versionen der Java-Plattform erstellt wurden, können nicht auf Rechnern ausgeführtf werden, die eine ältere JRE-Version benutzen. Die Version 1.0 von Java wurde im Jahr 1996 veröffentlicht, Java ist also noch relativ jung. Seit Ende 2004 ist die Version 1.5 von Java aktuell (besser bekannt als Java 5.0). Aufgrund größerer Änderungen (u.a. der Einführung von Generics) in Java 5.0 wird es vorraussichtlich noch einige Zeit dauern, bis sich Java 5.0 gegen den Vorgänger Java 1.4 durchsetzt. Seit Version 1.2 (bekannt als Java 2) werden außerdem drei Editionen der Java-Plattform unterschieden: J2ME (MicroEdition), J2SE (Standard Edition) und J2EE (Enterprise Edition). Hierbei ist SE die Desktop-Variante und EE eine für kommerzielle Anwendungen erweiterte Form von SE. Im Rahmen dieser Arbeit werden nur J2SE 1.4 und J2SE 5.0 betrachtet. Es gibt zudem verschiedene Implementierungen der JRE. So gibt es neben der bekanntesten Implmentierung von Sun auch noch das Blackdown Projekt. Im weiteren Verlauf der Arbeit soll nicht mehr zwischen den einzelnen Komponenten von Java unterschieden werden. Mit Java ist dann je nach Zusammenhang entweder die Java-API oder die Programmiersprache Java gemeint. Bei der Programmiersprache Java handelt es sich um die Standardsprache der Java-Plattform. Sie ist objektorientiert und speziell auf die Java-Plattform zugeschnitten. Auch wenn Java-Programme so gut wie immer in Java geschrieben werden ist jedoch nicht selbstverständlich. Mit einem geeigneten Compiler könnte man Quelltext jeglicher Programmiersprachen in Bytecode übersetzen und auf der JVM ausführen lassen, darunter in C++ geschriebener Quelltext. Da solche Compiler eher ein Nischendasein führen, stellt dies eher die Ausnahme dar. Im Rahmen dieser Arbeit wird daher mit einem Java-Programm immer die Programmiersprache Java verbunden.

Was bedeutet das alles in Hinsicht auf Java-Threads? Wie bereits erwähnt, sind Java-Threads fest in die Standardbibliothek integriert und das bereits seit Java 1.0. Allerdings gibt es aufgrund der unterschiedlichen Implementierungen der JRE keine einheitliche Implementierung der Java-Threads. Es ist somit nicht möglich, allgemein gültig zu sagen, wie die Threads letztendlich implementiert sind oder welches Threading-Modell verwendet wird. In älteren Versionen wurden noch keine Kernel-Level Threads verwendet, sondern Threads lediglich als User-Level Threads implementiert, die Green-Threads genannt wurden. Ein Java-Programm konnte lediglich eine

CPU des Rechners ausnutzen, selbst wenn es sich um ein Multiprozessorsystem handelte. Die heute verfügbaren Versionen basieren jedoch meist auf Kernel-Level Threads und bieten daher auch Multiprozessor-Unterstützung. Echtzeitfähigkeit sollte man jedoch (noch) nicht erwarten, zumindest solange man eine Software-JVM verwendet. Der Umstieg auf J2SE 5.0 ist auch im Bereich des Multi-Threading von besonderer Bedeutung, denn mit der neuen Version wurde ein neues Package namens java.util.concurrent unter der Schirmherrschaft von Doug Lea eingeführt, das viele neue Multi-Threading Features wie Task-Pools, Message-Queues oder Atomic-Locks bietet. Viele dieser neuen Features werden in späteren Kapiteln dieser Arbeit Erwähnung finden.

Multi-Threading in Java hebt sich vor allem in einer Hinsicht von anderen Threading-Systemen ab. Das liegt daran, dass in Java das Konzept der Monitore zur Synchronisation von Threads bereits in die Programmiersprache integriert ist. Hiermit können Methoden durch die Angabe des Schlüsselwort “synchronized“ als kritischer Abschnitt behandelt werden. Es sind auch kritische Abschnitte definierbar, die durch synchronized-Blöcke gekennzeichnet werden müssen. In anderen Sprachen können Monitore meistens nur durch spezielle Funktionsaufrufe simuliert werden. Weitere Informationen zu Monitoren findet man im Kapitel 4.6. Die Hauptfunktionalität des Multi-Threadings war bis zur Einführung von Java 5.0 lediglich auf die Klasse Thread und das Interface Runnable aufgeteilt. Da es ein Designziel von Java ist, die API möglichst kompakt zu halten, fällt die Feature-Liste der Java-Threads entsprechend klein aus. Neben Monitoren und Bedingungsvariablen findet man im Bereich Threading keine weiteren nennenswerten Features. Sowohl Mutexe als auch Semaphore existieren nicht, können jedoch leicht mit Hilfe von Monitoren nachgebildet werden. Durch die Einführung von Java 5.0 hat sich dies jedoch grundlegend geändert. Java besitzt mittlerweile eine der umfangreichsten Threading-Bibliotheken.

Wie erzeugt man aber einen Thread? Hierzu reicht es, ein Objekt einer von Thread abgeleiteten Klasse zu erzeugen. Allerdings muss der Thread nun noch gestartet werden. Dazu ruft man die von der Klasse Thread geerbte `start()`-Methode auf. Ein Ableiten von Thread ist oftmals nicht möglich oder sinnvoll. Deshalb kann man auch das Interface Runnable in einer Klasse implementieren und eine Instanz dieser Klasse erzeugen. Allerdings muss dann zusätzlich ein Objekt der Klasse Thread erzeugt werden, um den Thread starten zu können. Man sollte beachten, dass ein Objekt der Klasse Thread nicht mit dem eigentlichen User- oder Kernel-Level Thread übereinstimmt. Der erzeugte Thread führt lediglich den Programmcode des Thread-Objekts aus. Ruft das Thread-Objekt Methoden anderer Thread-Objekte auf, so führt der Thread auch den Code des anderen Thread-Objektes aus.

```
1 class MyThread
2 implements Runnable {
3     public void run() {
4         // Arbeite
5     }
6 }
```

```
 8 public class ThreadExample {
 9     public static void main(String[] args) {
10         // Thread erzeugen und starten
11         Thread thread = new Thread(new MyThread());
12         thread.start();
13     }
14 }
```

Beispiel 3.2: Ein einfaches Beispiel mit Java-Threads

3.3 .NET-Threads

.NET kann als Microsofts Antwort auf Java angesehen werden. Das Design ähnelt sehr stark der Java-Plattform. Mit .NET ist im Prinzip das .NET-Framework gemeint, wobei es sich um eine Plattform handelt, die aus einer API und einer virtuellen Maschine besteht. Das Äquivalent des Java-Bytecodes in .NET ist die Microsoft Intermediate Language (MSIL). Alle Programmiersprachen, die .NET unterstützen, werden von einem auf die jeweilige Sprache zugeschnittenen Compiler in diese Zwischensprache übersetzt. Der MSIL-Code wird dann, analog zur JVM in Java, von einer virtuellen Maschine namens Common Language Runtime (CLR) ausgeführt. Durch dieses Konzept kann man in jeder Programmiersprache, die von .NET unterstützt wird, Programme entwickeln. Nach der Übersetzung in die MSIL lassen sich die erzeugten Programme auf jedem System ausführen, auf dem das .NET Framework installiert ist. Zu den unterstützten Programmiersprachen gehören C#, Microsofts Java-Variante J#, Visual Basic, Delphi und natürlich C++. Programme, die in der Sprache C++ geschrieben werden, können entweder im unmanaged-Mode, quasi das gewohnte C++, oder im managed-Mode implementiert werden. Der managed-Mode hat nur noch wenig mit dem ursprünglichen C++ zu tun. Pointer werden hier mittels Garbage-Collection automatisch vom Framework verwaltet, außerdem kommen einige .NET-spezifische Sprachkonstrukte wie Delegates hinzu. Managed C++ ähnelt eher C# als dem ursprünglichen C++. Man sollte besser die Programmiersprache C# verwnden, die sozusagen die Hauptsprache des .NET-Frameworks darstellt und sich vom Aufbau an C++, Java und Delphi orientiert. Die Portabilität von .NET-Anwendungen ist trotz der virtuellen Maschine beschränkt, da Implementierungen des .NET von Microsoft nur für Windows angeboten werden und Linux nicht unterstützt wird. Für Linux gibt es jedoch eine OpenSource Implementierung von .NET namens Mono, die weitestgehend kompatibel zu .NET ist.

Seit der Einführung des .NET-Framework wird auch Multi-Threading unterstützt (Namespace System.Threading). Anders als in Java spielen Monitore in .NET keine allzu große Rolle. Der Compiler kann zwar angewiesen werden, eine Methoden zu synchronisieren, allerdings gibt es dafür kein eigenes Sprachkonstrukt, wie dies mit synchronized in Java der Fall ist. In .NET muss man sich mit sogenannten Attributen behelfen, die sehr stark an Compiler-Direktiven erinnern. In .NET werden jedoch stattdessen vorzugsweise Locks verwendet. So bietet .NET

analog zu den synchronized-Blöcken in Java lock-Blöcke an, die ein als Parameter übergebenes Objekt sperren. Im Prinzip unterscheidet sich hier nur der Name des Konstrukts. Neben Locks bietet .NET eine Vielzahl von Threading-Funktionalitäten, darunter Mutexe, Thread-Pools, ReadWrite-Locks und seit .NET 2.0 auch Semaphore.

```
1  using System;
2  using System.Threading;
3
4  namespace ThreadExample {
5      class MyThread {
6          public void Start() {
7              // Arbeite
8          }
9      }
10
11     class MainClass {
12         static void StaticStart() {
13             Console.WriteLine("Static Thread started!");
14         }
15
16         static void Main(string[] args) {
17             // Erzeuge einen Thread aus einer Klassen-Methode
18             Thread staticThread = new Thread(
19                 new ThreadStart(StaticStart));
20             // Erzeuge einen Thread aus einer Objekt-Methode
21             Thread otherThread = new Thread(
22                 new ThreadStart(new MyThread().Start));
23             // Starte die Threads
24             staticThread.Start();
25             otherThread.Start();
26             // Warte auf die Threads
27             staticThread.Join();
28             otherThread.Join();
29         }
30     }
31 }
```

Beispiel 3.3: Ein einfaches Beispiel mit .NET-Threads

3.4 Windows-Threads

Multi-Threading wurde in Windows Mitte der 90er Jahre mit der Win32-API eingeführt, die speziell für 32-bit Architekturen entwickelt wurde. Mit Win32 vollzog sich damit bei Micro-

soft der Wechsel von 16-bit auf 32-bit Betriebssysteme. WindowsNT 3.1 war 1993 das erste Windows-Betriebssysteme der 32-bit Generation gefolgt von Windows95. Bei dem 16-bit System Windows 3.1 hatte man das Augenmerk vor allem auf Multi-Tasking, also die gleichzeitige Ausführung mehrerer Programme zur gleichen Zeit, gelegt. Problematisch hierbei war vor allem, dass Programme non-preemptive ausgeführt wurden, d.h. dass der Wechsel zu einem anderen Programm nur geschehen konnte, wenn das Programm entweder die Kontrolle freiwillig abgab oder durch E/A-Operationen blockierte. Gab ein Programm die Kontrolle nicht mehr ab, konnte das gesammte System einfrieren. Mit Win32 wurde das Problem zwar weitestgehend gelöst, man musste sich jedoch neuen Herausforderungen stellen. Zum einen wurde im Heimbereich das Verlangen nach Multimedia immer größer, zum anderen begann der Boom des Internets, weshalb immer mehr Firmen Webserver betreiben können. Diese müssen eine große Zahl von gleichzeitigen Anfragen nach Internetseiten effizient erfüllen können. In diesen beiden Bereichen wird viel mit Threads gearbeitet, was mitunter ein Grund für die Einführung von Threads in Windows sein dürfte. Abgesehen davon bietet allein die Verwaltung einer grafischen Oberfläche mannigfache Möglichkeiten, um Threads zur Optimierung einzusetzen. Die Threading-Funktionalität, auch bekannt als Win32-Threads, wurde direkt in die Kernel32.dll eingefügt und steht in allen neueren Betriebssystemen, wie Windows95/98/Me und der WindowsNT-Familie, darunter auch WindowsXP und Windows-Server, zur Verfügung. Nach [SGG03, S. 143] verwenden zumindest Windows2000 und WindowsXP Kernel-Level Threads und das 1:1 Threading-Modell. Nach der Einführung von .NET war Microsoft stets bestrebt, Firmen dazu zu bewegen, neue Programme mit .NET zu entwickeln. Als Folge dessen haben die Win32-Threads an Bedeutung verloren, da .NET seine eigenen Threads mitliefert. Bei Applikationen, die nicht auf .NET basieren, werden Win32-Threads jedoch nach wie vor eingesetzt. Auch dürften .NET-Threads, da sie über eine virtuelle Maschine ausgeführt werden, in Bereichen, die Echtzeitanforderungen stellen, keine Alternative sein. Fraglich ist auch, ob die Windows-Version von .NET nicht sogar intern auf die Win32-Threads aufbaut. Das kann jedoch nur vermutet werden, da der Quelltext nicht offengelegt wird. Die Annahme ist zumindest berechtigt, da selbst die momentane Version der Pthreads-Win32 auf Win32-Threads aufbaut (siehe dazu die Datei create.c der Pthreads-Win32 Bibliothek, Version 2.7.0).

Auch Win32-Threads bieten eine große Funktionsvielfalt von Mutexen und Semaphoren bis Message-Queues und Task-Pools. Anstelle von Bedingungsvariablen, wie man sie von anderen Threading-Systemen kennt, arbeitet man bei Win32 mit sogenannten Event-Objekten. Ein Thread kann durch Warten auf einen solchen Event schlafen gelegt werden, und durch das Signalisieren des Events durch einen anderen Thread wieder geweckt werden. Events, die es in verschiedenen Ausprägungen gibt, gehen im Gegensatz zu Signalen jedoch nicht verloren. Mehr hierzu im Kapitel 4.5. Mutexe, Semaphore und Events können sogar auch zwischen Threads mehrerer Prozesse verwendet werden. Im Ganzen dürfte das Konzept der Win32-Threads für Programmierer, die bereits Erfahrung mit anderen Threading-Systeme haben, etwas verwirrend sein. Der Grund dafür ist, dass im Gegensatz zu den meisten anderen Threading-Systemen Mutexe, Semaphore und Events in Win32 allgemein als signalisierbare Objekte angesehen werden.

Das Signalisieren eines Events oder das Freigeben eines Mutex wird zwar noch mit auf den Typ des Objekts angepassten Funktionen realisiert, für das Warten auf eines dieser signalisierbaren Objekte werden jedoch gemeinsame Funktionen genutzt. In anderen Threading-Systemen wird zum Beispiel ein Mutex meist mit einer mutex_lock() Funktion gesperrt und mit mutex_unlock() entsperrt. Auf die Erfüllung einer Bedingungsvariable wird hingegen mit cond_wait() gewartet, wobei die Erfüllung der Bedingung mit cond_signal() signalisert werden kann. In Win32 entspricht mutex_unlock() der Funktion ReleaseMutex() und cond_signal() in etwa SetEvent(). Sowohl mutex_lock() als auch cond_wait() werden jedoch mit WaitForSingleObject() realisiert. Obwohl hierdurch eine eigentlich einheitliche Schnittstelle geschaffen wurde, wird diese durch das massive Angebot von Synchronisationsfunktionen wieder zunichte gemacht, wobei sich einige Funktionen auch noch überlappen. Ein Grund hierfür ist, dass viele Funktionen auch zur Interprozesskommunikation (IPC) genutzt werden können. Um Geschwindigkeitseinbußen zu vermeiden, wurden performantere Funktionen für Multithreading hinzugefügt. So gibt es neben Mutexen auch noch hierzu fast identische CriticalSection-Objekte, die zwar nicht von Threads mehrerer Prozessen geteilt werden können, dafür aber schneller gesperrt werden können. Ein anderer Grund ist es, dass es für viele Funktionen, darunter auch WaitFor ... Object(), eine Vielzahl von Varianten gibt, um für möglichst viele Spezialfälle gewappnet zu sein 4.4. Aus diesem Grund muss der Programmierer bei der Erzeugung von Threads genau aufpassen, dass er nicht die falsche Funktion verwendet . Es werden hierzu zwei Funktionen angeboten: CreateThread() und _beginthread()[2]. Hat man vor, die C-Standardbibliothek (LIBC) zu verwenden, z.B. um die Funktion printf() zu verwenden, sollte man auf jeden Fall _beginthread verwenden, da ansonsten Deadlocks auftreten können. Dies hat den Grund, dass nicht alle Funktionen der LIBC unter Windows Thread-sicher sind. Für Multithreading muss eine spezielle Variante der LIBC namens LIBCMT.lib (für LIBC Multihreaded) verwendet werden. Außerdem muss noch die Compileroption "/MT" gesetzt werden. Hierbei werden bei _beginthread() wichtige Initialisierungen der LIBCMT vorgenommen, was bei CreateThread() nicht der Fall ist (vgl. [msd03b] und [msd03a]). Kann auf die LIBC verzichtet werden, kann alternativ auch CreateThread() verwendet werden. Das gleiche gilt, wenn zwar die LIBC verwendet wird, jedoch jeder Aufruf einer LIBC-Funktion durch einen kritischen Abschnitt geschützt wird, z.B. mittels eines CriticalSection-Objekts (vgl. [msd03b]).

```
1 #include <windows.h>
2 #include <process.h>

4 void start(void* arg) {
5     /* Arbeite */
6     ...
7     _endthread();
8 }
```

[2]Verwendet man Microsofts objektorientierte MFC-Klassen, sollte man sich besser mit der Funktion AfxBeginThread() vertraut machen, da _beginthread() aus Gründen der Threadsicherheit inkompatibel zu MFC ist.

```
10 int main()
11 {
12     HANDLE hThread;
13     hThread = (HANDLE)_beginthread(start, 0, NULL);
14     WaitForSingleObject(hThread, INFINITE);
15     return 0;
16 }
```

Beispiel 3.4: Ein einfaches Beispiel mit Win32-Threads

Fibers: Die Threads der Win32-Threads

Dem aufmerksamen Leser wird vielleicht aufgefallen sein, dass die Überschrift dieses Abschnitts Windows-Threads und nicht Win32-Threads heißt. Das liegt daran, dass Windows (neben den .NET-Threads) noch eine andere Art von Threads bietet, die sogenannten Fibers (zu deutsch "Faser"). Fibers kann man sich als Thread im Thread vorstellen, d.h. Threads können (müssen aber nicht) noch einmal in mehrere Fibers unterteilt werden. Da es sich bei Fibers um eine pure User-Level Implementierung handelt, ist das Erzeugen neuer Fibers und das Umschalten zwischen Fibers sehr schnell möglich. Blockiert eine Fiber, betrifft dies jedoch den gesamten Thread und somit auch die anderen Fibers des Threads. Dies kommt dadurch Zustande, dass für alle Fibers eines Threads jeweils nur eine CPU verwendet wird. Zudem ist das Scheduling von Fibers non-preemptive, d.h. Fibers müssen selbst die Kontrolle über einen Thread mittels dem Befehl SwitchToFiber() an eine andere Fiber weitergeben. Die aktuelle Fiber übernimmt dann jeweils für die Zeit der Ausführung die Identität des Threads (vgl. [msd03d]). Interessanterweise bieten auch Fibers eine Art TLS (Thread Local Storage), hier FLS (Fiber Local Storage) genannt, mit der Fibers ihre eigenen spezifischen Daten in einer virtuellen "gemeinsamen Variablen" speichern können. Mehr dazu im Kapitel 4.10. Ein weiterer interessanter Punkt ist, dass Threads mit ConvertThreadToFiber() in Fibers konvertiert werden können. Nun kann man sich natürlich Fragen, wozu man Threads noch einmal in Fibers unterteilen sollte. Wie bereits in diesem Absatz und in Kapitel 2.2.4 erwähnt, können Fibers schnell gestartet und geschedult werden, da es sich um eine vom Kernel unabhängige User-Level Implementierung handelt. Die Unterstützung nur einer CPU fällt dabei nicht ins Gewicht, da man bereits durch eine geeignete Aufteilung der Threads für eine gute Lastenbalanzierung sorgen kann. Geeignet sind Fibers zum Beispiel für Webserver, bei denen zu jeder Anfrage eine neue Fiber erzeugt werden könnte. Es wäre auch denkbar Task-Pools damit zu implementieren und für jede Task eine neue Fiber zu erzeugen. Da die Erzeugung schnell ist, könnte man darauf verzichten Fibers, die evtl. gar nicht zum Einsatz kommen, "auf Halde" zu erzeugen. Ein solcher Task-Pool könnte flexibel auf eine schwankende Anzahl eingehender Tasks reagieren ohne unnötig Speicherplatz zu verbrauchen. User-Level Threads (hier eigentlich Fibers) besitzen keine zusätzlichen Datenstrukturen im Kernel. Das verringert den Speicherbedarf. Ein nicht zu unterschätzender Vorteil, wenn die Anzahl

der Fibers in die tausende geht. Hieran sieht man, dass Fibers bzw. User-Level Threads generell durchaus ihre Berechtigung haben, selbst wenn das Betriebssystem Kernel-Level Threads anbietet. Die Betrachtung der Fibers soll hiermit abgeschlossen werden. Für weitere Informationen soll auf [msd03d] verwiesen werden.

