Systemintegration des User-Level-Schedulings - Betriebssystemdienste zur Unterstützung der parallelen Programmierung

Der Technischen Fakultät der
Universität Erlangen-Nürnberg

zur Erlangung des Grades

DOKTOR-INGENIEUR

vorgelegt von

Christoph Koppe

Erlangen - 1998

Als Dissertation genehmigt von

der Technischen Fakultät der

Friedrich-Alexander Universität Erlangen-Nürnberg

Tag der Einreichung:	28.10.1998
Tag der Promotion:	??.??.1998
Dekan:	Prof. Dr. G. Herold
Berichterstatter:	Prof. Dr. F. Hofmann
	Prof. Dr. M. Dal Cin

© 1998 Christoph Koppe

Alle Rechte, auch das des auszugsweisen Nachdruckes, der auszugsweisen oder vollständigen Wiedergabe (Photokopie, Mikrokopie), der Speicherung in Datenverarbeitungsanlagen und das der Übersetzung, vorbehalten.

Inhaltsverzeichnis

Inhaltsverzeichnis v
Verzeichnis der Abbildungen ix
Verzeichnis der Tabellen xi

1. Einleitung ______ 1

1.1 Aufbau der Arbeit 3
1.2 Anmerkungen zum Sprachgebrauch 4

2. Das Betriebssystem - Flaschenhals der Softwareentwicklung ______ 5

2.1 Aufgaben eines Betriebssystems 5
2.2 Wechselwirkungen in der Entwicklung von Systemkomponenten 7
2.2.1 Ein erstes Beispiel: Der Cache 7
2.2.2 Eine Kategorisierung der Wechselwirkungen 8
2.2.2.1 Auswirkungen von Hardware auf Hardwareentwicklungen 8
2.2.2.2 Auswirkungen von Hardware auf Softwareentwicklungen 9
2.2.2.3 Auswirkungen von Software auf Softwareentwicklungen. 14
2.2.2.4 Auswirkungen von Software auf Hardwareentwicklungen 17
2.2.3 Relevanz für die Betriebssystemforschung 18
2.3 Auswirkungen auf die Betriebssystementwicklung 18
2.4 Der Flaschenhals Betriebssystem 20
2.5 Zusammenfassung 21

3. Betriebssystemarchitekturen und ihre Tragfähigkeit______ 23

3.1 Betriebssystemarchitekturen im Überblick 23
3.1.1 Monolithische Betriebssysteme 24
3.1.2 Mikrokern-Betriebssysteme 26
3.1.3 Betriebssysteme in der aktuellen Forschung 30
3.1.3.1 Exokernel 30
3.1.3.2 SPIN 33
3.2 Modernisierung vs. neue Architekturen 36
3.2.1 Erweiterbarkeit und Adaptierbarkeit 36
3.2.2 Weitere Kriterien 37
3.2.3 Die Rolle heutiger Betriebssystemarchitekturen 37
3.3 Zusammenfassung 38

4. Spezielle Probleme auf Multiprozessorsystemen 39
4.1 Speicherverwaltung / Caches 39
4.2 Prozesse, Threadverwaltung und Threadscheduling 41
4.3 Kernel-Threads 42
4.4 User-Level-Threads 44
4.4.1 Vorteile gegenüber Kernel-Threads 44
4.4.2 Probleme des User-Level-Schedulings 44
4.4.2.1 Scheduling auf zwei Ebenen (Two-Level-Scheduling) 45
4.4.2.2 Blockierungen im Betriebssystemkern 46
4.4.2.3 Systembedingte Verklemmungen 48
4.4.2.4 Signale, unterbrechendes Scheduling und Zeitscheibenverfahren 49
4.5 Lösungsansätze 50
4.5.1 Integration von User-Level-Threads 51
4.5.1.1 *Scheduler Activations* 51
4.5.1.2 XERO Threads 51
4.5.1.3 *Unstable Threads* 52
4.5.1.4 *First-Class User-Level-Threads* 52
4.5.1.5 Hierarchische Scheduler 53
4.5.1.6 Zusammenfassung 54
4.5.2 Thread-Scheduling-Konzepte erweiterbarer Betriebssysteme 54
4.5.2.1 SPIN 54
4.5.2.2 Exokernel 55
4.5.2.3 Neue Wege des Thread-Schedulings? 55
4.5.3 Threadscheduling auf Hardware-Ebene 56
4.6 Zusammenfassung 57

5. Das ELiTE-Projekt 59
5.1 Moderne Schedulingverfahren für UNIX-Systeme 59
5.2 Plattformen 60
5.2.1 Hardware 60
5.2.1.1 Convex/HP SPP 1000/1600 60
5.2.1.2 Sun Ultra Enterprise 3000 62
5.2.2 Systemsoftware 63
5.2.2.1 SPP-UX 63
5.2.2.2 Solaris 64
5.3 ELiTE Teilprojekte 64
5.3.1 Anwendungsinternes Scheduling 64
5.3.1.1 *m(icro)-threads* 65
5.3.1.2 *Sleeping-Threads* 65

5.3.1.3 Unterbrechungsmechanismen 65
5.3.2 Anwendungsübergreifendes Scheduling 66
5.3.2.1 *Follow-On* Scheduling 66
5.3.2.2 Speicherbandbreitenreservierung 66
5.3.3 Zusammenfassung 67
5.4 Struktur und Affinitätsmodelle der *m(icro)-threads* Bibliothek 68
5.4.1 Struktur der *m-threads* 68
5.4.2 Affinitätsmodelle 70
5.5 Zusammenfassung 71

6. *Sleeping-Threads* **73**

6.1 *Sleeping-Threads* - Eine Erweiterung des Kernel-Schedulers 74
6.1.1 Der Basismechanismus 75
6.1.2 User-Level Kontrolle über Ressourcen des Betriebssystemkerns 80
6.1.3 MACH / SPP-UX 83
6.1.4 Implementierung 84
6.1.4.1 Datenstrukturen 85
6.1.4.2 Anbindung des *Sleeping-Threads* Mechanismus 86
6.1.4.3 Vererbung an Serverthreads 88
6.1.4.4 Kritische Abschnitte 88
6.2 Anbindung des User-Level-Schedulings 90
6.2.1 Integration von *Sleeping-Threads* und *m-threads* 90
6.2.1.1 Datenstrukturen für virtuelle Prozessoren und User-Level-Threads . . . 91
6.2.1.2 *Sleeping-Threads* und Threadumschaltung im User-Level 93
6.2.1.3 Verwaltung gebundener Threads 94
6.2.1.4 Erweiterungen des *Sleeping-Threads* Mechanismus 95
6.2.2 Verwaltung der Reservethreads 95
6.2.3 Dynamische Partitionierung - Synchronisation mit CPU-Scheduling 96
6.2.4 Kosten der *Sleeping-Threads* 98
6.3 Einsatzgebiete und Anwendungen 99
6.3.1 Ergonomische Programmierung / Verklemmungsgefahr 99
6.3.2 E/A-intensive Anwendungen 100
6.3.3 Paging / *out-of-core*-Anwendungen 103
6.4 Zusammenfassung 109

7. Unterbrechungsmechanismen **111**

7.1 Standardmechanismen 111
7.2 Anforderungen an einen Unterbrechungsmechanismus 112
7.3 ELiTE Bounce (EBNC) 113
7.4 Anwendungen 114
7.4.1 User-Level-Scheduling 114

7.5 Zusammenfassung 115

8. Zusammenfassung 117

Literatur 119

Lebenslauf 125

Verzeichnis der Abbildungen

Kapitel 3: Betriebssystemarchitekturen und ihre Tragfähigkeit

Abb. 3.1: Beispielszenario für monolithische Betriebssysteme (UNIX) 25
Abb. 3.2: Mikrokern-Betriebssystem (allgemeine Architektur) 27
Abb. 3.3: Beispielszenario für Mikrokern-Betriebssysteme (UNIX auf Basis von MACH) 29
Abb. 3.4: Beispielszenario für Exokernel-Betriebssysteme 33
Abb. 3.5: Beispielszenario für erweiterbare Betriebssysteme (SPIN) 35

Kapitel 4: Spezielle Probleme auf Multiprozessorsystemen

Abb. 4.1: Two-Level-Scheduling 45
Abb. 4.2: Parallelitätseinbruch durch Blockierungen im Kern 47

Kapitel 5: Das ELiTE-Projekt

Abb. 5.1: Aufbau der Convex/HP SPP Architektur 61
Abb. 5.2: Das ELiTE-Projekt 67
Abb. 5.3: Struktur der *m-threads* Bibliothek 68