Windows bietet viele weitere Besonderheiten im Bereich des Multithreadings, die hier nicht alle erwähnt werden können. Z.B. kann man die Win32-Threads auch in Microsofts COM+ Objekten verwenden. Hier werden jedoch unterschiedliche Ebenen der Parallelität unterschieden, die als Apartment-Modelle bezeichnet werden. Wer mehr darüber erfahren will sei auf [msd03c] verwiesen.

3.5 Perl-Threads

Ein ebenfalls sehr interessantes Threading-System stellen die in Perl integrierten Threads dar. Interessant vor allem deshalb, weil es sich bei Perl um eine Skriptsprache handelt. Ein Bereich, in dem man Multi-Threading eigentlich nicht erwartet, aber wo es durchaus Sinn macht. Auf der einen Seite werden Skripte oft für Aufgaben eingesetzt, in denen keine oder nur wenig Benutzerinteraktivität nötig ist, wodurch viele Skripte gut parallelisierbar sein dürften. Auf der anderen Seite werden viele Skripte genau aus diesem Grund über Nacht im Hintergrund als Batch-Job gestartet, wobei die Ausführungszeit oft nicht zu Buche schlägt. Da hier mehrere Skripte gleichzeitig aktiv sind, kann ein Multiprozessorsystem bereits durch das alleinige Vorhandensein mehrerer Prozesse meist gut ausgelastet werden. Allerdings dringt Skripting in immer mehr Bereiche vor, die früher lediglich “richtigen“ Programmiersprachen vorbehalten waren, wie z.B. auch die Programmierung von Benutzeroberflächen[3]. Das liegt daran, dass die Entwicklungszeiten in den Bereichen, auf die Skripts zugeschnitten sind, einfach sehr viel kürzer sind, wodurch die Produktivität gesteigert werden kann. Diese Applikationen sollten aber auch bei Skriptimplementierung schnell laufen und nicht bei jeder Aktion blockieren. Somit ist Multithreading nicht länger die alleinige Domäne “ausgewachsener“ Programmiersprachen wie z.B. C. Neben Perl besitzen auch andere bekannte Skriptsprachen wie Python und Ruby Unterstützung für Multi-Threading. Allerdings ist die Perl-Implementierung in Hinsicht auf diese Arbeit die Interessanteste. Weil eine Betrachtung aller Skriptsprachen im Rahmen dieser Arbeit leider nicht möglich ist, wird lediglich Perl betrachtet. Dabei unterscheiden sich die Implementierungen unter den Skriptsprachen jedoch mindestestens genauso wie bei anderen nicht skript-basierten Threading-Systemen.

Die erste Thread-Implementierung wurde 1998 in Perl mit Version 5.005 eingeführt und auch 5005threads genannt (vgl. [per]. Zu finden sind die Threads im Paket “Thread“. Es wird jedoch empfohlen, diese Thread-Implementierung nicht mehr zu verwenden. Der Grund dafür ist, dass die gemeinsame Verwendung von 5005threads und regulären Ausdrücken zu Race-Conditions führen kann (vgl. [per]. Anstattdessen sollte man die im Jahr 2000 mit Perl 5.6 neu

[3] In diesem Gebiet ist vor allemn auch Python sehr beliebt.

eingeführten und im Jahr 2002 mit Perl 5.8 öffentlich gemachten ithreads (interpreter threads) verwenden, die sich im Paket “threads“ befinden. Die Entscheidung für eine der beiden Implementierungen muss jedoch bereits bei der Installation von Perl geschehen. Standardmäßig ist keine der beiden Implementierungen in Perl integriert. Die Unterstützung hierfür muss explizit beim Installieren, genauer gesagt beim Kompilieren der Perl-Umgebung ausgewählt werden. Neben einer stabileren Implementierung unterscheiden sich die ithreads in einem ganz entscheidenden Punkt: 5005threads haben wie in anderen Threading-Systemen auch einen gemeinsamen Adressraum, können also gegenseitig auf ihre Daten zugreifen. Zugriffe auf gemeinsame Daten müssen synchronisiert werden. Die ithreads haben hingegen standardmäßig einen *getrennten* Adressraum. Jeder Thread hat dadurch seine eigene Kopie einer Variable. Damit Variablen gemeinsam verwendet werden können, muss dies bei ihrer Deklaration mit Hilfe des “shared“-Attributs angegeben werden. In anderen Threading-Systemen muss zusätzlicher Aufwand betrieben werden um thread-eigene Instanzen einer Variable zu erzeugen, z.B. mit TLS (Thread Local Storage) 4.10. In Perl ist im Gegensatz dazu ein Mehraufwand nötig, um gemeinsamen Variablen zu erzeugen. In Analogie zu TLS könnte man dies eigentlich TGS (Thread Global Storage) nennen, obwohl ein solcher Begriff nicht existiert. Mit diesem Speichermodell widerspricht Perl der Definition eines Threads, war doch der gemeinsame Adressraum eines der Kennzeichen, die den Thread vom Prozess unterscheiden. Betrachtet man die Perl-Threads genauer, stellt man auch fest, dass sie auch von der Thread-Architektur nicht vergleichbar mit anderen Threading-Systemen sind. Erzeugt man in Perl einen neuen ithread, so wird der gesamte Interpreter kopiert. Es läuft also immer nur ein Thread auf einem Interpreter, daher auch der Name “interpreter threads“. Dadurch wird auch klar, wieso der Adressraum standardmäßig getrennt ist (vgl. [per05]).

Auch sonst bietet Perl einiges im Bereich des Multi-Threading. Neben Locks und Bedingungsvariablen werden sogar Message-Queues angeboten.

```
1 #!/usr/bin/perl
2 use threads;

4 sub start {
5     # Arbeite ...
6 }

8 $thread = threads->new(\&start);
9 $thread->join();
```

Beispiel 3.5: Ein einfaches Beispiel mit Perl-Threads

3.6 Weitere Threading-Systeme

An dieser Stelle sollen ein paar ausgesuchte Threading-Systeme genannt werden, die im Rahmen dieser Arbeit nicht ausführlicher betrachtet werden konnten.

- Boost-Threads: Sie sind Teil der Boost C++-Bibliotheken. Diese gelten als heißer Kandidat für die Integration in einen zukünftigen C++-Standard. Es existieren bereits Implementierungen für verschiedene Plattformen, darunter Windows, Linux, Mac OS-X und Solaris. Zur Synchronisation stehen u.a. (rekursive) Mutexe, Bedingungsvariablen und Barrieren bereit. Weitere Informationen unter [boo06].

- Solaris-Threads: Solaris-Threads bieten das Konzept der LightWeight Processes (LWP), einem sehr interessantem M:N Threading-Modell. Weitere Informationen unter [LB98].

- Nano-Threads: Eine User-Level Thread-Bibliothek. Programmcode wird durch einen automatisch parallelisierenden Compiler in einen hierarchischen Abhängigkeitsgraphen, wodurch besonders effiziente Parallelisierung erreicht werden soll. Weitere Informationen unter [nan06].

- Ada: Eine Programmiersprache, die vor allem (aber nicht nur) von staatlichen Behörden, darunter auch dem Militär verwendet wird. Ada ist zwar kein Threading-System im Sinne dieser Arbeit. Im Unterschied zu anderen Sprachen werden in Ada keine Prozesse, sondern Tasks unterschieden, die durch eine Compileroption auch als Threads ausgeführt werden können. Alleine das Rendez-vous Prinzip Adas zur Synchronisierung von Tasks ist es Wert, sich mit dieser Sprache zu beschäftigen. Weitere Informationen unter [ada06].

- Python: Eine weitere sehr bekannte Skriptsprache mit Multithreading-Unterstützung. Weitere Informationen unter [pyt06].

Kapitel 4

Eigenschaften von Threading Systemen

In diesem Kapitel sollen die Eigenschaften von Threading-Systemen betrachtet werden. Mit Eigenschaften sind in diesem Zusammenhang vor allem Konzepte und Features gemeint, die von den Threading-Systemen implementiert werden. Einige dieser Eigenschaften werden von allen Threading-Systemen unterstützt, andere hingegen nur von einigen wenigen. In den einzelnen Abschnitten dieses Kapitels wird jeweils eine Eigenschaft betrachtet und anhand eines oder mehrerer Threading-Systeme aus Kapitel 3 verdeutlicht. Gleichen sich mehrere Threading-Systeme in einer Eigenschaft, soll in diesen Fällen nur ein Repräsentant betrachtet werden.

4.1 Plattformunabhängigkeit

Mittlerweile arbeiten viele Firmen auf Workstations und Servern unterschiedlichster Architekturen und unter verschiedenen Betriebssystemen. Darunter Workstations mit x86-CPU und Windows oder Linux, Apple Rechner mit PowerPC-CPU und OS-X, Server mit UltraSparc-CPU und Solaris. Vor allem im Server-Bereich gibt es eine Vielzahl inkompatibler Architekturen und Betriebssysteme. Hat man ein Programm für eine Plattform programmiert, muss man sie meist komplett umschreiben, um sie auch auf einem anderen System laufen zu lassen. Da das Monopol von Microsoft langsam zu bröckeln beginnt, kann es sich ein Entwickler heute kaum noch leisten, Programme lediglich für eine Architektur zu entwerfen. Da die Akzeptanz für Betriebssysteme außerhalb der Microsoft Welt immer größer wird, sollten Programme auf einer möglichst großen Anzahl mehr oder minder inkompatibler Systeme laufen. Dies betrifft sowohl den Server- (Solaris, AIX, Windows Server, usw.) als auch den Workstationbereich (WindowsXP, Linux und Mac OS-X).

Aus diesem Grund sollte man darauf achten, Programme möglichst plattformunabhängig zu gestalten, um sie für eine größtmögliche Anzahl von Zielplattformen bereitstellen zu können. Plattformunabhängigkeit ist sehr stark mit dem Begriff der Portabilität verknüpft. Die Portabilität ist umso größer, je geringer der Zeitaufwand zur Portierung eines Programmes auf andere Plattformen ist (vgl. [Por06]). Je plattformunabhängiger ein Programm, desto besser ist dem-

nach die Portabilität. Dies gilt natürlich auch und sogar in besonderem Maße für Multithreading. Viele Threading-Systeme beruhen auf der bereits vorhandenen Threading-Funktionalität der darunterliegenden Architektur oder des Betriebssystems.

Doch was bedeutet Plattformunabhängigkeit im Zusammenhang mit Threading-Systemen? Im Prinzip kann man die Kriterien der Plattformunabhängigkeit von Programmen (vgl. [Pla06]) weitestgehend auch auf die des Multithreadings übertragen. Ein Threading-System ist demnach plattformunabhängig, wenn die parallelen Anteile von Programmen, die auf diesem Threading-System beruhen, auf mehreren Plattformen lauffähig sind, d.h. unabhängig von darunterliegender Hardware, Architektur oder Betriebssystem.

Der Grad der Plattformunabhängigkeit eines jeden Thread-Systems kann jeweils in eine von drei Kategorien eingeteilt werden:

Interoperabel Ein auf dem Threading-System aufbauendes Programm lässt sich ohne Anpassung auf mehreren unterschiedlichen Plattformen ausführen.

Quelltextkompatibel Der Quelltext ist kompatibel zu mehreren Plattformen. Das Programm muss für weitere Zielplattformen lediglich neu kompiliert werden, ohne dass eine Änderung des Quellcodes erforderlich ist.

Plattformgebunden Das Programm ist nur auf einer bestimmten Plattform lauffähig. Soll das Programm auch auf einer anderen Plattform ausgeführt werden, muss der Quelltext aufwändig unter Verwendung eines anderen Threading-Systems portiert werden.

4.1.1 Interoperable Threading-Systeme

Es gibt zwei Arten von interoperablen Thread-Systemen: Zwischencode-basierte und Skript-basierte. Beide Systeme haben gemein, dass der Portierungsaufwand nicht von jedem Programmierer selbst betrieben werden muss, sondern von den Entwicklern des Threading-Systems. Deren Aufgabe ist es, eine auf die Plattform zugeschnittene Laufzeitumgebung (Interpreter und dazugehörige Laufzeitbibliothek) zu entwickeln. Auf jedem System, für das eine Implementierung des Interpreters existiert, kann ein Programm, das auf dem Threading-System basiert, ohne Anpassung seitens des Programmierers gestart werden. Somit ergibt sich eine sehr starke Plattformunabhängigkeit. Ein Nachteil hierbei ist jedoch, dass zum Ausführen eines Programmes der dazugehörige Interpreter für eine Plattform zum einen existieren und zum anderen auch installiert sein muss. Ein dritter Schwachpunkt ist der größere Overhead, den die Interpreter mit sich bringen, wodurch sich die Ausführungsgeschwindigkeit im Gegensatz zu nativen Programmen stark verringert. Echtzeitanwendungen z.B. sind mit diesen Programmiersystemen im Allgemeinen nicht realisierbar. Aufgrund der immer schneller werdenden CPUs ist dieser Kritikpunkt in normalen Anwendungen in den meisten Fällen zu vernachlässigen.

Zwischencode-basierte Threading-Systeme

Zur Gattung der interoperablen Zwischencode-basierten Threading-Systeme zählt man solche, bei denen der Quelltext des Programmes in einen Zwischencode übersetzt wird. Dieser wird dann auf den jeweiligen Plattformen von einer Virtuellen Maschine (engl.: Virtual Machine) ausgeführt. Die Virtuelle Maschine kann man sich hierbei als eine virtuelle CPU mit eigener, ebenfalls virtueller, Systemarchitektur vorstellen, deren Maschinencode dem Zwischencode entspricht. Unter virtuell ist hierbei zu verstehen, dass alles durch Software emuliert wird. Die Architektur und die CPU-Befehle sind dadurch auf jedem System gleich. Ein Threading-System muss demnach die Thread-Erzeugung und Steuerung nur noch für ein Zielsystem implementieren, nämlich das der Virtuellen Maschine. Jedes Programm ist dann in der Lage, auf einem beliebigen System ohne jegliche Anpassung über die Virtuelle Maschine gestartet zu werden.

Multithreading-Unterstützung ist für die gebräuchlichsten Virtuellen Maschinen bereits fest integriert. Standardbibliotheken sorgen dafür, dass in jeder verwendeten Programmiersprache, deren Quelltext in Zwischencode konvertiert werden, die einheitliche Funktionen für das Multithreading zur Verfügung stehen. Die bekanntesten Vertreter dieser Gattung sind Java-Threads und .NET-Threads.

.NET-Applikationen und somit .NET-Threads sind auf alle Systeme portierbar, die ein .NET-Framework installiert haben. Allerdings trifft dies momentan lediglich auf Windows-Rechner zu, da das .NET-Framework von Microsoft nur für diese Systeme angeboten wird. Die Portabilität ist somit in der Praxis sehr beschränkt realisierbar.

Durch das .NET kompatible OpenSource-Projekt Mono wurde jedoch die Möglichkeit geschaffen, .NET-Programme z.B. auch unter Unix oder Mac OS X auszuführen.

Java-Implementierungen hingegen gibt es für fast alle modernen Betriebssysteme. Aber auch hier ist die Plattformunabhängigkeit nicht uneingeschränkt, wenn man die auf einem System eingesetzte Version des Java Runtime Environment (JRE) als Teil der Plattform ansieht. Es ist z.B. nicht möglich, Java-Programme, die die erweiterten Threading-Fähigkeiten von Java 5.0 nutzen, mit einer älteren Version des JRE zu starten.

Skript-basierte Threading-Systeme

Unter diese Überschrift fallen Threading-Systeme, die entweder direkt in eine Skriptsprache integriert sind oder die zumindest als Bibliothek für eine Skriptsprache zur Verfügung stehen. Allerdings müssen Skript-basierte Threading-Systeme nicht zwingend interoperabel sein. Trifft dies schon nicht für die Skript-Sprache an sich zu, so kann dies natürlich auch nicht für das jeweilige Multithreading gelten. Interpretierende Skriptsprachen gehören jedoch in diese Kategorie, solange die Sprache alle Betriebssystemaufrufe in eigenen Bibliotheken schachtelt. Somit haben auch die Befehle zur Thread-Verwaltung keinen Bezug auf das Betriebssystem mehr. Es genügt dann, das Skript auf einem anderen Betriebssystem von dem hierfür angepassten Interpreter ausführen zu lassen. Allerdings muss natürlich ein angepasster Interpreter existieren.

Prominente Beispiele für interpretierende Skriptsprachen sind Ruby und Perl[1].

4.1.2 Quelltextkompatible Threading-Systeme

Ein Threading-System ist dann quelltextkompatibel, wenn eine Portierung des parallelen Anteils eines Programmes auf ein neues System dadurch möglich ist, den Quelltext auf diesem System ohne Anpassung einfach nur neu zu kompilieren. Threading-Systeme, die diese Eigenschaften besitzen, sind entweder bereits in eine portable Programmiersprache integriert oder als Bibliotheken realisiert, die dann aber für mehrere Systemen vorhanden sein müssen. Die Definition fester Standards für Threading-Systeme bieten eine weitere Möglichkeit, um Quelltextkompatibilität zu erhalten. Auf diese Weise können unterschiedliche, aber zum Standard konforme, Bibliotheken auf den jeweiligen Plattformen zum Einsatz kommen.