Kapitel 6: Sleeping-Threads

Abb. 6.1: Geparkte *Sleeping-Threads* in Reserve 76
Abb. 6.2: Blockierter Kernel-Thread ersetzt durch geparkten Thread. 77
Abb. 6.3: Deblockierter Kernel-Thread 78
Abb. 6.4: Deblockierter Kernel-Thread fortgesetzt aus dem User-Level 79
Abb. 6.5: Aufbau der Datenstruktur im gemeinsamen Speicherbereich 87
Abb. 6.6: Einbindung der *Sleeping-Threads* in die *m-threads* Datenstrukturen 92
Abb. 6.7: Die Endlosschleife des Switcherthreads. 93
Abb. 6.8: *grep* mit parallelem Einlesen der Daten 100
Abb. 6.9: Paralleles *du* auf vier virtuellen Prozessoren 102
Abb. 6.10: Paralleles *du* auf einem virtuellen Prozessor 103
Abb. 6.11: Initialisierung mit 49 Threads bei zunehmender Matrixgröße. 104
Abb. 6.12: Glättung mit 49 Threads bei zunehmender Matrixgröße 105
Abb. 6.13: Glättung mit 25 bis 625 Threads (*noaff* und *vtime*) 106
Abb. 6.14: Glättung mit 25 bis 64 Threads (modifiziertes Verfahren) 107
Abb. 6.15: Glättung mit 25 bis 64 Threads 108

Verzeichnis der Tabellen

Kapitel 6: Sleeping-Threads

Tab. 6.1: Kritische Abschnitte über Systemaufruf bzw. Trap 89
Tab. 6.2: Umschaltung auf neuen Thread nach Blockierung im Kern 99
Tab. 6.3: *Sleeping-Threads* Overhead 99

Kapitel 7: Unterbrechungsmechanismen

Tab. 7.1: Signalzustellungszeiten nach erkanntem Timerablauf 112

1 Einleitung

In der Programmierung paralleler Rechensysteme sind heute zwei Modelle weit verbreitet. Ein Programmiermodell, das hauptsächlich auf nachrichtengekoppelten Systemen eingesetzt wird, basiert auf unabhängigen Aktivitätsträgern, die über das Versenden und Empfangen von Nachrichten die parallele Bearbeitung eines Problems ermöglichen. Das zweite Modell orientiert sich an der Architektur von speichergekoppelten Systemen. Mehrere unabhängige Aktivitätsträger, die auf mehreren Prozessoren parallel ausgeführt werden, können gemeinsamen Speicher nutzen. Über den Zugriff auf gemeinsame Daten ist in diesem Modell eine parallele Bearbeitung von Problemen möglich.

Um die Bearbeitung ständig umfangreicherer Aufgaben auf parallelen Rechensystemen mit einem vertretbaren Aufwand an Zeit zu ermöglichen, wurde für beide Programmiermodelle an immer effizienteren Realisierungen gearbeitet. In nachrichtengekoppelten Systemen wird u. a. immer leistungsfähigere Kommunikationshardware eingesetzt, die zum Teil exklusiv durch den Kommunikationsmechanismus des Programmiermodells genutzt wird. In speichergekoppelten Systemen werden statt der Betriebssystemabstraktion des Prozesses - einer Ausführungsumgebung, die neben einem Aktivitätsträger andere wichtige Ressourcen zur Abarbeitung einer Anwendung umfaßt - Threads eingesetzt. Threads stellen unabhängige Aktivitätsträger dar, die parallel ausgeführt werden können aber alle anderen Ressourcen eines Prozesses, darunter den Speicher, gemeinsam nutzen.

Die weitere Fortentwicklung des Konzepts der Threads führte u. a. zu einer Realisierung außerhalb des Betriebssystems. Diese sogenannten User-Level-Threads sind sehr viel effizienter und flexibler als ihr im Betriebssystem realisiertes Gegenstück, die Kernel-Threads. Der Preis für die Realisierung außerhalb des Betriebssystems ist jedoch eine ungenügende Integration mit anderen Systembestandteilen, die zu massiven Problemen und Einschränkungen führt. Es stellt sich daher die Frage, warum für einen - im Bereich des parallelen Rechnens so wichtigen - Systemdienst wie der Threadverwaltung Aspekte der Effizienz, Flexibilität und der Systemintegration nicht im gleichen Maß umgesetzt werden konnten.

Die vorliegende Arbeit geht dieser mit der Entwicklungsgeschichte heutiger Betriebssysteme eng verwobenen Fragestellung nach. Wie sich zeigen wird, hat diese Fragestellung einen sehr viel allgemeineren Charakter und ist in vielen anderen Bereichen ebenfalls von Bedeutung. Ausgehend von dieser Betrachtung werden verschiedene bisher entwickelte Lösungsansätze, sowohl allgemeiner Art als auch speziell auf das Problem der Threads bezogen, dargestellt und diskutiert sowie eine eigene Lösung für die Systemintegration der User-Level-Threads vorgestellt.