Im folgenden soll das bekannteste Beispiel eines quelltextkompatiblen Systems betrachtet werden, die POSIX-Threads. Bei Pthreads handelt es sich nicht um eine konkrete Implementierung, sondern einen Standard (Siehe Kapitel 3.1). Theoretisch sollte ein Programm, das Pthreads verwendet, auf jedes andere Betriebssystem mit Pthreads-Implementierung portierbar sein. In der Praxis funktioniert dies sehr gut. Lediglich die Teile des Programms, die nicht die Pthreads-Bibliothek oder andere standardisierte Funktionen verwenden, müssen angepasst werden. Das Problem hierbei ist jedoch, dass nicht alle Implementierungen der Pthreads-Bibliotheken zu hundert Prozent zu dem POSIX-Standard kompatibel sind. Zum einen kann dies dazu führen, dass sich Programme erst gar nicht kompilieren lassen. Das ist dann der Fall, wenn nicht standardisierte, implementierungsspezifische Funktionen[2] verwendet wurden. Zum anderen kann es sein, dass eine Funktion zur Verfügung steht, sich jedoch anders verhält, als im Pthread-Standard beschrieben wurde. Hinzu kommt, dass es im Laufe der Zeit mehrere Erweiterungen des Pthread-Standards gab. Man sollte aus diesem Grund nur die Funktionen eines Standards verwenden, für die es auf allen Zielplattformen auch eine Pthreads-Implementierung gibt. Selbst wenn sich eine Implementierung strikt an einen Pthread-Standard hält, so muss nicht jede Funktionalität implementiert sein. Funktionen die dies betrifft geben einen Fehlercode (ENOTSUP) zurück, der die mangelnde Funktionalität anzeigt. Dies betrifft vor allem Betriebssystem-spezifische Funktionalität wie z.B. die Erzeugung von Echtzeit-Threads (siehe Kapitel 4.11).

Auch wenn die Threading-Systeme dieser Kategorie eine passable Plattformunabhängigkeit erreichen, ist diese nicht annährend so gut wie bei interoperablen Systemen (siehe Beispiel Pthreads). Der große Vorteil von quelltextkompatiblen Threading-Systemen liegt darin, dass sie durch den nativen Programmcode eine sehr hohe Ausführungsgeschwindigkeit im Vergleich zu den interoperablen Systemen besitzen. Sie vereinen die Vorteile aber auch die Nachteile der

[1] In Perl werden die Skripte zwar vor der Ausführung vorkompilieren, man spricht aber trotzdem meist von einer interpretierenden Skriptsprache.

[2] Zusätzlich zum Standard angebotene Funktionen werden im Funktionsnamen durch "_np" (non portable) gekennzeichnet.

Threading-Systeme der anderen beiden Kategorien.

4.1.3 Plattformgebundene Threading-Systeme

Threading-Systeme dieser Kategorie sind nur auf eine bestimmte Architektur spezialisiert. Eine Portierung eines Programmes auf eine andere Plattform unter Verwendung des gleichen Threading-Systems ist nicht möglich. Gründe hierfür können sein, dass eine Implementierung des Threading-Systems nur für eine Plattform existiert. Zum einen kann es sein, dass eine Portierung des Systems nicht möglich, da die Routinen zu sehr auf spezifischer Betriebssystem-Funktionalität beruhen. Zum anderen kann es sein, dass die Entwickler eines Threading-Systems kein Interesse an dessen Portierung haben. So wäre eine Portierung von Win32-Threads auf die Betriebssysteme der Konkurrenz wie Linux oder Mac OS-X wohl kaum im Interesse von Microsoft. Ein möglicher dritter Fall hätte seine triviale Ursache darin, dass ein Threading-System einfach zu unbedeutend ist, um auf andere Systeme portiert zu werden.

4.1.4 Zusammenfassung

Eine hundert-prozentige Plattformunabhängigkeit ist nur theoretisch möglich. Vorteile in der Plattformunabhängigkeit müssen mit Nachteilen in der Ausführungsgeschwindigkeit teuer erkauft werden. Threading-Systeme durch Spezifikation eines Standards Plattformunabhängig zu gestalten, ist in der Theorie zwar eine gute Idee, scheitert jedoch daran, dass die Standards oft nicht vollständig implementiert oder unterschiedlich interpretiert werden. Hier hilft zwar eine Zertifizierung der Standardkonformität durch eine Prüfstelle; Implementierungen die ein solches Zertifikat vorweisen können, sind jedoch meist mit enormen Kosten verbunden und daher nur für den kommerziellen Gebrauch einsetzbar.

4.2 Implementierungsort

Es gibt viele Möglichkeiten, Multithreading in ein Programm einzubinden. In neuere Programmiersprachen wie Java ist eine Multithreading-Unterstützung bereits fest integriert. Ältere Sprachen jedoch, wie C oder C++ bieten nicht die Möglichkeit, Threads zu erzeugen oder zu verwalten. Für Sprachen ohne eigene Threading-Unterstützung benötigt man daher Bibliotheken, die fehlende Funktionalität bieten. Es macht also Sinn, zu unterscheiden, wo die eigentliche Threading-Funktionalität eines Threading-Systems implementiert wird. Dies soll im folgenden als Implementierungsort verstanden werden. Folgende Implementierungsorte sind denkbar:

1. Einbindung in die Programmiersprache
2. Bibliotheken

3. Kernel

Eine Implementierung in eine Programmiersprache geht normalerweise immer mit einer dazugehörigen Klassenbibliothek einher. So wurde zum Beispiel in Java das Schlüsselwort “synchronized“ zur Synchronisierung von Methodenaufrufen durch Threads direkt in die Sprache integriert. Die Erzeugung der Threads und weitere Synchronisationsmechanismen wie Bedingungsvariablen werden jedoch als Bibliotheksfunktionen realisiert. Eine Beispiel für eine Bibliotheksimplementierung sind Pthreads-Bibliotheken. Pthreads ist ein Standard, der Funktionen und ihr Verhalten vorgibt, nicht aber ihre Implementierungen. Aus diesem Grund können Threads auf einer Rechnerplattform von verschieden Pthread-konformen Bibliotheken auf völlig unterschiedliche Weise realisiert werden. So können die Threads entweder auf Kernel-Level-Basis oder ganz in der Bibliothek implementiert werden. Es gibt aber auch einen dritten Weg, an Multithreading-Funktionalität zu gelangen. In neueren Linux-Versionen ist es z.B. durchaus möglich (wenn auch nicht üblich) durch Systemaufrufe aus einem Programm im User-Mode heraus Kernel-Threads zu erzeugen. Diese könnten dann vom Programmierer verwendet werden, ohne eine Bibliothek oder Programmiersprachen-Konstrukte zu verwenden. Aus Gründen der Portabilität sollte dies jedoch nur Kernelprogrammierern vorbehalten bleiben.

An dieser Stelle noch eine kurze Bemerkung zu Linux. In dieser Arbeit wurde bereits davon gesprochen, dass Linux Kernel-Threads unterstütze. Ganz korrekt ist diese Formulierung jedoch nicht. Linux kennt im Grunde keine Threads. Sowohl Threads als auch Prozesse werden über eine clone() Funktion erzeugt. Diese Funktion kann durch Angabe eines Parameters dazu bewogen werden, einen Prozess mit gemeinsamen Adressraum zu erzeugen, wodurch eigentlich ein Thread erzeugt wird. In Bezug auf Linux-Kernel wird daher nicht von Prozessen oder Threads gesprochen, sondern allgemein von Tasks. Die Bezeichnung von Tasks mit gemeinsamen Adressraum als Kernel-Level Thread scheint daher berechtigt und soll auch im weiteren Verlauf der Arbeit beibehalten werden.

4.3 Erzeugung von Threads

Lässt man ein serielles Programm laufen, so wird dieses Programm von einem Thread, dem Haupt-Thread, abgearbeitet. Dieser Thread wird implizit vom Betriebssystem erzeugt. Um Parallelität innerhalb eines Prozesses zu erlangen, reicht ein Thread jedoch nicht. Man benötigt mindestens einen weiteren Thread. Die Unterschiede in der Thread-Erzeugung, die es zwischen den einzelnen Threading-System gibt, unterscheiden sich mehr oder minder stark. Threading-Systeme lassen sich bezüglich der angebotenen Thread-Erzeugung in Gruppen einteilen.

In nicht objektorientierten Threading-Systemen wie Win32- und POSIX-Threads wird zur Erzeugung neuer Threads meist eine Methode in der Form thread_create (start_func , param) angeboten. Hierbei soll der Parameter start_func für einen Funktionszeiger auf eine Funktion stehen, die der neu erzeugte Thread ausführen soll. Der Parameter param steht für einen Parameterwert, mit dem die Startfunktion aufgerufen werden soll. Hierüber kann man z.B. den

Threads bestimmte Ausprägungen geben, die zu einem differenzierten Verhalten der einzelnen Threads führt. Die Funktion thread_create () gibt nach erfolgreicher Erzeugung des Threads einen Thread-Identifier oder ein Handle auf einen Thread zurück, der bzw. das den Thread eindeutig identifiziert.

Die oben angegebenen Parameter einer Funktion zur Thread-Erzeugung sind lediglich eine minimale Variante. So müssen bei den Win32-Threads in der Funktion CreateThread() zusätzlich die zu reservierende Stackgröße und Attribute, die das Verhalten der Threads ändern, übergeben werden. Auch POSIX-Threads bieten viele Attribute an, die beim Aufruf der Funktion pthread_create () übergeben werden können. Hiermit kann unter anderem der Scheduling-Priorität des Thread verändert werden.

Auf objektorientierten Sprachen aufbauende Threading-Systeme gehen einen anderen Weg. Hier wird die Threading-Funktionalität meist durch eine Thread-Klasse oder ein Interface angeboten. Will man einen einen Thread erzeugen, muss man eine Instanz einer von "Thread" geerbten Klasse erzeugen. Man kann auch eine Klasse das Interface Runnable implementieren lassen, das daraufhin instanziiert, aber nicht direkt gestartet werden kann. Hierzu muss zusätzlich ein Objekt der Klasse Thread erzeugt werden.

Ein großer Unterschied in diesem Bereich ist das eigentliche Starten des Threads. In einigen Threading-Systemen wie Pthreads starten die Threads sofort nach der Erzeugung, in .NET und Java hingegen müssen sie explizit mit einer speziellen Funktion (in Java Thread. start () gestartet werden.

Im Rahmen der Threaderzeugung in objekt-orientierten Sprachen soll darauf aufmerksam gemacht werden, dass ein Thread und das Thread-Objekt nicht unbedingt das gleiche sind. Man sollte eher davon sprechen, dass ein Thread bei der Erzeugung an ein Objekt gebunden wird. Ruft ein Thread eines Thread-Objekts in Java Methoden eines anderen Thread-Objekts auf, so arbeitet der Thread nämlich auch den Programmcode des anderen Objekts ab.

Werden Threads erzeugt, sollte man darauf achten, dass sie nicht sofort nach der Erzeugung unfreiwillig gelöscht werden. In Pthreads sind die erzeugten Threads nur so lange aktiv, wie der Hauptthread läuft. Dieser sollte deshalb bevor er sich beendet auf die Beendigung der weiteren Threads des Programmes warten, um diese nicht mitten in ihrer Ausführung zu unterbrechen. Hierzu kann die Funktion pthread_join () genutzt werden, die den aufrufenden Thread so lange schlafen legt, bis der Thread, auf den gewartet werden soll, beendet ist. So gut wie alle Threading-Systeme bieten eine join () Methode, darunter auch Java und Perl. In Java hingegen kann der Hauptthread abbrechen, ohne dass die anderen Threads dadurch beendet würden. Die JVM beendet sich erst dann, wenn das Programm nur noch von sogenannten Daemons (Hintergrundthreads) ausgeführt werden. Da neue Threads jedoch standardmäßig als Vordergrundthreads erzeugt werden, bricht das Programm normalerweise erst nach Beendigung des letzten Threads ab [Oec01, S. 153].

4.4 Semaphore, Mutexe, Locks

Um eine Synchronisierung der Threads in kritischen Abschnitten zu erreichen, muss ein Threading-System selbst Mechanismen hierfür anbieten. Ohne Unterstützung des Threading-Systems bzw. des Betriebssystems ist eine Synchronisation normalerweise nicht möglich. Ein solcher Mechanismus wird ***Semaphor*** genannt. Semaphoren wurden erstmals um 1965 von E.W. Dijkstra verwendet. Wie man gleich bemerken wird, eignen sich Semaphore besonders gut, um die Anzahl von Threads in kritischen Abschnitt zu begrenzen oder eine begrenzte Anzahl von Ressourcen zu verwalten. Semaphore verhalten sich wie Zählvariablen, die mittels spezieller *atomischer* Befehle hoch- (V-Operation) bzw. heruntergezählt (P-Operation) werden. Zu Beginn wird der Semaphor auf einen vom Benutzer gewählten Wert gesetzt. Solange dieser Wert >0 ist, kann ein Thread den Semaphor ungestört mit der P-Operation herunterzählen. Sollte der Semaphor den Wert 0 besitzen, so wird der Thread schlafengelegt, bis er durch das Hochzählen (Freigeben) des Semaphor durch einen anderen Thread wieder aufgeweckt wird. Wichtig ist, dass das Herunter-/Herauf-zählen sowie das evtl. nötige Schlafenlegen atomisch ausgeführt wird. D.h. es findet kein Kontextwechsel zwischen den einzelnen Operationen statt. Die Implementierungen in den Threading-Systemen entsprechen zwar alle im Prinzip der oben angebenen Beschreibung von Semaphoren, weisen aber trotzdem mehr oder minder große nicht zu vernachlässigende Unterschiede auf.

Zuerst sollte man sich natürlich fragen, welche Threading-Systeme denn überhaupt Semaphoren anbieten. Dies sind die meisten, darunter BeOS, Win32 und Perl. Auch der Pthreads-Standard sieht Semaphore vor, jedoch erst mit Einführung von POSIX 1003.1b. Die meisten Pthreads-Implementierung unterstützen Semaphoren bereits. Man sollte jedoch nicht versuchen, die SystemV Semaphore mit Pthreads zu verwenden, denn diese sind *nicht* threadsicher (siehe 4.9. Sowohl bei Java als auch .NET waren in den Ursprungsversionen keine Semaphore vorgesehen. Der Grund hierfür dürfte die Java-Philsophie gewesen sein, die Standardbibliothek so klein wie möglich zu halten. Semaphore lassen sich jedoch sehr leicht mit Monitoren nachbilden. Mittlerweile wurden die Java- als auch .NET-Klassenbibliotheken mit Erscheinen von Java 5.0 bzw. .NET 2.0 um Semaphore-Klassen erweitert. Ein Beispiel für ein Threading-System, in dem hingegen keine Semaphore zur Verfügung stehen, ist das bereits veraltete OS/2.

Fundamental unterscheiden sich die Betrachtungsweisen bezüglich der Semaphore durch die Threading-Systeme. Verwendet man Semaphore z.B. zur Verwaltung einer beschränkten Anzahl von Ressourcen, so will man meist nicht nur wissen, ob denn noch eine Ressource zur Verfügung steht, sondern wie viele dies noch genau sind. Ein direkter Zugriff auf den Inhalt des Zählerstands des Semaphors ist zwar in keinem Thread-System ohne unerlaubte Tricks möglich, dafür kann bei Pthreads und BeOS der Wert mittels eines Aufrufs in der Form sem_getvalue() abgefragt werden. Bei den Win32- und Java-Threads handelt es sich bei den Semaphoren um vollständig gekapselte Datentypen. Ein Zugriff auf den Zählerwert ist nicht möglich. D.h. das man eine weitere Variable einführen muss, die man selbständig auf die aktuelle Zahl der Ressourcen setzen muss. Neben dem Nachteil der Redundanz sollten man auch beachten, dass diese

Variable ebenfalls von mehreren Threads geteilt wird und Zugriffe geschützt werden müssen. Obwohl die Bereitstellung einer sem_getvalue() sehr konfortabel ist, verleitet sie Programmierer jedoch dazu, den Wert auzulesen, ohne sie durch einen kritischen Abschnitt zu schützen. Dies kann leicht zu Race-Conditions führen, da der abgefragte Wert ein von mehreren Threads verwendeter Wert ist. Sollte der ausgelesene Wert z.B. dafür verwendet werden, zu entscheiden, ob ein Thread einen kritischen Abschnitt betreten soll oder nicht, muss das Auslesen und die davon abhängige Entscheidung in einem kritischen Abschnitt geschehen, da sonst der Wert bei der Entscheidung bereits geändert worden sein könnte.

In vielen Fällen benötigt man die Eigenschaft einer Zählvariablen gar nicht. Darf ein kritischer Abschnitt, wie in den meisten Fällen. nur von einem Thread betreten werden, so genügt die Information, ob bereits ein Thread den kritischen Bereich betreten hat oder nicht. Man würde den Semaphore deshalb bei der Initialisierung auf den Wert 1 setzen. Da diese Art von Semaphore nur zwei Werte (1 und 0) annehmen kann, nennt man sie ***binäre Semaphore*** oder ***Mutex*** (Mutual Exclusion, engl. für "wechselseitiger Ausschluss"). Da hiermit der Zugriff auf kritische Abschnitte geregelt wird, werden die Mutex-Primitive in einigen Sprachen wie .NET auch selbst als Critical-Sections bezeichnet. Fast alle Threading-Systeme bieten für die Verwaltung von Mutexes oder Critical-Sections zusätzliche Funktionen an. Selbst in den Versionen von Java und .NET, die keine Semaphoren implementieren, werden Mutexe oder Critical-Sections angeboten, da die Verwendung von Mutexen sehr viel häufiger vorkommt und sich Semaphore hierdurch einfach nachbilden lassen. Mutexe oder Critical-Section werden auch oft ***Lock*** (Sperre) genannt.

Perl zeigt, dass man durchaus auch ohne mutex_unlock() auskommen kann. Hier können beliebige Variablen (die jedoch wegen des getrennten Adressraumes als "shared" deklariert werden müssen) durch eine lock()-Funktion gesperrt werden. Ein Entsperren geschieht automatisch, sobald der Block, in dem die Variable gesperrt wurde, beendet ist.

```
1 my $lock_var : shared;    # gemeinsame Variable deklarieren
2
3 {
4     lock($lock_var);  # Sperre anfordern
5     # Arbeite mit der Sperre
6 } # Sperre implizit freigeben
```

Der Aspekt des Schlafenlegens sollte etwas genauer betrachtet werden. Es könnte durchaus sein, dass das "Schlafenlegen" in einem Threading-System so realisiert wurde, dass ein Thread in einer Schleife immer wieder prüft (auch Polling genannt), ob das Lock mittlerweile freigegeben wurde. Dieses aktive Warten (***busy-waiting*** gennant) verbraucht CPU-Leistung. Daher sollte solch eine Lösung vermieden werden. Sinnvoller ist es, den Thread so schlafen zu legen, dass sie keine CPU-Zyklen mehr verbrauchen und erst nach dem Aufwecken wieder aktiv werden. Der Thread wird hierbei blockiert. Threading-Systeme, die auf Kernel-Level-Threads aufbauen, haben diese Möglichkeit. Bei User-Level-Bibliotheken, die nicht auf Kernel-Level-Threads aufbauen, ist dies nicht möglich. Hier würde nicht nur der Thread, sondern der gesamte

Prozess blockiert. Das Lock könnte nie wieder freigegeben werden, da der Thread, der das Lock besitzt ebenfalls schläft. Der Prozess hängt im Dauerschlaf fest. User-Level-Bibliotheken können das Blockieren eines Threads jedoch auch dadurch erreichen, dass der Thread einfach so lange nicht mehr vom Scheduler ausgewählt wird, bis das Lock wieder freigegeben wurde. Dies ist möglich, da der Scheduler ein Teil der User-Level Bibliothek ist.

Wie bereits erwähnt, wird ein Thread bei der Ausführung der P-Funktion auf einem Semaphor/Mutex schlafen gelegt, wenn der Wert des Semaphor/Mutex bereits den Wert 0 besitzt. Erst wenn ein anderer Thread den Semaphor mit der V-Operation wieder freigibt, *kann* der Thread geweckt werden. Wenn aber mehrere Threads auf die Freigabe des kritischen Abschnitts warten, kann beim Verlassen eines Threads auch nur ein ein Thread den kritischen Abschnitt verlassen. Die Frage ist nun, welcher der Threads dies sein soll. Hier kommt der Begriff der ***Fairness*** ins Spiel. Die Auswahl des nächsten Threads für den Zutritt in den kritischen Abschnitt ist dann fair, wenn:

1. Jeder Thread in endlicher Zeit den kritischen Abschnitt betreten kann.
2. Ein länger wartender Thread zuerst den kritischen Abschnitt betreten darf.

Sind diese Eigenschaften nicht gegeben, könnte es passieren, dass ein Thread ewig auf den Eintritt in den kritischen Abschnitt warten muss, da ihm ständig ein anderer Thread zuvorkommt. Dies bezeichnet man als ***Aushungern*** eines Prozesses (engl.: ***Starvation***). Fairness wird in Hinsicht auf eine einfacher zu realisierende Implementierung in den meisten Bibliotheken nicht gewährleistet. Pthreads z.B. macht keine Aussage darüber, welcher der wartenden Threads in den kritischen Abschnitt entlassen wird, sobald ein anderer Thread ihn verlässt. Einige Pthreads-Bibliotheken verwenden hierzu priorisierte Warteschlangen. Dadurch wird gewährleistet, dass Threads mit höherer Priorität vor denen mit niedrigerer Priorität den kritischen Abschnitt betreten können. In BeOS hingegen wird nur eine Warteschlange pro Semaphor verwaltet. Wartende Threads werden ungeachtet ihrer Priorität an das Ende der Liste angehangen. Hier kommt immer der Thread zum Zug, der am längsten gewartet hat. Es handelt sich demnach um eine sehr faire Lösung. In Java 5.0 kann im Konstruktor eines Semaphors angegeben werden, ob die Zuteilung eines Threads fair sein soll oder nicht. Diese Einstellung bleibt für den Rest der Lebenszeit des Semaphor unveränderbar. Wurde ein fairer Semaphore erstellt, so werden die wartenden Thread wie bei BeOS in einer Warteschlange (FIFO-Prinzip) verwaltet und es kommt der Thread als nächstes zum Zug, der am längsten gewartet hat. Bei einem “unfairen“ (eher “nicht fairen“) Semaphor wird keine Aussage gemacht. Warum ist Fairness wichtig? Zum einen sicherlich um ein Aushungern eines Threads zu verhindern. Zum anderen kann Fairness auch zu einer besseren Lastenverteilung (engl.: ***Load-Balancing***) führen, da die Threads ihre Teilaufgaben gleichmäßiger ausführen können. Man betrachte ein Programm mit einem kritischen und einem nicht kritischen Anteil, wobei beide Teile gleich groß sein sollen. Das Programm soll von zwei Threads abgearbeitet werden. Bei “schlechter“ Fairness kann es z.B. passieren, dass zuerst nur der erste Thread rechnet und den zweiten Thread blockiert. Zum Schluss, wenn der

erste Thread fertig ist, rechnet nur noch der zweite Thread. Hier ist kaum Parallelität zu finden, obwohl der nicht-kritische Abschnitt parallel ausgeführt werden könnte.

Man kann sich auch fragen, ob es wirklich fair ist, wenn ein Thread mit hoher Priorität auf Threads mit niedriger Priorität warten muss, um in den kritischen Abschnitt zu gelangen, denn schließlich geht die Priorisierung hierdurch verloren. Außerdem wird die Abarbeitung des niedrig-priorisierten immer wieder durch den höher-priorisierten Thread unterbrochen. Der teuere Kontextwechsel ist jedoch vergebens, da der hochpriorisierte Thread nicht weiterarbeiten kann. Es wäre demnach sinnvoll, wenn der niedrig priorisierte Thread kurzzeitig eine höhere Priorität zugewiesen bekäme, um den kritischen Abschnitt so schnell wie möglich verlassen zu können und danach dem hochpriorisierten Thread wieder die Kontrolle überlassen zu können. Diese ***Priority Inversion*** genannte Taktik kann zum Beispiel bei der Verwendung von Mutexes einer Pthreads-Bibliothek verwendet werden. Der Pthreads-Standard definiert zwei verschiedene Ausprägungen der Priority Inversion, die jedoch nicht bei allen Implementierungen zur Verfügung stehen müssen. Bei der ersten Variante (festgelegt durch das Mutex-Attribut PTHREAD_PRIO_INHERIT) wird einem Thread, der einen Mutex für einen kritischen Abschnitt besitzt und dadurch höher priorisierte Threads blockiert, die Priorität zugewiesen, die der höchsten Priorität der auf den Mutex wartenden Threads entspricht. Bei der zweiten Variante (PTHREAD_PRIO_PROTECT) wird dem Mutex selbst eine Priorität zugewiesen. Blockiert ein Thread einen oder mehrere solcher Mutexe, wird ihm die höchste Priorität der von ihm blockierten Mutexe verliehen.

Programmierer, die damit anfangen, sich in Pthreads einzuarbeiten, begehen zu Beginn meist einen folgenschweren Programmierfehler. Nehmen wir an, es soll erreicht werden, dass ein Thread so lange wartet, bis ein anderer Thread eine bestimmte Stelle im Programmtext erreicht hat. Dazu wird ein Mutex mit Anfangswert 0 (zu Beginn einmal pthread_mutex_lock() für den Mutex ausführen) angelegt und gesperrt (nochmals pthread_mutex_lock()). Der Thread wird nun wie gewünscht schlafen gelegt. Wenn der andere Thread die festgelegte Stelle im Programm erreicht hat, ruft dieser pthread_mutex_unlock() auf den Mutex auf. Daraufhin wird der schlafen gelegte Thread geweckt und kann weiterlaufen. Zumindest *glaubt* das der Programmierer. In Wirklichkeit ist es nur ein Zufall, dass das gewünschte Verhalten gezeigt wird, denn der Pthread-Standard sieht vor, dass ein Mutex nur durch seinen Besitzer entsperrt wird. Da Pthreads im Normalfall die korrekte Verwendung der lock/unlock-Methoden nicht überprüfen, bekommt er keinen Fehler zurückgemeldet. Nach Aussage der man-pages zu pthread_mutex_lock() bzw. pthread_mutex_unlock() ist dieses Verhalten nicht auf andere Implementierungen der Pthreads übertragbar. Der Zustand des Mutex wird deshalb als undefiniert angesehen. Sowohl Win32 als auch .NET verwenden an dieser Stelle die gleiche Logik. Es sind aber durchaus auch Varianten vorstellbar, in denen ein Thread einen Mutex, der von einem anderen Thread gesperrt wurde, entsperren kann. Dieses verhalten kann man in Win32 mittels Events erreichen, da diese nach einer Signalisierung im Gegensatz zu Bedingungsvariablen nicht zurückgesetzt werden (siehe MSDN zu ResetEvent() und WaitForSingleObject()).

Nach der Definition eines Semaphors wird bei jeder P-Operation die Zählvariable dekre-

mentiert und der Thread gegebenenfalls schlafen gelegt. Was passiert aber, wenn ein Thread z.B. einen Mutex nicht nur einmal, sondern mehrmals gewollt oder ausversehen sperrt? Bei strenger Auslegung müsste der Thread blockieren und warten, bis ein anderer Thread den Mutex entsperrt. Das ist aber z.B. in Pthreads nicht möglich, da nur der Besitzer des Mutex diesen entsperren kann. Dieser ist dazu aber nicht in der Lage, da er schlafen gelegt wurde – ein Deadlock ist entstanden. Ein Threading-System hat folgende Möglichkeiten, auf das mehrfache Sperren eines Mutex zu reagieren:

1. Jeder Mutex darf nur einmal gesperrt werden. Deadlocks werde in Kauf genommen.
2. Ein Mutex kann von seinem Besitzer mehrmals gesperrt werden (***rekursiver Mutex***).
3. Es wird mit einem Fehler abgebrochen, wenn ein Mutex mehrmals gesperrt wurde.

Die zweite Variante vermeidet Deadlocks. Viele Threading-Systeme, darunter Pthreads, bieten diese sogenannten rekursiven Mutexe an. Ein Mutex kann dadurch mehrmals von demselben Thread gesperrt werden. Allerdings wird er danach erst wieder entsperrt, wenn die gleiche Anzahl von Entsperr-Operationen (V-Aufrufe) ausgeführt wurde. Der Nachteil hierbei ist die langsamere Ausführung der Sperr-Vorgänge, da die Implementierung aufwendiger ist.

Aber nicht nur, wenn ein Thread versucht, einen von ihm selbst gesperrten Mutex zu sperren, kann es zu Deadlocks kommen. Auch wenn Threads zyklisch auf die Freigabe eines von einem anderen Thread gesperrten kritischen Bereich warten, sind Deadlocks vorprogrammiert. Eine Möglichkeit, diese Deadlocks aufzulösen wäre es, beim Aufruf der lock-Funktion ein Timeout zu definieren, nach dem der Thread die Aquirierung des Locks aufgibt und aus dem Schlaf zurückkehrt. Win32-Threads bieten diese Möglichkeit sowohl für Mutexe als auch für Semaphore. Diese Funktionalität sollte jedoch nur genutzt werden, um das Programm im Deadlockfall mit einer Fehlermeldung abbrechen zu können. Es sollte aber auf keinen Fall schlampigen Programmierstil fördern. Die Funktion kann leicht dazu verführen, die saubere Synchronisation eines Programmes zu vernachlässigen, da man ja, wenn die Synchronisation doch nicht ganz korrekt war, einem Deadlock durch den Timeout entgehen kann. Wurde der Timeout ausgelöst, da ein Deadlock auftrat, versucht man es eben einfach noch mal neu, in der Hoffnung, dass es nun schon funktionieren wird. Das ist aber grundlegend falsch. Man sollte falsche Synchronisierung auf keinen Fall einfach Ignorieren, da die Korrektheit des Programmes im Ganzen in Frage gestellt ist.

Blockiert ein Thread, der einen Mutex für sich sperren wollte, kann er keine weiteren Aktionen ausführen. Hierdurch könnte Parallelität verlorengehen, da der Thread evtl. in dieser Zeit eine andere Aufgabe verrichten könnte, für die er das Lock nicht braucht. Fast alle Threading-Systeme bieten hierzu eine Funktion der Form mutex_trylock() an. Hierbei versucht der Thread, den Mutex zu sperren, blockiert aber nicht, wenn dies nicht gelingen sollte. Somit kann ein Thread nach einer gescheiterten Anfrage einer anderen Aufgabe nachgehen und in regelmäßigen Abständen den Status des Locks prüfen. Der Thread muss sich auf diese Weise nicht schlafen legen, sondern kann bis zum Erwerb des Locks einer sinnvollen Aufgabe nachgehen.

Nach [NBF98, S. 68-69] ist diese Vorgehensweise jedoch nicht zu empfehlen. So wäre es sauberer, zuerst einen neuen Thread zu erzeugen, an den die Arbeit delegiert werden kann, und dann zu versuchen, den Mutex zu erlangen. Hierbei bleibt die Parallelität erhalten und man erhält eine bessere Programmstruktur. Allerdings ist die Erzeugung von neuen Threads sehr zeitaufwendig, wodruch man sich die verbesserte Struktur teuer erkauft. Ein weiteres Problem bei mutex_trylock() ist es, dass die Zuweisung eines Locks an einen Thread, der nicht blockierend auf dieses Lock wartet, nur noch weiter verzögert wird. Das Lock muss nämlich beim Aufruf der Funktion entweder bereits freigegeben worden sein oder genau zu dem Zeitpunkt des Aufrufs frei gegegeben werdens. Das ist in vielen Fällen nicht der Fall. Somit könnte der Thread ausgehungert werden (vgl. [NBF98, S. 69]).

Interessant wäre es in einigen Fällen auch zu wissen, welcher Thread einen bestimmten Mutex gesperrt hat. Hierdurch könnte man eine Nachricht an den Thread schicken, den Mutex zu entlocken, z.B. um einen Deadlock zu vermeiden (“hold-and-wait“ Bedingung nicht mehr erfüllt). Java und BeOS bieten eine solche Funktionalität, Pthreads hingegen aus Geschwindigkeitsgründen nicht. Eine Pthreads-Implementierung, wie in AIX der Fall, kann aber durchaus eine nicht-portable Funktion hierfür anbieten.

Die P- und V-Operationen müssen einen Bezug zu dem Semaphor oder dem Mutex besitzen, der heruntergezählt bzw. gesperrt werden soll. Ein Semaphor bzw. Mutex kann dazu mittels eines Handles, eines Zeigers auf eine Datenstrukturen oder einer Objekt-Referenz identifiziert werden. In Win32 können Semaphore/Mutexe aber auch über Namen (Zeichenketten) angesprochen werden. Der Vorteil hierbei ist, dass zuvor kein Lock-Objekt angelegt werden muss. Ein weiterer Vorteil ist, dass Lock-Objekte somit auf einfache Weise von Threads verschiedener Prozesse verwendet werden können. Man benötigt keinen gemeinsamen Adressraum, sondern einfach nur den Namen des Locks. Damit kann man dem Betriebssystem anweisen, das dazugehörige Lock-Objekte zu übergeben. Das funktioniert natürlich nur, solange das Threading-System die Verwendung von Locks zwischen verschiedenen Prozessen erlaubt. In Win32 und Pthreads ist dies der Fall.

Bestimmte Programmabschnitte, darunter Initialisierungen, sollen meist nur von dem ersten Thread durchgeführt werden, der den entsprechen Abschnitt erreicht. Eine einfache Lösung hierfür wäre es, den Block in einen kritischen Abschnitt zu packen, in dem durch eine if-Bedingung getestet wird, ob der Abschnitt bereits durchlaufen wurde. Es gibt aber z.B. in Pthreads eine Funktion namens pthread_once(once_control , routine), die genau dieses Verhalten liefert. Die Datenstruktur once_control, die zuvor initialisiert werden muss, speichert hierbei ab, ob bereits ein Thread den Abschnitt betreten hat. Erreicht ein Thread den Funktionsaufruf und der Abschnitt wurde noch nicht betreten, so führt der Thread den Abschnitt in Form der Routine routine aus. Ansonsten wird die Funktion ignoriert. Es ist fraglich, inwiefern eine solche Funktion eine Vereinfachung bringt. Der Aufwand im Vergleich zur einfachen Lösung dürfte der gleiche sein. Lediglich die Lesbarkeit des Quelltextes dürfte hierdurch erhöht werden.

Auf der einen Seite ist es sinnvoll, kritische Abschnitte im Quelltext so klein wie möglich zu halten, da diese – zumindest, wenn nur ein Thread ihn betreten darf – nicht parallelisiert werden

können. Auf der anderen Seite wäre es evtl. möglich zwei benachtbarte kritische Abschnitte miteinander zu vereinen, wenn man einige Instruktionen hinzunimmt, die eigentlich nicht in einen kritischen Abschnitt müssen. Dies kann in einigen Fällen sinnvoll sein, da der Eintritt in einen kritischen Abschnitt (z.B. wegen langsamer Systemaufrufe) meist sehr lange dauert. Dies trifft vor allem dann zu, wenn die kritischen Abschnitte sehr klein sind, wie z.B. bei Zuweisung eines Additionsergebnisses an eine Variable. Einige Thread-Systeme bieten als Alternative zu sehr kleinen kritischen Abschnitten sogenannte ***atomic locks***, die einzelne *einfache* Anweisungen atomisch Ausführen. Java 5.0 bietet hier spezielle Klassen an, die Basisdatentypen wie Integer oder Objekt-Referenzen kapseln. Über die Methoden AtomicInteger.addAndGet(delta) oder AtomicInteger.getAndSet(newValue) können z.B. AtomicInteger-Objekte um einen Wert value erhöht werden und der neue Wert zurückgegeben werden (addAndGet()) bzw. der Wert des Objekts neu gesetzt werden und der alte Wert zurückgegeben werden. Das besondere hierbei ist, dass die hierzu benötigten Instruktionen atomisch, also ununterbrechbar, ausgeführt werden. Einfache Additionen müssen daher nicht mehr durch einen eigenen kritischen Abschnitt geschützt werden.

Da Locks oftmals nur für sehr kurze Zeit gesperrt werden, wäre es eigentlich sinnvoll, einen Thread nicht sofort schlafen zu lassen, da das Schlafenlegen und Aufwecken des Threads sehr viel länger dauern dürfte als die verbliebene Zeit, für die der sperrende Thread das Lock noch braucht. Durch die Verwendung von ***Spin-Locks*** kann hier entgegengewirkt werden.

Oftmals kann man in Programmen eine typische ***Leser-Schreiber*** (engl.: ***Reader-Writer***) Struktur erkennen. D.h. es gibt Threads, die nur lesend auf einen gemeinsamen Speicher zugreifen wollen und Threads die in den gemeinsamen Speicher schreiben wollen. Wie bereits erwähnt stellt ein gleichzeitiges Lesen der Speicherstelle durch mehrere Leser kein Problem dar. Ein Schreiber darf allerdings nicht zusammen mit anderen Lesern oder Schreiber auf den Speicher zugreifen. Es ist jedoch nicht ratsam, den kritischen Abschnitt durch ein Lock für jeweils nur einen Thread zu sperren, da durchaus mehrere Leser den kritischen Abschnitt betreten dürften. Eine Abhilfe schaffen sogenannte ***Reader-Writer Locks***. Sobald der erste Leser den kritischen Abschnitt betritt, können weitere Leser folgen, Schreiber hingegen werden jedoch ausgesperrt. Hat sich der letzte Leser aus dem kritischen Abschnitt entfernt, wird den Schreibern ein Eintritt in den kritischen Abschnitt wieder gewährt. Betritt ein Schreiber zuerst den kritischen Abschnitt, wird der Eintritt für alle anderen Threads, sowohl Leser als auch Schreiber, blockiert. Dieses Verhalten kann in allen Sprachen nachgebildet werden. Hierbei können Programmierfehler dazu führen, dass es zu Deadlocks kommt oder der wechselseitge Ausschluss nicht korrekt funktioniert. Da Reader-Writer Locks häufig verwendet werden und sich deren Struktur nur geringfügig unterscheidet, bieten einige Threading-Systeme wie Pthreads, Java 5.0 und .NET diesen Sperrmechanismus bereits standardmäßig an. Die zuvor genannten Beschreibung der Reader-Writer Locks bevorzugte die Leser beim Eintritt in den kritischen Abschnitt. Kommen immer wieder neue Leser nach, ergibt sich für einen Schreiber keine Möglichkeit, in den kritischen Abschnitt zu gelangen. Er kann dadurch ausgehungert werden. Es ist genausogut möglich, die Schreiber zu bevorzugen. Sobald ein Schreiber den kritischen Abschnitt anfordert,

können die Leser, die sich noch im kritischen Abschnitt befinden, diesen normal beenden. Danach erhält jedoch sofort der Schreiber eine Sperre auf den kritischen Abschnitt. Hierbei können wenige Schreiber eine große Anzahl von Lesern blockieren, was das Programm stark verlangsamen könnte und ebenfalls nicht fair ist. Eine faire Strategie, bei der Leser und Schreiber gleiche Chancen haben, den kritischen Abschnitt zu betreten, ist nicht leicht zu implementieren. Sollte ein Threading-System eine faire Implementierung anbieten, erspart dies dem Programmierer viel Arbeit. Folgende Stragtegien der Auswahl von am kritischen Abschnitt anstehenden Lesern bzw. Schreibern sind möglich:

STRATEGIE	BESCHREIBUNG
Leser bevorzugt	Solange sich noch ein Thread im kritischen Abschnitt befindet, dürfen neu ankommende Threads den Abschnitt betreten.
Schreiber bevorzugt	Schreiber haben Vorrang vor den Lesern. Sobald der erste Schreiber am kritischen Abschnitt eintrifft, darf kein weiterer Leser eintreten.
FIFO-Ordnung	Leser/Schreiber werden in der Reihenfolge ihres Eintreffens am kritischem Abschnitt eingelassen. Es gibt keine Bevorzugung.
Keine	Leser/Schreiber bekommen in einer nicht vorhersehbaren Reihenfolge Zugriff auf den kritischen Abschnitt.