Betriebssysteme haben als sehr komplexe Softwaresysteme an einer zentralen Stelle eine bremsende Wirkung auf weite Bereiche der Softwareentwicklung. Sowohl von seiten neuer Hardwareentwicklungen als auch aus verschiedenen Bereichen der Anwendungssoftware werden immer wieder neue Anforderungen an die Systemsoftware und an den Betriebssystemkern herangetragen. Diese Anforderungen können aufgrund einer bereits erreichten Komplexität nicht schnell genug oder aufgrund gewachsener struktureller Probleme in den Betriebssystemen überhaupt nicht oder nicht in der gewünschten Form umgesetzt werden. Seitens der Anwendungssoftware führt dies in letzter Konsequenz dazu, daß Lösungen außerhalb oder an den Betriebssystemen vorbei realisiert werden, wobei die Konsequenzen einer schlechten Systemintegration in Kauf genommen werden müssen. Einen generellen Ausweg aus dieser Situation können nur neue Betriebssystemarchitekturen bieten, die Anpassungen oder Erweiterungen mit geringerem Aufwand und in kürzerer Zeit erlauben.

Gerade die neuen und vielversprechenden erweiterbaren Betriebssystemarchitekturen haben jedoch den Status von Forschungsprojekten noch nicht überwunden, so daß weiterhin über Lösungen im Rahmen von heutigen kommerziellen Betriebssystemen nachgedacht werden muß. Generelle Ansätze - wie sie in Zusammenhang mit neuen Architekturen diskutiert werden - sind in diesem Rahmen nicht realisierbar, einzelne Probleme - wie an verschiedenen Beispielen gezeigt werden wird - können jedoch durchaus gelöst werden.

Auf Multiprozessorsystemen zeigen einige der Probleme und Defizite in heutigen Betriebssystemen drastischere Auswirkungen als auf Monoprozessorsystemen. Insbesondere trifft dies auf die Verteilung der Prozessorressourcen zwischen den einzelnen Threads einer oder mehrerer Anwendungen zu. Es stellt sich im verstärkten Maß die Frage, ob Kernel-Threads mit einer guten Systemintegration oder effiziente und flexible, aber problembeladene Realisierungen im User-Level eingesetzt werden sollen. Diese Frage muß für jede Anwendung im Einzelfall entschieden werden und führt aufgrund eines fehlenden Mittelwegs oft zu unbefriedigenden Ergebnissen. Die Probleme eines ungenügend mit den restlichen Systemkomponenten integrierten User-Level-Schedulings liegen in unterschiedlichen Bereichen. Zu den entscheidenden Problemen gehören:

- Blockierungen im Betriebssystemkern können weder behandelt noch erkannt werden. Dies kann zu dramatischen Parallelitätseinbrüchen bis hin zu Verklemmungen führen.
- Unsynchronisiertes Scheduling durch den Betriebssystemkern und im User-Level kann die erwarteten positiven Effekte speziell angepaßter Schedulingstrategien im User-Level ins Leere laufen lassen oder sogar ins Negative umkehren.

- Auf Unterbrechungen oder Zeitscheibenverfahren basierende Schedulingstrategien können nicht oder nicht in effizienter Form im User-Level realisiert werden.

Innerhalb von ELiTE (**E**rlangen **Li**ghtweight **T**hread **E**nvironment), einem Forschungsprojekt an der Universität Erlangen-Nürnberg, das sich auf Basis eines sehr breit angelegten Profils mit dem Thema Threadscheduling auf Multiprozessorsystemen befaßt, wurden Betriebssystemmechanismen (*Sleeping-Threads* und EBNC (**E**lite **B**ou**nc**e) Timer) entwickelt, die eine bessere Integration der User-Level-Threads erlauben und dadurch unter anderem die genannten Probleme vermeiden können. Weitergehende Untersuchungen im Zusammenspiel mit anderen innerhalb des ELiTE-Projekts entwickelten Verfahren zeigen, daß ein gut integriertes User-Level-Scheduling als Grundlage zur Lösung anderer Probleme in der Systemsoftware beitragen kann. Im Rahmen dieser Arbeit wird ein neues Verfahren zur Berechnung von Problemen, deren Speicherbedarf die Hauptspeicherkapazitäten eines Rechensystems übersteigt (*out-of-core*), vorgestellt. Auf Basis eines integrierten User-Level-Schedulings können die hohen Anforderungen solcher Anwendungen an die virtuelle Speicherverwaltung eines Rechensystems in Grenzen gehalten werden. Dies zeigt die zentrale Bedeutung des Schedulings sowie seine Auswirkungen auf andere Systemdienste gerade auf modernen, leistungsfähigen Rechensystemen.