Bei der Erzeugung eines Read-Write Locks der Klasse ReentrantReadWriteLock unter Java 5.0 kann der Programmierer im Konstruktor mittels eines Fairness-Parameters selbst entscheiden, welche Strategie angewandt werden soll. Zur Auswahl steht eine faire Strategie (FIFO-Ordnung) und eine nicht-fairen Strategie, bei der, aus Sicht des Programmierers willkürlich, entweder ein Reader oder Writer eingelassen wird.

Bei einigen parallelen Programmen muss man kurz nachdem man einen kritischen Abschnitt freigegeben hat einen weiteren kritischen Abschnitt sperren. Betrachten wir z.B. ein Programm, in dem eine gemeinsame Variable hochgezählt und beim Erreichen eines bestimmten Wertes auf der Konsole ausgegeben werden soll. Sowohl die Variable als auch die Konsole müssen durch einen kritischen Bereich geschützt werden. Da die Konsole auch in anderen Programmabschnitten gebraucht wird, die nichts mit dem Hochzählen der Variable zu tun haben, sollen zwei getrennte kritische Abschnitte betrachtet werden. Sie sollen mittels der Locks “count_lock“ für das Hochzählen und “console_lock“ für die Ausgabe gesperrt werden.

```
1 lock(count_lock);
2 if(++count > VALUE) {
3     lock(console_lock);
4     output(count_value);
5     unlock(console_lock);
6 }
7 unlock(count_lock);
```

Ein Thread A, der dieses Programm ausführt, sperrt zuerst count_lock und zählt dann die Variable hoch. Besitzt die Variable den gewünschten Wert, wird console_lock gesperrt und die

Ausgabe getätigt. Schließlich werden die beiden Locks freigegeben. Dieser Programmablauf birgt jedoch die Gefahr eines Deadlocks. Was passiert, wenn der Thread A count_lock sperrt und kurz darauf ein anderer Thread B console_lock sperrt. Thread A blockiert und wartet, bis Thread B das Lock freigibt. Nachdem Thread B die Konsole für sich hat, soll er den Wert der Zählvariablen ausgeben. Er versucht, count_lock zu sperren und scheitert – ein Deadlock ist entstanden. In diesem Fall lässt sich der Deadlock durch ***hierarchisches Locking*** beheben. Hierbei werden alle Locks durchnummeriert. Ein Thread darf dann nur noch Locks sperren, deren Nummer größer ist, als die der bereits gesperrten Locks. Ansonsten muss er erst Locks freigeben und darf dann erst das gewünschte Lock sperren. Hierarchisches Locking muss jedoch vom Programmierer selbst programmiert werden. Ein Threading-System, dass eine solche Funktionalität anbietet, ist nicht bekannt.

Deadlocks treten sehr häufig im Zusammenhang mit der inkrementellen Anforderung einer beschränkten Anzahl von Ressourcen auf. Hierunter könnte man sich z.B. einen Ringbuffer oder ähnliches vorstellen, von dem ein Thread eine gewisse Anzahl von Speicherplätzen für sich reservieren kann. Man stelle sich einen Semaphor vor, der mit der Anzahl der freien Plätze im Ringbuffer initialisiert wird. Jeder Thread, der in den Ringbuffer schreiben will, muss den Semaphor um die Anzahl gewünschter Speicherplätze herunterzählen. Ist kein Platz mehr im Buffer, legt er sich schlafen. Der Ringbuffer soll nur dann geleert werden, wenn der Thread seinen aus mehreren Speicherplätzen bestehenden Datensatz vollständig in den Buffer geschrieben hat. Nehmen wir an, der Buffer hätte insgesamt 20 Speicherplätze. Der erste Thread will 15 Speicherplätze allokieren, der zweite 10. Verwendet man die Semaphore der Pthreads, muss der erste Thread 15x und der zweite 10x die Funktion sem_wait() aufrufen. Würde der Programmcode der beiden Threads sequentiell ablaufen, gibt es keine Probleme. Zuerst allokiert Thread1 15 Speicherplätze und gibt sie dann wieder frei. Danach kann Thread2 seine 10 Speicherplätze allokieren und ebenfalls wieder freigeben. Wird der Programmcode jedoch parallel ausgeführt, kann folgendes passieren: Thread1 ruft 11x sem_wait() auf, dann wird ein Kontextwechsel zu Thread2 durchgeführt, der 9x sem_wait() aufrufen kann. Dann blockiert er, da kein Speicher mehr frei ist. Thread1 kommt wieder zum Zug und blockiert ebenfalls. Da der Buffer nicht geleert werden kann, tritt ein Deadlock ein.

Solche Deadlocks könnten ganz einfach vermieden werden, wenn der Semaphor nicht inkrementell, sondern auf einen Schlag neu gesetzt werden könnte. Hierzu müsste man anstatt n-mal die P() Operation auszuführen, einmal eine P(n) Operation aufrufen können. Kann der Semaphor die Anforderung des Threads nicht vollständig erfüllen, wird der Wert des Semaphor nicht verändert und der Thread schlafengelegt. Unterstützt ein Semaphor diese Syntax, bezeichnet man ihn als ***additiven Semaphore***. Im Gegensatz zu Pthreads unterstützen die Semaphore von Java 5.0 diese Funktionalität. In Win32 ist hingegen zwar eine V(n)-Funktion für den Semaphor vorhanden, eine P(n)-Funktion fehlt jedoch, ließe sich aber mit WaitForMultipleObjects () etwas umständlich nachbauen.

Was aber, wenn man nicht nur mehrere Ressourcen eines sondern mehrerer Typen haben will. Hier helfen auch additive Semaphore nicht. Man benötigt Gruppensemaphore. Die in die

SystemV IPC-Bibliothek (InterProcess Communication) eingebauten Semaphore (siehe Linux man-pages zu semget()) sind auf diese Weise implementiert. Hierbei wird nicht nur ein Semaphor, sondern eine ganze Semaphorengruppe erzeugt. Will ein Prozess den Wert des Semaphors verändern, schreibt er in eine Datenstruktur, wie viele Ressourcen er von jedem Semaphor dieser Gruppe freigeben bzw. sperren will. Nur wenn die Anforderung im Ganzen erfüllt werden kann, wird sie auch ausgeführt. Andernfalls wird der Thread schlafengelegt. Man sollte die IPC-Calls jedoch nicht mit Pthreads verwenden. Zum einen gehören sie nicht zum POSIX-Standard, wodurch die Portabilität leidet, zum anderen könnte es sein, das die Implementierung nicht threadsicher ist. Für weitere Informationen zu additiven Semaphoren und Gruppensemaphoren und Informationen wie man diese (in Java) implementieren kann, sei auf [Oec01] verwiesen.

Nachdem ein Lock verwendet wurde, muss es in einigen Threading-Systemen, wie Pthreads oder Win32-Threads, explizit vom Programmierer wieder freigegeben werden.[3] Ansonsten werden die belegten Resourcen nicht freigegeben und es droht ein Speichermangel. In Java geschieht dies hingegen automatisch. Die Gefahr beim manuellen Freigeben besteht nun darin, dass ein Lock freigeben werden könnte, das momentan noch gesperrt ist. In Win32 hat ein solches Lock einen undefiniertem Zustand. Deadlocks könnten auftreten. Pthreads hingegen lässt einen Aufruf zum Freigeben des Locks gar nicht erst zu und gibt stattdessen eine Fehlermeldung zurück.

4.5 Bedingungsvariablen

In vielen parallelen Programmen müssen Threads darauf warten, dass eine spezielle Bedingung erfüllt ist, z.B. ein Puffer gefüllt wurde. Hier könnte man mit Busy-Waiting in einer Schleife die Erfüllung der Bedingung testen. Dieses Vorgehen verbraucht aber unnötig Prozessorleistung. Außerdem muss beim Testen einer Variable in jedem Schleifendurchgang ein Lock gesperrt werden, da es sich hierbei um Zugriff auf gemeinsamen Speicher handelt. Eine Alternative die sich in fast allen Threading-Systemen (darunter Java, Pthreads und Perl) durchgesetzt hat, sind die Bedingungsvariablen. Mittels zweier Funktionen cond_wait() und cond_signal() kann auf die Erfüllung der Bedingung gewartet oder diese signalisiert werden. Hierzu wird in Pthreads eine Datenstruktur namens pthread_cond_t definiert, die jedoch nur für jeweils eine Bedingung verwendet werden sollte. Sie wird Bedingungsvariable genannt. Man muss hierbei beachten, dass diese Datenstruktur keinerlei Informationen enthält, worum es sich bei der zu erfüllenden Bedingung handelt. Der Thread der die Bedingung erfüllt, muss daher wissen, dass zum einen ein anderer Thread auf die Erfüllung dieser Bedingung wartet und zum anderen welche Bedingungsvariable signalisiert werden soll. Ein Thread testet zuerst, ob die Bedingung, auf die er wartet erfüllt ist. Ist dies nicht der Fall, ruft er die cond_wait() Funktion auf die Bediungsvariable auf. Allerdings muss er zuvor (auch vor dem Testen der Bedingung) einen kritischen Abschnitt gesperrt haben. Jede Bedingungsvariable ist fest mit eoiner Bedingung und einem kritischen

[3] In Pthreads müssen z.B. Mutexe mit der Funktion pthread_mutex_destroy() freigegeben werden.

Abschnitt verbunden. Wartet ein Thread auf eine Bedingungsvariable, legt er sich schlafen und gibt den kritischen Abschnitt frei. Dadurch ist es anderen Threads möglich, in den kritischen Abschnitt einzutreten und die Bedingung zu erfüllen. Wird eine Bedingung signalisiert, wird der wartende Thread zwar geweckt, muss sich meist jedoch erneut für den kritischen Abschnitt bewerben. Es wird immer nur ein Thread geweckt. Welcher das ist, ist in den meisten Fällen nicht definiert. Es gibt jedoch meist auch eine Funktionen cond_broadcast, mit der alle Threads auf einmal geweckt werden können.

Ein wichtiger Unterschied zwischen den Systemen ist es, ob die Bedingung nach der Signalisierung des schlafenden Threads von diesem noch einmal geprüft werden muss, oder ob dieser davon ausgehen kann, dass die Bedingung nun korrekt ist. In Java und Pthreads ist dies auf jeden Fall nötig, da in der Zwischenzeit ein anderer Thread zur Ausführung gekommen sein kann, der den Wert der Bedingung verändert hat. In den Threading-Systemen unterscheidet sich auch der Gültigkeitsbereich des cond_signal () Aufrufs. So darf diese Funktion in Java nur innerhalb eines Monitors aufgerufen werden. In Pthreads ist jedoch durchaus erlaubt, sie außerhalb eines kritischen Abschnitts aufzurufen.

Als letzter Unterschied sei das Verhalten von cond_signal () genannt, wenn kein Thread auf die signalisierte Bedingung wartet. In Pthreads und Java wird das Signal in diesem Fall verworfen. In Win32-Threads, in denen mit Events eine etwas andere Variante der Signale implementiert wurde, werden Events, wenn sie signalisiert wurden, im Normalfall jedoch nicht zurückgesetzt. Erst wenn ein Thread auf diesen Event wartet, wird er zurückgesetzt, aber auch dieses Verhalten ist optional.

Bislang wurden nicht für alle Aspekte der Bedingungsvariablen Beispiele für Threading-Systeme angegeben. Das liegt daran, dass sich im Kapitel 4.6 viele der hier erwähnten Konzepte überlagern. Daher soll an dieser Stelle auf dieses Kapitel verwiesen werden.

4.6 Monitore

Das Prinzip der Monitore ist sehr stark mit dem der Mutexe verwandt. Die Synchronisierung per Mutex ist sehr fehleranfällig. Der Benutzer muss selbst dafür sorgen, dass die Mutexe im Laufe des Programmes an den richtigen Stellen gesperrt oder entsperrt werden. An dieser Stelle können sich leicht Fehler einschleichen, da z.B. ein zu spätes Sperren eines Mutex zu einer Race-Condition führen könnte, ein zu frühes Sperren jedoch zu einem Deadlock. An dieser Stelle greift das auf Ideen von C. Hoare [Hoa74] und P.B. Hansen [Han81] basierende Konzept der Monitore, das Mitte der 70er Jahre entstand. Monitore werden nicht vom Programmierer, sondern vom Compiler verwaltet. Es handelt sich somit um ein Programmiersprachen-Konstrukt. Man kann sich einen Monitor wie eine Klasse oder eine Struktur vorstellen, in der mehrere Methoden definiert werden können. Zu jedem Zeitpunkt darf immer nur ein Thread den Monitor betreten, d.h. Methoden des Monitors aufrufen. Im Prinzip funktioniert dies so, als würde ein Thread, der eine Methode des Monitors betritt, zu Beginn implizit einen Mutex sperren, der für

alle Methoden des gleichen Monitors verwendet wird. Weitere Threads müssen dann so lange warten, bis der erste Thread den Monitor verlassen hat. Man kann mit Monitoren also sehr leicht kritische Abschnitte schützen, da Mutexe ohne großen Aufwand durch Monitoren nachgebildet werden können.

In Hoares und Hansens Monitor-Modellen ist auch ein Bedingungsvariablen-Mechanismus vorgesehen. D.h. innerhalb eines Monitors (und auch nur dort), kann ein Thread auf die Erfüllung (signal) einer Bedingungsvariablen warten (wait). Nach einem wait() wird die Kontrolle des nun wartenden Threads über den Monitor abgeben. So lange, bis ein anderer Thread im Lauf seiner Abarbeitung dafür sorgt, dass die Bedingung eintritt. Die Erfüllung der Bedingung wird dem schlafenden Thread entweder explizit oder implizit signalisiert, wodurch der wartende Thread weiterarbeiten kann. Wem diese Beschreibung eines Monitors noch sehr schwammig vorkommt, hat vollkommen Recht. Das liegt vor allem daran, dass es sehr viele Implementierungs-Möglichkeiten gibt, die jeweils verschiedene Monitorverhalten hervorrufen. Die Unterschiede sollen in diesem Kapitel etwas genauer betrachtet oder zumindest angerissen werden. Mittlerweile gibt es einige Threading-Systeme, die Monitore unterstützen, darunter Java und .NET. Da Monitore nach dieser Definition in die Programmiersprache eingebaut sind, können Threading-Systeme, die als Bibliothek realisiert wurden, keine Monitore anbieten.

Welche Unterschiede gibt es nun in Bezug auf die Monitore? Da wäre zum einen die Unterscheidung, ob das Aufwecken eines Threads *implizit* oder *explizit* geschehen muss. Bei der expliziten Variante muss ein Thread, der auf die Erfüllung einer Bedingung wartet, ein wait() auf einer zuvor hierfür angelegten Bedingungsvariablen aufgerufen haben. Ein Thread, der dafür sorgt, dass die Bedingung eintrifft, ruft ein signal() auf der Bedingungsvariable auf, und weckt damit (gegebenenfalls) den schlafenenden Thread. Da eine Bedingungsvariable jeweils nur für eine zu erwartende Bedingung stehen sollte, wäre es schön, ganz auf Bedingungsvariablen zu verzichten und direkt auf die Erfüllung der Bedingung zu warten. Ein Thread, der darauf wartet, dass ein Puffer vollständig gefüllt ist, könnte dies dann so tun:

```
1 wait(buffer_size > buffer_max);
```

Ein Problem hierbei wäre, dass die in dem wait-Aufruf angegebene Bedingung nur einmal vor dem Schlafenlegen ausgewertet wird, was zu einem niemals endendem Schlaf führen würde. Dies ließe sich jedoch dadurch umgehen, den wait-Befehl in ein Schlüsselwort der Programmiersprache umzuwandeln, das gesondert behandelt wird. Im Falle der Programmiersprache C wären an Stelle der Bedingung auch Funktionzeiger möglich. Der Nachteil ist jedoch der höhere Aufwand der Auswertung im Gegensatz zu Bedingungsvariablen. Hier müsste quasi nach jedem Kontextwechsel oder sogar jeder Rechenoperation eines Threads sämtliche Warte-Bedingungen neu ausgewertet werden, da eine Bedingung nun erfüllt sein könnte. Diese Variante ist daher sehr ineffizient und ist in den bekannteren Threading-Systemen nicht anzutreffen. Alle in dieser Arbeit betrachteten Threading-Systeme arbeiten mit expliziter Signalisierung. Die implizite Variante wurde jedoch in einer in den 70er Jahren von Brinch Hansen entwickelten Sprache namens Edison, die speziell für Multiprozessor-Systeme gedacht war, verwirklicht [Weg03].

Ein weiteres Kriterium betrifft die Semantik des signal-Befehls. Was passiert mit dem Thread der ein signal () aufgerufen hat? Kann er weiterlaufen, bis er den Monitor verlassen hat oder muss er sofort nach dem Aufruf die Kontrolle an den wartenden Thread übergeben? Hier gibt es zwei Ansätze, die zu verschiedenen Monitor-Typen führen, zum einen den Hoare'schen Typ und zum anderen den Mesa Typ [wik06b].

Der Hoare'sche Monitor-Typ Er ist nach seinem Erfinder C.A.R. Hoare benannt. Sollte ein Thread ein signal () auf eine Bedingungsvariable ausführen, auf die ein anderer Thread wartet, so muss der signalisierende Thread die Kontrolle umgehend an den wartenden Thread übergeben und sich selbst schlafen legen. Der signalisierende Thread darf erst weitermachen, nachdem der aufgeweckte Thread den Monitor freigegeben hat. Dieser Monitor-Typ bietet einen nicht zu unterschätzenden Vorteil, da die Bedingung auf die der nun aufgeweckte Thread gewartet hat, nach dem Aufwecken auf jeden Fall erfüllt ist. So kann es z.B. nicht sein, dass ein weiterer Thread zwischen der Signalisierung und dem Aufwachen des wartenden Threads die Erfüllung der Bedingung wieder rückgängig gemacht hat. Somit ist ein testen der Bedingung nach dem Aufwachen und ein evtl. damit verbundenes erneutes Schlafenlegen, wenn die Bedingung doch nicht erfüllt sein sollte, unnötig. Ein signal () darf daher nur innerhalb und nicht außerhalb eines Monitors aufgerufen werden. Allerdings schafft diese Implementierung auch größe Probleme. Im Prinzip befinden sich nach der Übergabe des Monitors durch ein signal zwei Threads im Monitor. Nachdem der signalisierende Thread weiterrechnen darf, könnte ein anderer Thread die vom Monitor geschützten Daten verändert haben, wodurch der eigentliche Sinn des Monitors ad absurdum geführt wird. In [Weg03] wird auch auf das Problem hingewiesen, das entsteht, wenn der geweckte Thread kurz nach dem Aufwachen bereits wieder auf eine andere Bedingung warten sollte. Darf in diesem Fall der zuvor schlafen gelegte signaliserende Thread weiter machen? Aus diesen Gründen war Per Brinch Hansen der Meinung, dass ein signal $-$() nur am Ende einer Monitor-Prozedur stattfinden darf [Han81]. Hoare hingegen zeigte sich jedoch in [Hoa74] nicht davon überzeugt. Der Autor des bereits zitierten Werkes [Weg03] teilt die Meinung von Brinch Hansen. Im Allgemeinen dürften die Meinungen in dieser Hinsicht, wie bereits bei den beiden Vordenkern, auseinander gehen. Ein grundlegendes Problem der Hoare'schen Monitore ist jedoch, dass ein broadcast (), also ein signal , dass alle wartenden Threads auf einmal weckt, nicht möglich ist. Dies liegt daran, dass nicht alle geweckten Threads gleichzeitig den Monitor betreten können. Einer der geweckten Thread kann die Erfüllung der Bedingung nichtig machen, somit müssten die weiteren Threads doch prüfen, ob die Bedingung noch gilt. Der Vorteil dieses Monitor-Typen wäre damit aufgehoben. Keines der in dieser Arbeit betrachteten Threading-Systeme verwendet den Hoare'schen Monitor-Typen. Es wird lediglich der im Folgenden erklärte Mesa Monitor-Typ verwendet.

Der Mesa Monitor-Typ Er wurde nach der von Xerox PARC entwickelten Programmiersprache Mesa benannt, in der er zum ersten Mal zum Einsatz kam. Im Gegensatz zum Hoare'schen Monitor behält der signalisierende Thread die Kontrolle über den Monitor. Erst nachdem der

Thread freiwillig den Monitor verlassen hat, wird ein wartender Thread aufgeweckt. Wenn der wartende Thread geweckt wurde und weiterrechnet, kann es jedoch sein, dass die Bedingung, auf deren Erfüllung der Thread gewartet hat, bereits nicht mehr erfüllt ist. Zum Beispiel könnte ein anderer Thread in der zwischen der Signalisierung der Bedingung und dem Aufwachen des Threads die Kontrolle des Monitors übernommen haben und etwas getan haben, wodurch die Bedingung nicht mehr erfüllt ist. Zum Beispiel könnte man sich vorstellen, dass zwei Verbraucher-Threads darauf warten, dass ein Puffer gefüllt wird. Nachdem ein Erzeuger-Thread die Kontrolle über den Monitor an sich genommen hat und den Puffer mit einem Element gefüllt hat, weckt er beide Verbraucher-Threads. Nachdem der Erzeuger den Monitor verlassen hat, übernimmt der erste Verbraucher den Monitor und leert den Puffer. Er verlässt den Monitor und der zweite Verbraucher versucht sein Glück und greift ins Leere. Ohne Prüfung der Bedingung würde das Programm ab dieser Stelle entweder fehlerhaft weiterlaufen oder abstürzen. Daher ist es unbedingt notwendig, die Bedingung nach dem Aufwachen nochmals zu prüfen. Sollte die Bedingung nicht mehr erfüllt sein, muss sich der Thread wieder schlafen legen, bis die Bedingung wieder erfüllt ist. Die Signalisierung einer Bedingungsvariablen bedeutet daher nicht unbedingt, dass ein aufgeweckter Thread die Bedingung auch wirklich erfüllt vorfindet. Um dies auszudrücken, wird in Threading-Systemen, wie z.B. Java, die diesen Monitor-Typ unterstützen, häufig auch der Befehl notify () anstatt signal () verwendet. Die hiermit einhergehende notwendige Prüfung der Variablen ist der Nachteil im Gegensatz zu dem Hoare'schen Monitor. Die Programmierung wird hierdurch jedoch viel sauberer. So kann ein signal () an jeder Stelle einer Monitor-Prozedur geschehen und muss nicht am Ende der Monitor-Prozedur erfolgen. Es wäre sogar denkbar, eine Signalisierung außerhalb des Monitors zuzulassen. In Java ist dies mit Object.signal() nicht möglich, da in diesem Fall eine Exception ausgelöst wird. Bei diesem Monitor-Typen ist sogar eine Signalisierung aller Threads gleichzeitig (broadcast ()) möglich.

Unterscheiden lässt sich bei beiden Typen, wie dies in ähnlicher Weise bereits bei Mutexen und Semaphoren der Fall war, welcher schlafende Thread nach einem signal () aufgeweckt wird. In Java gibt es hierzu keine Garantie [Oec01, S. 55], es wird zufällig einer der schlafenden Threads geweckt. In anderen Systemen wären aber z.B. auch FIFO- oder durch Prioritäten gesteuerte Warteschlangen möglich. Da der Begriff Warteschlange meist eng mit dem FIFO-Prinzip verbunden ist, sollte an dieser Stelle eher von einem Warteraum gesprochen werden. Pro Monitor kann es jedoch mehrere getrennte Warteräume geben. So wird in [Weg03] nach Warteräumen unterschieden für Threads,

1. die neu in den Monitor eintreten wollen (Entry-Queue)
2. die nach einem wait () geweckt wurden und nun den Monitor anfordern (Waiting-Queue)
3. die nach einem signal () wieder in den Besitz des Monitor kommen wollen (Signaller-Queue)

Die Threads dieser Warteräume können unterschiedlich priorisiert werden. In [Weg03] werden hierzu 13 Priorisierungen aufgelistet, von denen allerdings nur 6 als praktikabel bewertet werden. Sei E die Priorität der Entry-Queue, W die der Waiting-Queue und S die der Signaller-Queue. Ein als Signal-and-Urgent-Wait Monitor bezeichneter Monitor hat z.B. die Priorisierung E < S < W. Dies würde z.B. dem Hoare'sche-Monitor entsprechen. Monitore mit W < S würden hingegen dem Mesa-Typen entsprechen. Da in den bekannstesten Monitor-Implementierung meist Monitore mit W < S verwendet werden, soll an dieser Stelle eine weitere Betrachtung dieses Themas ausbleiben. Für eine ausführliche Diskussion dieses Themas sei auf [Weg03] verwiesen.

Nun wollen wir kurz eine konkrete Monitor-Implementierungen anhand von Java betrachten. In die Programmiersprache ist ein Schlüsselwort namens "synchronized" integriert. Der Monitor ist in Java quasi ein Objekt oder eine Klasse. Deklariert man eine Methode als synchronized, sorgt der Compiler für die Synchroniserung der Methode. Ruft ein Thread eine mit synchronized markierte Objekt-Methode auf, so wird intern das Objekt, auf der die Methode aufgerufen wurde, gesperrt (Der Monitor wurde betreten). Ruft nun ein zweiter Thread die gleiche oder eine weitere synchronized-Methode des gesperrten Objekts auf, muss der Thread warten, bis der erste Thread mit der Ausführung der Methode fertig ist. Wird im Gegensatz dazu eine synchronized Klassenmethode aufgerufen, wird ein lock auf das Klassenobjekt und nicht auf eine Instanz ausgeführt. Locks auf Klassenobjekte und Instanzen dieser Klasse stören sich jedoch gegenseitig nicht.

Den Begriff des Monitors wird kann auch anders definiert werden, als es in dieser Arbeit getan wurde. In einer alternativen Definition müssen Monitore nicht unbedingt in die Programmiersprache integriert sein. So können Monitore z.B. auch über Locks oder CriticalSections nachgebildet werden. In .NET existiert z.B. eine Klasse Monitor, mit der man analog zu Javas synchronized-Blöcken und den lock-Blöcken in C# einen Monitor nachbilden kann. Hierzu muss jedoch ein Objekt angegeben werden, dass von dem Monitor gesperrt werden soll. Alle Bereiche die durch Monitore geschützt werden, die das gleiche Objekt sperren, können dann maximal von einem Thread abgearbeitet werden. Im Prinzip sind nach dieser Definition auch einfache Implementierungen über Locks möglich, z.B. mit Mutexen.

4.7 Kommunikation

Kommunikation ist zwischen Threads aufgrund des gemeinsamen Adressraums sehr leicht möglich. Eine einfache Kommunikation mehrerer Threads untereinander könnte mittels einer durch einen Mutex geschützten Variable erfolgen. Jeweils zum Schreiben oder Lesen wird der Mutex von einem Thread angefordert, und dann auf die Variable zugegriffen. Für viele Formen der Kommunikation, bei denen Daten zwischen Threads ausgetauscht werden, eignet sich diese einfache Form schon sehr gut, z.B. um den Status eines Threads zu signalisisieren. In vielen Fällen reicht eine Variable jedoch nicht. Man benötigt meist Ringpuffer, Queues oder Stacks.

Diese können natürlich auch durch Verwendung von Arrays oder Objekten mit synchronisierten Zugriffsmethoden realisiert werden. Der Synchronisationsaufwand bei einer threadsicheren Implementierung solcher Kommunikationsmechanismen ist jedoch nicht gerade gering. Außerdem können sich leicht Fehler durch inkorrekte Synchronisation in Form von Deadlocks oder Race-Conditions einschleichen. Da vom Benutzer selbst erstellte Kommunikationsmechanismen meist nur in begrenztem Rahmen angewandt werden, könnten diese Fehler aber lange Zeit unentdeckt bleiben. Es wäre daher wünschenswert, wenn ein Threading-System bereits ein breites Spektrum von Kommunikationsmechanismen mitbringen würde. Der Programmierer könnte sich dann mehr auf das eigentliche Programm konzentrieren und erhält aufgrund der größeren Entwicklergemeinde eine besser getestete Funktionaliät. Vor allem Threading-Systeme neueren Datums wie Java und .NET glänzen mit einer Vielzahl von Möglichkeiten in diesem Bereich.

Betrachten wir einmal ein typisches und sehr häufig auftretendes Erzeuger-Verbraucher Problem, bei dem ein oder mehrere Threads Daten erzeugen und ein oder mehrere Threads die Daten verwerten. Diese Art von Problem wird häufig mittels Message-Queues implementiert. Erzeuger stellen ihre Daten ans Ende der Schlange an und Verbraucher holen sich die Daten, die am Anfang der Schlange anstehen. Sind für einen Thread keine Daten mehr in der Queue, ist es meist wünschenswert, dass sich dieser Thread schlafen legt, bis wieder Daten von einem Erzeuger eingestellt wurden. Genau dieses Verhalten liefert Perl mit der threadsicheren “Queue“ Datenstruktur des “Thread“ Pakets (Thread::Queue), die nicht nur für die alten (5.005) Threads sondern auch für die neueren ithreads (5.6.0 Threads) verwendet werden kann. Nach der Erzeugung einer Queue kann man Werte mit Hilfe einer enqueue Methode in die Queue einfügen und mit dequeue holen. Ist die Queue leer, wartet ein Thread auf neue Daten. Alternativ kann auch die nicht blockierende dequeue_nb Methode verwendet werden. In dem Fall kann ein Thread, der leer ausgegangen ist, anstatt zu schlafen, etwas sinnvolles tun. In Java stellt sich die Situation etwas anders dar. Hier gibt es zwar Klassen des Packets java.util wie LinkedList, mit denen man eine Queue realisieren kann, allerdings ist diese standardmäßig nicht threadsicher. Java bietet jedoch für threadsichere Listen und andere Datenstrukturen des java.utils Pakets über die Klasse Collections Wrapper an. Verwendet man eine LinkedList als Queue, verhält sich diese jedoch immer noch nicht wie gewünscht. Ein Thread, der ein Element aus einer leeren Liste holen will (z.B. mit LinkedList.removeLast()), wird nicht schlafen gelegt, sondern erhält eine Exception. Das Schlafenlegen und Wecken des Threads muss durch den Programmierer selbst realisiert werden, was immer noch einen hohen Programmieraufwand erfordert. Mit Java 1.5 wurden diese “Missstände“ jedoch behoben. Die Klasse LinkedBlockingQueue des neuen java.util.concurrent Pakets ist threadsicher und blockiert einen Thread, der ein Element aus einer leeren Queue entnehmen will. Man kann sogar ein Timeout angeben, nach dem der blockierte Thread deblockiert wird. In Pthreads gibt es hingegen überhaupt keine Message-Queues.

Message-Queues lassen sich vor neben Erzeuger-Verbraucher Problemen auch gut für Pipelining einsetzen.

Win32-Threads bieten daneben sogenannte CompletionPorts. Hierbei können Threads auf Eingabe in eine Datei warten. Sobald Daten in die Datei geschrieben wurden, wacht der Thread

auf und kann die Daten bearbeiten.

In einigen Fällen kann es wichtig sein, dass zwei Threads zur gleichen Zeit (synchron) miteinander kommunizieren. Dies kann man sich so vorstellen, dass ein Thread, der mit einem anderen Thread kommunizieren will, zuerst wartet, bis der andere Thread bereit ist, mit ihm zu kommunizieren. Ist der andere Thread bereit werden die Kommunikationsdaten ausgetasucht. Danach könnnen beide Threads weiterrechnen. Ada unterstützt dies durch das sogenannte Rendez-vous Prinzip [Weg03, S. 89ff]. Da die Syntax nicht sehr verständlich ist, soll an dieser Stelle nur auf weiterführende Literatur verwiesen werden [ada06].

Für die Synchronisation von Threads mag es in vielen Fällen notwendig sein, dass ein oder mehrere Threads auf andere Threads warten müssen. Man könnte dies über Bedingungsvariablen realisieren. Einige Sprachen, darunter Pthreads und Java bieten hierzu Barrieren an. In Pthreads kann mit pthread_barrier_wait () darauf gewartet werden, dass eine angegebene Zahl von Threads die Barriere erreicht haben. Mit Erreichen der Barriere ist hiermit ein Aufruf von pthread_barrier_wait () gemeint, der das gleiche Barrieren-Objekt wie die bereits wartenden Threads verwendet.

Der Autor von [Lea97], der u.a. für die neuen Threading-Konstrukte in Java 5.0 verantwortlich ist, gibt in seinem Werk zahlreiche Design-Pattern zur Realisierung von Kommunikation in Java-Threads an, wie z.B. Futures. Viele dieser Pattern sind auch auf andere Programmiersprachen übertragbar. Die Programmierung mit Threads (zumindest in Java) wird hierdurch um einiges sauberer.

4.8 Thread-Cancelation und Thread-Suspension

In einigen Situationen ist es notwendig, Threads vorzeitig abzubrechen. In der Informatik gibt es einige Probleme, die teilweise nur durch Ausprobieren (trial and error) lösbar sind. Für eine einfache Wegesuche in einem Labyrinth könnte man z.B. an jeder Abzweigung für jeden möglichen Weg einen neuen Thread erzeugen und diesen prüfen lassen, ob der Weg zum Ziel führt. Nachdem ein Thread einen Weg durch das Labyrinth gefunden hat, wäre es Unsinn (solange nicht der kürzeste Weg gesucht wird), die weiteren Threads weiterrechnen zu lassen. Hier wäre ein Abbrechen der rechnenden Threads sinnvoll. Zu beachten ist, dass zur Lösung des Labyrinthproblems gemeinsame Datenstrukturen notwendig sind, z.B. das Labyrinth selbst oder Informationen, um den anderen Threads mitzuteilen, in welchen Abschnitten bereits gesucht wurde. Dies wird in Zusammenhang mit dem Thread Abbruch noch eine Rolle spielen. Eine ähnliche Situation tritt auch bei einigen mathematischen Problemen auf. Zum Auffinden von großen Primzahlen könnte man den zu untersuchenden Bereich in mehrere Bereiche einteilen, die unabhängig jeweils von einem Thread auf Primzahlen getestet werden können. Findet ein Thread die erste Primzahl, wird der Algorithmus abgebrochen. Auch bei Programmen, die Benutzer-Interaktionen erlauben, müssen häufig Threads abgebrochen werden. Z.B. wenn ein Benutzer eine Suche startet, die parallel von einem Thread abgearbeitet wird, um Eingaben des

Benutzers während der Suche zu erlauben. Bricht der Benutzer die Suche ab, läuft dies auf einen Abbruch des Threads hinaus.

Das Abbrechen von Threads wird von den meisten Threading-Systemen unterstützt. So bietet die Pthreads Bibliotheken eine pthread_cancel und Java eine Thread.stop() Methode, die jedoch als deprecated markiert und nicht mehr verwendet werden sollte. Betrachtet man die Methode Thread.stop() und den Grund dafür [jav05], dass man diese Methode nicht mehr verwenden sollte etwas genauer, bemerkt man schnell, dass das Abbrechen von Threads doch nicht so trivial ist, wie man es sich denken mag. Der Haken an der Verwendung der Methode liegt darin, dass alle Monitore, die von einem mit Thread.stop abgebrochenem Thread gesperrt wurden, sofort freigegeben werden, ohne dass der Thread die Objekte, die durch den Monitor geschützt werden, in einen konsistenten Zustand überführen kann. Greift zu einem späteren Zeitpunkt ein weiterer Thread auf ein solches "zerstörtes" Objekt zu, kann dies zu unvorhergesehen Fehlern führen. Eine Abhilfe, außer ganz auf Thread.stop() zu verzichten, ist nach [jav05] nicht möglich. Hier wird stattdessen ein "sanftes" Abbrechen empfohlen. Will ein Thread einen anderen Thread abbrechen, so setzt er eine durch einen Monitor geschützte Variable. Der abzubrechende Thread liest in regelmäßigen Abschnitten diese Variable und und beendet sich, wenn dies durch den Variableninhalt signalisiert wird. Dies setzt jedoch die Kooperation der Threads vorraus, d.h. der abzubrechende Thread muss die Abbruchvariable erstens lesen und zweitens auch darauf reagieren. Eine interessante Möglichkeit zum Abbruch von Threads auf diese kann in Java auch mit Hilfe der Thread.interrupt() Methode geschehen. Ein Thread könnte diese Methode auf dem Thread-Objekt des abzubrechenden Threads aufrufen, um diesen Thread einen Hinweis zu geben, dass dieser abbrechen soll. Der Thread prüft mittels Thread.isInterrupted(), ob er abbrechen soll. Wenn ja, tut er dies, sonst arbeitet er wie gewohnt weiter. Das besondere an der Methode Thread.interrupted() ist, dass selbst Threads, die sich schlafengelegt haben Thread.sleep(), hierdurch im Schlaf (mit einer Exception) unterbrochen werden und sich beenden können. Eine Schattenseite hat jedoch auch dieses Verfahren. Threads, die abgebrochen werden sollen, müssen auf die Möglichkeit vorbereitet sein, dass sie während dem Schlaf aufgeweckt werden können. Zwar müssen Exceptions der Ausprägung InterruptedException immer abgefangen werden. Allerdings werden diese Exceptions nicht geeignet behandelt, wodurch es durch die vorzeitige Beendigung des Schlafes zu Fehlern kommen kann.

Die Frage ist nun, ob Thread-Cancelation generell in allen Threading-Systemen vermieden werden sollte. Thread-Cancelation ist in den Threading-Systemen jedoch sehr unterschiedlich implementiert. Im Prinzip kann man zwischen folgenden Implementierungen unterscheiden:

1. Sofortiger Abbruch
2. Verschobener Abbruch (Deferred Cancelation) mittels Cancelation Points
3. Kooperativer Abbruch (Hinweis an abzubrechenden Thread)

Bei dem ersten Punkt werden Threads sofort abgebrochen, egal an welcher Stelle sie sich

befinden. Das kann zu Deadlocks oder anderen Problemen führen, wenn der abzubrechende Thread gerade im Besitz von Locks oder einer critical section im Allgemeinen ist.

4.9 Threadsicherheit

Der Begriff Threadsicherheit ist schon des öfteren gefallen und bezeichnet eine Eigenschaft einer Bibliothek oder Bibliotheksfunktion. Ist eine solche threadsicher (engl.: thread-safe), bedeutet das, dass sie gleichzeitig von mehreren Threads aufgerufen werden kann, ohne das es zu Race-Conditions kommt. Dies könnte z.B. der Fall sein, wenn z.B. Variablen in einer Funktion, die von zwei Threads gleichzeitig manipuliert werden, nicht geschützt werden. Neben den beiden Möglichkeiten threadsicher und nicht threadsicher wird die Threadsicherheit in [LB98, S. 200] insgesamt sechs Kategorien eingeteilt Begriff. Darunter ist z.B. Multithreading hot (MT hot), für Funktionen, die sicher sind und eine schnelle Implementierung besitzen. Diese Begriffe werden aber in den Dokumentationen der Bibliotheken normalerweise nicht benutzt. Bei der Programmierung mit Threads ist es sehr wichtig, zu wissen, ob eine verwendete Bibliothek threadsicher ist oder nicht. Ist eine Bibliothek nicht threadsicher, sollte man den Einsatz vermeiden und eine Alternative suchen. Ist der Einsatz der Bibliothek unumgänglich, so müssen alle Aufrufe der Bibliotheksfunktionen durch ein gemeinsames Lock gesperrt werden. Weiß man, dass die Funktionen der Bibliothek nicht die gleichen Variablen untereinander verwenden und nur der Aufruf der gleichen Funktion durch mehreren Threads kritisch ist, genügt es auch, jede Funktion durch ein eigenes Lock zu schützen. Normalerweise wird in der Dokumentation der Bibliothek angegeben, ob sie oder eine ihrer Funktionen threadsicher ist oder nicht. Ist keine Angabe vorhanden, sollte man zur Sicherheit annehmen, dass sie es nicht ist. Das gilt selbst dann, wenn man den Quelltext kennt, da dieser sich in einer anderen Implementierung ändern kann. Wie erwähnt kann eine ganze Bibliothek threadsicher sein oder nur einzelne Funktionen hiervon. Selbst wenn ein Autor weiß, dass seine Bibliothek für Multithreading verwendet werden könnte, gibt es Gründe, die Bibliothek trotzdem nicht threadsicher zu gestalten. So ist neben einem besseren Design auch meist das zeitaufwendige Sperren kritischer Bereiche notwendig ist, um Threadsicherheit zu gewährleisten. Die Ausführungszeit kann demnach gesteigert werden, wenn auf Threadsicherheit verzichtet wird.

Im Rahmen dieser Arbeit ist vor allem interessant, ob die Standardbibliothek eines Threading-Systems threadsicher ist. In Java ist dies z.B. nicht der Fall. Auf Datenstrukturen des java.util Pakets wie Listen, Bäume und Hashes kann nicht von mehreren Threads gleichzeitig zugegriffen werden, dies wird in der JavaDoc zu Java 1.4 deutlich hervorgehoben. Es wird dort vorgeschlagen, entweder selbst für die Synchronisation zu sorgen oder die Wrapper der Collections-Klasse (statische Methoden der Form Collections.synchronized...()) zu verwenden, wobei die Wrapper wohl die sauberere Variante darstellen dürften. In Perl sind ebenfalls nicht alle Bibliotheken threadsicher. In der Dokumentation wird jedoch im Normalfall angegeben, ob eine Bibliotheksfunktion threadsicher ist oder nicht. Hier kommt jedoch noch erschwerend hinzu, dass Perl als interpretierte Sprache einen Interpreter benötigt. Dieser dürfte fast immer in C

geschrieben sein und verwendet folglich die Funktionen der C-Bibliothek des Betriebssystems. Sind diese schon nicht threadsicher, so kann es eine Perl-Funktion, die auf dieser Funktionalität aufbaut ohne entsprechende Gegenmaßnahmen evtl. auch nicht sein. Somit ist es möglich, dass eine Perl-Funktion unter einer Laufzeitumgebung threadsicher ist und unter einer anderen nicht und zur RaceCondition führt (vgl. [per05]). Im allgemeinen ist in den meisten Fällen auch eine sichere Verwendung von nicht explizit threadsicheren Bibliotheken möglich. Der Grund dafür ist, dass zumindest bei der Verwendung von ithreads standardmäßig kein gemeinsamer Adressraum verwendet wird, wodurch die Wahrscheinlichkeit, dass sich zwei Threads gegenseitig stören, sehr gering wird [per05]. Bei den Pthreads werden die meisten Bibliotheken, die man für ein Programm benötigt, mit hoher Wahrscheinlichkeit threadsicher sein. Das liegt daran, dass der POSIX-Standard nicht nur die Pthreads, sondern auch die meist gebräuchlichsten Funktionen wie printf(), scanf() und sogar Systemaufrufe wie read(), write() und usleep() (alle definiert in unistd.h) definiert. Laut Standard müssen die meisten der in POSIX definierten Funktionen threadsicher sein (vgl. [NBF98]). Für einige nicht threadsichere Funktionen wie localtime() gibt es alternative threadsichere Varianten, die mit einem abschließendem "_r" gekennzeichnet werden (localtime_r ()). Andere Funktionen besitzen neben der Standardvariante eine ungeschützte Variante, die performanter ist. Sie werden mit "_unlocked" gekennzeichnet, z.B. getc_unlocked () (vgl. [NBF98]).

4.10 Thread Local Storage und Thread Specific Data

Multithreading zeichnet sich vor allem dadurch aus, dass die Threads eines Prozesses einen gemeinsamen Adressraum besitzen. Der positive Aspekt hierbei ist, dass die Threads auf einfache Weise über den gemeinsamen Speicher, z.B. eine globale Variable Kommunizieren können. Der Nachteil ist jedoch, dass ein Thread keine globalen Variablen anlegen kann, die nur für ihn sichtbar ist. In einigen Fällen kann so etwas sinnvoll sein. So wird bei der Programmierung mit C meist die in ISO-C und POSIX definierte globale Variable "errno" ausgelesen, um festzustellen, ob bei einem zuvor aufgerufenen Systemaufruf ein Fehler aufgetreten ist. Bei der Verwendung von Pthreads kommt es aber normalerweise vor, dass mehrere Threads auf diese Variable zugreifen. Betrachten wir zwei Threads A und B. Thread A hat einen fehlgeschlagenen Systemaufruf ausgeführt und Thread B einen Systemaufruf, der erfolgreich beendet wurde. Beide Threads lesen nun errno. Nun kann es passieren, dass einer oder sogar beide ein falsches Ergebnis lesen, da beide beim Ausführen des Systemaufrufs den Wert von errno gegenseitig überschrieben haben. Abhilfe schafft hier TSD (Thread-Specific Data), wie es in Pthreads heißt. TSD wird sogar nach Aussage von [LB98, S. 130ff] automatisch für die Variable errno angewendet, ohne das der Programmierer hiervon etwas mitbekommt. Die Variable errno wird also nicht wirklich von den Threads geteilt, sondern jeder Thread greift auf seine eigene lokale Variable zu, obwohl der Zugriff scheinbar auf eine globale Variable erfolgt. Wie aber funktioniert TSD? Bleiben wir zunächst bei den Pthreads. Hier kann man TSD Datensätze erzeugen, indem man zuerst mit einer Funktion namens pthread_key_create () einen Schlüssel erzeugt. Danach

kann man mit pthread_setspecific () und pthread_getspecific () auf den "Inhalt" des Schlüssels zugreifen. Jeder Thread bekommt hierbei, obwohl er auf den selben Schlüssel zugreift unterschiedliche Daten zurückgeliefert.

```
1  #include <pthread.h>

3  pthread_key_t key;

5  void work() {
6      /* Fortlaufende Nummer des Threads aus TSD-Bereich */
7      int threadID = (int)pthread_getspecific(key);
8      /* Arbeiten */
9      switch(threadID) { ... }
10 }

12 void* thread_start(void* arg) {
13     /* TSD-Bereich auf fortlaufende Nummer setzen */
14     pthread_setspecific(key, arg);
15     work();
16 }

18 int main() {
19     pthread_t thread1, thread2;

21     /* Erstelle einen Schlüssel für TSD */
22     pthread_key_create(&key, NULL);

24     /* Erzeuge zwei Threads mit fortlaufender Nummer */
25     pthread_create(&thread1, NULL, thread_start, (void*)1);
26     pthread_create(&thread2, NULL, thread_start, (void*)2);
27     ...
28     /* Lösche TSD-Schlüssel */
29     pthread_key_delete(key);
30     ...
31 }
```

Nicht nur Pthreads, sondern auch viele andere Threading-Systeme bieten TSD. In Win32 wird dieses Konzept Thread Local Storage (TLS) genannt, funktioniert im Prinzip aber genauso. Mit TlsAlloc() kann man einen Schlüssel erzeugen, der den Keys in Pthreads entspricht. Werte für diesen Index können mit TlsSetValue() und TlsGetValue() unter Angabe des Schlüssels analog zu den Pthreads gesetzt oder gelesen werden. Win32 bietet hier jedoch eine Besonderheit und zwar die Unterstützung von TLS durch den Compiler. Schreibt man das Attribut "__declspec(thread)" vor eine globale Variable, wird diese für jeden Thread in einen getrennten Adressraum abgelegt. Die Variable ist somit quasi lokal, wie bei den bereits beschriebenen

TLS-Daten auch. Jeder Thread hat eine unterschiedliche Sicht auf die Variable (vgl. [msd03f]).

Beispiel: __declspec(thread) **int** tlsData ;

Selbst .NET bietet eine Form von TLS, indem man mit Thread.AllocateNamedDataSlot() einen threadspezifischen Datenslot anlegen kann, in dem die Daten abgelegt werden können. Es ist jedoch fraglich, ob TLS, bzw. TSD in objektorientierten Sprachen überhaupt noch gebraucht werden, da Threads ohnehin über Objekte definiert werden, für die man ganz einfach lokale Variablen anlegen kann, indem man sie als private deklariert. Die Perl ithreads benötigen kein TSD, da die Daten ohnehin standardmäßig nicht geteilt werden. Hier müssen die Daten explizit geteilt werden, also eine Art Thread Global Storage (TGS) auch wenn es diesen Begriff nicht gibt.

4.11 Scheduling

Das Umschalten der Threads übernimmt der Scheduler. Das kann entweder der System-Scheduler im Falle von Kernel-Threads oder für User-Level Threads ein spezieller Scheduler, der mit der Bibliothek mitgeliefert wird. Der Scheduler sorgt im allgemeinen dafür, dass alle Threads einen gleichermaßen langen Zeitschlitz zur Berechnung auf der CPU bekommen. Dies muss aber nicht sein. Wir im Abschnitte 3.4 erwähnt, sind Fibers zum Beispiel non-preemptive. Die Threads sind somit selbst die Scheduler. Auch wenn es einen Scheduler gibt, muss dieser nicht alle Threads gleich behandeln. Nehmen wir einmal hypothetisch an, die gesamte Boardelektronik eines Flugzeuges würde über ein Programm mit mehrere Threads gesteuert. Sowohl die Flugsteuerung als auch die Unterhaltungselektronik werden jeweils durch einen Thread realisiert. In einem ungünstigen Moment startet ein Passagier einen Film und blockiert damit das gesamte System, der Thread für die Flugsteuerung, der das Flugzeug lenkt, kommt nicht zur Ausführung und das Flugzeug kommt außer Kontrolle. Ehrlich gesagt, ein ziemlich unrealistisches Beispiel. Es zeigt jedoch sehr gut die Problemantik. So müsste die Flugsteuerung höchste Priorität und die Unterhaltungselektronik die geringste besitzen. Die Flugsteuerung muss sogar innerhalb kürzester Abstände, am besten sofort, reagieren können. Dies nennt sich Echtzeitanforderung. In fast allen Threading-Systeme können die Prioritäten der Threads unabhängig voneinander geändert werden. So bietet Java mit Thread. setPriority () die Möglichkeit, die Priorität eines Threads zwischen den Werten Thread.MIN_PRIORITY (in Java 1.4.2 ist dies der Wert 1) und Thread.MAX_PRIORITY (in Java 1.4.2 der Wert 10) zu wählen. Threads besitzen standardmäßig die Priorität Thread.NORM_PRIORITY (in Java 1.4.2 der Wert 5). Nach [Oec01, S. 152] kann kein direkter Bezug zwischen Priorität und Bevorzugung durch den Scheduler hergestellt werden. Man kann lediglich sagen, dass ein Thread mit höherer Priorität tendenziell öfter vom Scheduler Rechenzeit zugewiesen bekommt. In der zuvor angegebenen Quellen wird die Faustregel angegeben, E/A-intensive Threads mit hohen Prioritäten zu versehem und rechenintensive Threads mit niedrigen. Hierdurch soll gewährleistet sein, dass sofort auf E/A-Operationen reagiert werden kann. Dies dürfte vor allem bei Programmen mit Benutzerinteraktivität stimmen.

Reagiert das Programm schnell auf die Eingaben des Benutzer, empfindet er das Programm meistens schneller, auch wenn die Berechnungen im Hintergrund noch nicht bereit sind. In Bezug auf Java ist dies schon alles, was im Bereich des Schedulings vom Programmierer getan werden kann.

Kommen wir nun zu den Pthreads. Auch hier können die Prioritäten eingestellt werden und zwar über Thread-Attribute, die mit den Funktionen pthread_attr_setscope (), pthread_attr_setschedpolicy () und pthread_attr_setschedparam () gesetzt werden können. Der Geltungsbereich (engl.: scope) gibt an, ob die Threads relativ zu anderen Threads im Prozess oder zu Threads des gesamten Systems geschedult werden. LinuxThreads unterstützen laut man-Page lediglich den System-Geltungsbereich, im Falle von NPTL geben die man-Pages keine Auskunft. Der Einfluss des Scopes auf das Scheduling ist zu komplex, um auf die Einzelheiten eingehen zu können, vor allem wenn mehrere Prozessoren zu Verfügung stehen. Deshalb soll an dieser Stelle auf [NBF98, S.144 ff] verwiesen werden. Mit pthread_attr_setschedpolicy () kann für jeden Thread die vom Scheduler zu verwendende Scheduling-Strategie gewählt werden. Zur Auswahl stehen Round-Robin (SCHED_RR), FIFO-Scheduling (SCHED_FIFO) und eine nicht weiter spezifizierte Strategie (SCHED_OTHER). Die Standardeinstellung ist SCHED_OTHER, die im Gegensatz zu den anderen beiden normalerweise eine Strategie ohne Echtzeit-Unterstützung ist. In einigen Implementierungen kann es jedoch auch sein, das SCHED_OTHER gleich einer der beiden anderen Strategie ist. Normalerweise unterstützt SCHED_OTHER auch Prioritäten. Wie viele Prioritätsstufen dies aber sind, ist jedoch Implementierungsabhängig.
Nun zu den Echzeit-Strategien. Man sollte beachten, das diese nur genutzt werden können, wenn das Programm mit Root-Rechten ausgeführt wird. Jeder Echzeit-Thread kann eine von mindestens 32 Prioritäten annehmen (vgl. [NBF98, S 153]. Für jede dieser Prioritäten existiert eine eigene Warteschlange. FIFO- und RR-Threads teilen sich jedoch die Warteschlangen gleicher Priorität. Muss ein neuer Thread ausgesucht werden, so wählt der Scheduler den Thread mit der höchsten Priorität, der sich am Anfang der Schlange befindet. Dies kann sowohl ein FIFO- oder RR-Thread sein. Erst wenn kein Real-Time Thread vorhanden ist, wird ein regulärer (nicht Real-Time) Thread gewählt. Bei der FIFO-Strategie darf ein Thread so lange laufen, bis er blockiert oder selbst die Kontrolle abgibt. Einen preemptiven Abbruch des Threads von Seiten des Scheduler gibt es nicht. Gibt ein Thread die Kontrolle ab, wird er ans Ende der Queue seiner Priorität gestellt. Hat ein Thread z.B. wegen E/A-Operationen blockiert muss er hierzu jedoch erst wieder deblockieren. Bei der Round-Robin Strategie bekommt jeder Thread einen Zeitfenster zugeteilt, in dem er rechnen darf. Ist das Zeitfenster abgelaufen, bricht der Scheduler den Thread ab und stellt ihn ans Ende der Queue mit der Priorität des Threads. Das gleiche passiert, wenn der Thread blockiert. Sobald ein Real-Time Thread egal welcher Ausprägung mit höherer Priorität lauffähig wird, wird der momentan laufende Thread abgebrochen. Wie man sieht, kann man mit Echtzeit-Threads sehr schnell auf Ereignisse reagieren, da die Threads sehr schnell hintereinander und – zumindest bei den FIFO-Threads – für einen nahezu beliebigen Zeitraum auf einer CPU laufen dürfen. In [NBF98, S. 151] wird empfohlen, Threads eines zeitkritischen Programmes, die in einer bestimmten Zeitspanne ausgeführt

werden müssen, als FIFO-Threads mit hoher Priorität zu realisieren und die anderen Threads des Programmes mit etwas geringeren Anforderungen als Round-Robin-Threads zu erzeugen. So faszinierend Echtzeit-Threading auch ist, im realen Programmieralltag wird man sie kaum brauchen. Es sei denn, man erstellt zeitkritische Treiber oder Flugzeugsteuerungen. In [LB98, S. 78] wird davon abgeraten Echtzeit-Threads zu verwenden, wenn man sie nicht wirklich braucht. So könnten zum Beispiel alle Bereiche der Benutzerinteraktion und der Multimediaanwendungen auch ohne Echtzeit-Threads bedient werden. Das große Problem in der Verwendung von Echtzeit-Threads liegt darin, dass die Gefahr zu groß ist, die Prioritäten zu hoch anzusetzen und damit das gesamte System instabiel zu machen. Die ersten Versuche des Autors dieser Arbeit im Bereich der Echtzeit-Threads endete mit einem eingefrorenen Linux-System und einem Zwangsneustart. Selbst winzige Fehler im Quelltext können bereits den Absturz oder das Aushungern des Systems verursachen. Nicht ohne Grund sind Root-Rechte zur Ausführung nötig.

In Win32 funktioniert das Scheduling ähnlich wie bei den Pthreads. Es gibt 32 Prioritätsstufen, deren Threads jeweils als gleichwertig bezüglich des Schedulings angesehen werden (vgl. [msd03e]). Im Round-Robin werden die Zeitschlitze auf die Threads aufgeteilt, wobei wie bei den Echtzeit-Pthreads immer erst die Threads mit der höchsten Priorität geschedult werden. Läuft ein Thread mit niedriger Priorität, während ein höher priorisierter Thread lauffähig wird, wird der Thread mit der niedrigeren Priorität unterbrochen und der mit der höheren Priorität auf die CPU geschedult. Interessant ist bei den Win32-Threads vor allem die Berechnung der Prioritäten der Threads, die sich aus der Prioritätsklasse (von der IDLE_PRIORITY_CLASS bis zur REALTIME_PRIORITY_CLASS) des Prozesses und der Threadpriorität (von THREAD_PRIORITY_IDLE bis THREAD_PRIORITY_TIME_CRITICAL) zusammensetzen und eine der 32 Prioritätsstufen bilden.

Es gibt noch viel mehr Bereiche beim Multithreading, in denen Scheduling eine Rolle spielt. So bei dem Mapping der User-Level auf die Kernel-Level – soweit vorhanden. In diesem Bereich sind vor allem die LWPs der Solaris-Threads interessant, die ein sehr ungewöhnliches zweistufiges M:N Threading-Modell nutzen. Ebenfalls in diesem Kontext sollten zumindest kurz die Begriffe Scheduler Activations und Sleeping Threads erwähnt werden, die ein frühzeitiges Blockieren der User-Level Threads verhindern sollen. Aus Platzgründen muss aber hierzu auf weitergehende Literatur hingewiesen werden, so z.B. [LB98].

4.12 Speichermodelle

Jedes Threading-System hat sein eigenes Speichermodell. Hiermit ist vor allem gemeint, wie der Zugriff auf gemeinsame Variablen realisiert wird. Hierbei müssen die Begriffe der Atomizität und der Sichtbarkeit unterschieden werden. Atomizität bedeutet hierbei, dass ein Zugriff lesender oder schreibender Art ununterbrechbar (atomisch) ausgeführt wird. Sind Zugriffe im Gegensatz dazu nicht atomisch, kann folgendes Szenario vorkommen: Ein Thread versucht auf eine Variable lesend zuzugreifen, die ein anderer Thread gerade beschreibt. Wird dann ein

Kontext-Wechsel während dem Schreibvorgang ausgeführt, liest der Leser einen ungültigen Wert. Noch schlimmer wird es, wenn beide Threads gleichzeitig schreinen wollen. Bei atomischen Zugriffen auf Variablen, kann eine Unterbrechung während dem Lesen oder Schreiben nicht vorkommen. Der Begriff der Sichtbarkeit sagt hingegen etwas anderes aus. Bedingt durch die Speicherhierarchie moderner Rechnerarchitekturen werden Daten meist in Speichern mehrerer Speicherebenen gehalten. So zum Beispiel im Hauptspeicher, in einem oder mehreren Caches und Registern. Ändert die CPU einen Wert im Register oder Cache und schreibt ihn nicht sofort zurück in den Hauptspeicher, dürfen die Daten im Hauptspeicher nicht verwendet werden. Das ist vor allem bei Multiprozessorrechner ein Problem. Hier könnte eine CPU lesend auf ein Datum im Hauptspeicher zugreifen wollen, das zwar von einer anderen CPU geändert aber noch nicht in den Hauptspeicher zurückgeschrieben wurde. Die CPU liest dann den veralteten Wert und eine Race-Condition entsteht. Diese Fehler sind vor allem deswegen teuflisch, da sie auf einem Ein-Prozessor-System nicht auftreten können und erst auf Multiprozessor-Systemen zuschlagen. Um diese Situation zu vermeiden werden zum einen intelligentere Speicheranbindungen auf Hardwareebene eingesetzt (siehe [RR00]). Zum anderen kann dieses Problem auf Softwareebene behoben werden, wenn vor dem Zugriff auf gemeinsame Daten ein sogenannter Memory-Flush durchgeführt werden. Hierbei schreiben die Prozessoren ihre in den Caches gehaltenen veränderten Daten zurück, wodurch Inkonsistenzen vermieden werden können. Viele Threading-Systeme rufen ein solches Flush implzit auf, sobald ein Thread einen Mutex, eine CriticalSection oder ein ähnliches Konstrukt zum Zugriff auf gemeinsamen Speicher sperrt. Andere Threading-Systeme tun dies nicht.

Betrachten wir zunächst .NET und beginnen mit einem stark vereinfachtes Beispiel, dass in ähnlicher Form bereits in [yod06] verwendet wurde. Dabei soll Thread1 einen anderen Thread2 durch setzen eines Flags beenden.

```
1   ...
2   static bool abort;

4   static void Thread1()
5   {
6       abort = true;
7   }

9   static void Thread2()
10  {
11      while(true) {
12          if(abort) break;
13      }
14  }
```

Nach [yod06] könnte es in diesem Beispiel vorkommen, dass Thread2 niemals abbricht. Das liegt daran, dass Thread2 auf einer veralteten Kopie von “abort“ weiterarbeitet. Dies lässt sich jedoch durch Einsatz eines Monitors-Objekts oder eines darauf basierenden lock-Blocks

vermeiden. Beim Eintritt in einen lock-Block bzw. Monitor werden die Daten durch die CPUs zurückgeschrieben und die Werte der Variablen neu eingelesen. Beim Verlassen des Blocks bzw. Monitors werden die Daten wieder zurückgeschrieben. Die Daten, die innerhalb eines solchen Blocks bzw. Monitors gelesen werden, müssen demnach konsistent sein. Alternativ können in .NET Variablen einiger primitiver Datentypen als volatile deklariert werden. Dadurch werden sie vor jedem Lesevorgang neu eingelesen und der Wert der Variablen nach jedem Schreibvorgang in den Hauptspeicher herausgeschrieben [yod06]. Bleiben wir erst einmal beim .NET-Speichermodell. Dieses garantiert nämlich atomische Lese- und Schreibzugriffe auf Datentypen, die die Größe eines int-Werts nicht überschreiten. So können z.B. von mehreren Threads gemeinsam verwendete int-Werte ohne Schutz durch ein Lock verändert werden. Allerdings sollte man die Variablen dann als volatile deklarieren, damit es nicht aus den zuvor genannten Gründen zu einer Race-Condition kommt.

```
1   ...
2   static bool abort;
3   static object abortLock = new object();

5   static void Thread1()
6   {
7   lock(abortLock) {
8           abort = true;
9       }
10  }

12  static void Thread2()
13  {
14      while(true) {
15          lock(abortLock) {
16              if(abort) break;
17          }
18      }
19  }
```

Das Java Speicher-Modell hat einige Ähnlichkeiten mit dem des .NET-Frameworks. So sind Zugriffe auf primitive Datentypen außer double und long, sowie der Zugriff auf Referenzen atomisch. Allerdings nützt dies nicht sehr viel, da keine Garantie gegeben wird, das die gelesenen Werte auch dem zuletzt geschriebenem Wert entsprechen. Deklariert man Variablen primitiver Typen jedoch als volatile, so hat das den selben Effekt, als wären die Zugriffe auf die Variablen durch Locks synchronisiert worden.

Speichermodelle gehen weit über das hinaus, was im Rahmen dieser Arbeit präsentiert werden kann. So wird z.B. in Java auch festgelegt, an welchen Stellen ein Compiler optimieren und dazu den Code umstellen darf und an welchen nicht. Dies ist wichtig, denn es konnte hierbei dazu kommen, dass ein vom Programmierer einwandfrei programmierter Quelltext in

inkorrekten Code umgewandelt wird. Dies könnte im Bereich des Multithreading z.B. ein Lock betreffen, dessen Sperrungsfunktion vom Compiler an eine andere Position verschoben wird. Aus Perfomance-Sicht mag die Entscheidung zwar günstig sein, in Hinblick auf die Korrektheit des Programmes jedoch ein fataler Irrtum. Speichermodelle werden auch deshalb entwickelt, um den Programmierer vor solchen Compiler-Irrtümern zu schützen. Allerdings sind die meisten Speichermodelle so komplex und für Aussenstehende schwer zu verstehen, dass ein tieferer Einblick in konkrete Speichermodelle ausbleiben soll.

Kapitel 5

Zusammenfassung und Ausblick

Im Laufe der Arbeit wurden die Unterschiede zwischen den Threading-Systemen aufgezeigt und miteinander verglichen. Dabei fällt auf, dass es auf vielen Gebieten wie Thread-Erzeugung und Thread-Synchronisation große Parallelen zwischen den Threading-Systemen gibt. So scheint die Erzeugung von Threads zum einen mittels einer thread_create ()-Funktion in nicht objektorientierten Sprachen und zum anderen durch Instanziierung eines Objekts einer Thread-Klasse in objektorientierten Sprachen ein ungeschriebener Standard. Das gleiche gilt für die Synchronisation mittels Mutexen oder Semaphoren. In anderen Gebieten fallen die Unterschiede hingegen sehr viel extremer aus, z.B. bei der Thread-Cancelation oder dem Funktionsumfang an komplexeren Datenstrukturen wie Task-Pools oder Message-Queues. Betrachtet man die Standardbibliotheken neuerer Programmiersysteme wie Java und .NET, fällt auf, dass Multi-Threading ein fester Standard geworden ist und die Notwendigkeit, eine externe Threading-Bibliothek einbinden zu müssen, entfällt. Außerdem geht der Trend zu immer umfangreicheren Threading-Bibliotheken. Früher waren lediglich Funktionen verfügbar, um einen Thread zu erzeugen und ihn mit Mutexen und ‚wenn man Glück hatte, Bedingungsvariablen zu synchronisieren. Leser-/Schreiber-Locks oder Task-Queues mussten selbst implementiert werden. Diese Arbeit entfällt in neueren Systemen, da diese Features bereits in das Threading-System implementiert werden. Der Programmierer kann sich mehr auf das eigentliche Aufgabe konzentrieren und hat keine Probleme mehr mit fehlerhaften eigenen Implementierungen dieser Datenstrukturen. Dieser Trend kann vor allem an den neuen Versionen von .NET und Java aber auch der stetig wachsenden Pthreads-Bibliothek festgemacht werden. Auf der anderen Seite hat dies zur Folge, dass Threading-Bibliotheken immer weiter aufblähen und unübersichtlicher in der Handhabung werden. Der zunehmende Einsatz von virtuellen Maschinen und Skriptsprachen sorgt zwar für Portabilität und schnelle Enwicklungszeiten, hat jedoch den Nachteil langsamerer Ausführungszeiten. Da an der Ausführungsgeschwindigkeit von interpretierten Code selbst nicht viel geändert werden kann, ist es mit Multi-Threading möglich, diese Geschwindigkeitsdefizite durch geeignete Parallelisierung zumindest etwas zu kompensieren. Auch die in ihrer Zahl immer weiter zunehmenden Multi-Core Systeme machen Multi-Threading attraktiver. Im Prinzip unterstützen bereits alle der in dieser Arbeit untersuchten Threading-Systeme Multiprozessor Architekturen.

Im Ergebnis aus dem Vergleich der Threading-Systeme kann man jedoch keinen eindeutigen Gewinner feststellen. Jedes System hat Stärken und Schwächen. So wird man in Java wahrscheinlich nie Programme mit Echtzeitanforderungen schreiben können, solange der Code von einer virtuellen Maschine ausgeführt wird. Anders sieht dies natürlich aus, wenn man Java-Programme in nativen Code umwandelt oder auf Architekturen laufen lässt, die die JVM in Hardware implementieren, wie dies bei einigen Embedded Systemen bereits der Fall ist. Umgedreht werden Pthreads nie vollkommen portabel sein. Selbst wenn es einen Gewinner unter den Threading-Systemen gäbe, wäre damit noch lange nicht gesagt, dass man dieses Threading-System in einem konkreten Fall auch verwenden kann. So sind viele Threading-Systeme an spezielle Umgebungen gebunden, wie die Win32-Threads an Windows oder Java-Threads an die Programmierung mit Java. Die Bibliotheken können meist auch nicht einfach ausgetauscht werden. Auch in Zukunft wird es wohl keine Bibliothek geben, die allen Ansprüchen gerecht werden kann. Man kann jedoch von den anderen Threading-Systemen lernen und die ein oder andere Funktionalität, die sich in einem Threading-System als nützlich erwiesen hat, übernehmen.

Wie wird es im Threading-Bereich weitergehen? Die meisten wissenschaftlichen Arbeiten über Multi-Threading wurden bereits in den 80er Jahren abgeschlossen. Heutige Arbeiten beschäftigen sich vor allem mit Scheduling-Verfahren. Schreibt man den Trend der letzten Jahre fort, wird wohl vor allem immer mehr Funktionalität in Form von vielseitig einsetzbaren Datenstrukturen, wie es mit den Task-Pools und Leser-/Schreiber-Locks bereits der Fall war, in die Bibliotheken integriert. Damit könnte die Produktivität erhöht und die Fehlerrate gesenkt werden. Denkbar wäre auch, dass sich ein Design-Pattern ähnlicher Ansatz durchsetzen könnte, wie es zum Beispiel bei dem parallelen Programmiersystem COPS der Fall ist. Allerdings steckt dieser Zweig momentan noch in den Kinderschuhen. Reale Programme lassen sich nur sehr schwer in die Form eines Pattern bringen. Der Speedup fällt beim Einsatz Pattern-basierter Systeme im Gegensatz zu manuell parallelisiertem Code auch meist sehr mager aus.

Literaturverzeichnis

[ada06] *Die Programmiersprache Ada*. http://ada-deutschland.de, März 2006.

[aus06] *POSIX FAQ der OpenGroup*. http://www.opengroup.org/austin/papers/posix_faq.html, Februar 2006.

[boo06] *Boost-Threads*. http://www.boost.org/doc/html/threads.html, März 2006.

[Doe] DOEBEN. http://www.fbmnd.fh-frankfurt.de/~doeben/I-RT/I-RT-TH/VL6/i-rt-vl-vl6.html. POSIX-Konformität.

[Dre05] DREPPER, ULRICH: *Design of the New GNU Thread Library*. http://people.redhat.com/~drepper/glibcthreads.html, Dezember 2005. Diskussion zu Threading-Modellen durch Ulrich Drepper.

[Fly72] FLYNN, M.J.: *Some Computer Organizations and their Effectiveness*. 1972.

[Han81] HANSEN, PER BRINCH: *Edison: A Multiprocessor Language*. Software - Practice and Experience, Seiten 325–361, 1981.

[Hoa74] HOARE, C. A. R.: *Monitors: An Operating System Structuring Concept*. Communications of the ACM, 17(10):549–557, Oktober 1974. http://www.acm.org/classics/feb96.

[int] *Intels Prognose zur Entwicklung von Multiprozessorsystemen*.

[jav05] *Plattformunabhängigkeit*. http://java.sun.com/j2se/1.5.0/docs/guide/misc/threadPrimitiveDeprecation.html, Dezember 2005.

[Krü01] KRÜGER, GUIDO: *GoTo Java 2*. Addison-Wesley, 2. Auflage, 2001.

[LB98] LEWIS, BIL und DANIEL J. BERG: *Multithreaded Programming with Pthreads*. Sun Microsystems Press, 1998.

[Lea97] LEA, DOUG: *Concurrent Programming in Java: Design Principles and Pattern*. Addison Wesley, 1997.

[MK00] MAGEE, JEFF und JEFF KRAMER: *Concurrency: state models and Java programs*. John Wiley and Sons, 2000.

[Moo65] MOORE, GORDON E.: *Cramming more components onto integrated circuits*. ftp://download.intel.com/research/silicon/moorespaper.pdf, 1965.

[msd03a] *Microsoft MSDN: Bibliotheksfunktionen zur Thread-Kontrolle*. ms-help://MS.MSDNQTR.2003FEB.1031/vccore/html/_core_C_Run.2d.Time_Library_Functions_for_Thread_Control.htm, Februar 2003.

[msd03b] *Microsoft MSDN: Bibliotheksunterstüzung für Multithreading*. ms-help://MS.MSDNQTR.2003FEB.1031/vccore/html/_core_Library_Support_for_Multithreading.htm, Februar 2003.

[msd03c] *Microsoft MSDN: COM+ Threading Models*. ms-help://MS.VSCC.2003/MS.MSDNQTR.2003FEB.1031/cossdk/htm/pgcontexts_0acz.htm, Februar 2003.

[msd03d] *Microsoft MSDN: Fibers*. ms-help://MS.MSDNQTR.2003FEB.1031/dllproc/base/fibers.htm, Februar 2003.

[msd03e] *Microsoft MSDN: Scheduling Prioritäten*. ms-help://MS.MSDNQTR.2003FEB.1031/dllproc/base/scheduling_priorities.htm, Februar 2003.

[msd03f] *Microsoft MSDN: Thread Attribute*. ms-help://MS.VSCC.2003/MS.MSDNQTR.2003FEB.1031/vclang/html/_pluslang_The_thread_Attribute.htm, Februar 2003.

[nan06] *Das NANOS-Projekt*. http://research.ac.upc.edu/nanos, März 2006.

[NBF98] NICHOLS, BRADFORD, DICK BUTTLAR und JAQUELINE PROULX FARRELL: *Pthreads Programming*. O'Reilly, 1998.

[npt06] *Beschreibung eines Bugs in NPTL, der die POSIX-Konformität verletzt*. http://nptl.bullopensource.org/index.php?archive=full, Februar 2006.

[Oec01] OECHSLE, RAINER: *Parallele Programmierung mit Java Threads*. Fachbuchverlag Leipzig, 2001.

[pas06] *Übersicht über die POSIX-Versionen und Standards*. http://www.pasc.org/standing/sd11.html, März 2006.

[per] *PerlDoc zu Perl5.8.0: Modul Threads*. perldoc -m Threads.

[per05] *Perlthrtut - tutorial on threads in Perl*. `http://perldoc.perl.org/perlthrtut.html`, Dezember 2005.

[Pla06] *Plattformunabhängigkeit*. `http://de.wikipedia.org/wiki/Plattformunabh%C3%A4ngigkeit`, Januar 2006.

[Por06] *Portabilität*. `http://de.wikipedia.org/wiki/Portabilit%C3%A4t`, Januar 2006.

[pos06] *Testprogramm für POSIX-Konformität*. `http://posixtest.sourceforge.net/`, Februar 2006.

[Pth06] *Homepage der Pthreads-Win32*. `http://sourceware.org/pthreads-win32`, Februar 2006.

[pyt06] *Die Skriptsprache Python*. `http://www.python.org/`, März 2006.

[RR00] RAUBER, THOMAS und GUDULA RÜNGER: *Parallele und verteilte Programmierung*. Springer, 2000.

[SGG03] SILBERSCHATZ, GALVIN und GAGNE: *Operating System Concepts*. John Wiley and Sons, 6. Auflage, 2003.

[uni06] *Offizieller POSIX-Standard als HTML-Version*. `http://www.unix.org/version3/`, Februar 2006.

[Weg03] WEGNER, LUTZ: *Grundlagen der Betriebssyteme*. Universität Kassel, 3. Auflage, 2003.

[wik06a] *Wikipedia: Java-Plattform*. `http://de.wikipedia.org/wiki/Java-Plattform`, Februar 2006.

[wik06b] *Wikipedia: Monitor*. `http://de.wikipedia.org/wiki/Java-Plattform`, Februar 2006.

[wik06c] *Wikipedia: Windows NT*. `http://de.wikipedia.org/wiki/Windows_NT`, Februar 2006.

[yod06] *Tutorial zum Thema Multithreading*. `http://www.yoda.arachsys.com/csharp/threads/volatility.shtml`, Februar 2006.