1.1 Aufbau der Arbeit

Der erste Teil der vorliegenden Arbeit, eine Situationsanalyse, beginnt in Kapitel 2 mit einer näheren Betrachtung der bremsenden Wirkung des Betriebssystems auf andere Bereiche der Softwareentwicklung. In einer systematischen Auflistung von Beispielen für die wechselseitige Einflußnahme zwischen Entwicklungen im Bereich von Hard- und Software werden Probleme bei der Umsetzung unterschiedlicher Anforderungen in der Systemsoftware aufgezeigt. In Kapitel 3 werden verschiedene Betriebssystemarchitekturen - von monolithischen UNIX-Systemen bis hin zu erweiterbaren Betriebssystemen aus der aktuellen Forschung - bezüglich ihrer Eignung für eine Umsetzung dieser Anforderungen untersucht. Kapitel 4 konkretisiert die erkannten Probleme und Defizite der Systemsoftware in Hinblick auf Multiprozessorsysteme. Nach einer Untersuchung der Probleme des User-Level-Schedulings werden unterschiedliche Lösungsansätze zur Integration des Threadschedulings vorgestellt, die von erweiterbaren Betriebssystemarchitekturen bis hin zu Hardwarelösungen reichen.

Im zweiten Teil der Arbeit werden nach einer Vorstellung des ELiTE-Projekts in Kapitel 5 zwei Teilprojekte eingehender behandelt. In Kapitel 6 wird die Funktionsweise und Implementierung des Betriebssystemmechanismus *Sleeping-Threads* zur Integration von User-Level-Threads beschrieben und die Anpassung eines User-Level-Schedulers an diesen Integrationsmechanismus erläutert. Anschließend werden mögliche Anwendungen integrierter User-Level-Threads vorgestellt und anhand von Messungen analysiert. Kapitel 7 geht mit den EBNC-Timern auf die Grundlagen für ein effizientes unterbrechendes oder zeitscheibengesteuertes User-Level-Scheduling, insbesondere auf komplexen Multiprozessorarchitekturen ein. Eine Zusammenfassung in Kapitel 8 schließt die Arbeit ab.

1.2 Anmerkungen zum Sprachgebrauch

Technisch/wissenschaftliche Themen aus dem Bereich der Informationstechnik führen bei einer deutschsprachigen Darstellung zu unschönen sprachlichen Formen. Aufgrund fehlender deutschsprachiger Begriffsbildungen kommt es zu umständlichen Übersetzungen wie zum Beispiel "Ablaufplankoordinator auf Benutzerebene" für "User-Level-Scheduler" oder "Ein- und Auslagerung von Speicherseiten" für "Paging". Um dies zu vermeiden, werden in der vorliegenden Arbeit viele englischsprachige Begriffe verwendet. Dies erfolgt entweder ohne Hervorhebung in eingedeutschter Form (Beispiel: Paging oder Thread) oder in kursiver Schrift, wenn eine eingedeutschte Form zum Beispiel aufgrund der Konjugationsform nicht möglich ist (Beispiel: *direct-mapped* oder *multithreaded*).

Ebenfalls kursiv dargestellt werden Bezeichnungen im Sinn von Eigennamen (Beispiel: *Sleeping-Threads* oder *m-threads*). Direkt aus einem Programmtext übernommene Bezeichnungen, beispielsweise für Variablen, Typbezeichnungen oder Funktionen, werden im Gegensatz zum übrigen Textfluß mit fester Buchstabenbreite dargestellt (Beispiel: `mthread_t`).

2 Das Betriebssystem - Flaschenhals der Softwareentwicklung

Die Frage, inwiefern Betriebssysteme sich als Engpaß für die Fortentwicklung anderer Softwarebereiche erweisen können, ist äußerst relevant. Während die Hardware von Rechensystemen immer leistungsfähiger wird und neue Konzepte anbietet, die sowohl neue Möglichkeiten für die Programmierung bieten als auch von der Systemsoftware unterstützt werden müssen, hinkt die Entwicklung der Software hinterher. Die Systemsoftware spielt hierbei in zweifacher Hinsicht eine Schlüsselrolle. Betriebssysteme sind hoch komplexe Programmsysteme, die nur sehr langsam angepaßt und weiterentwickelt werden und nicht nur mit der schnell fortschreitenden Hardwareentwicklung, sondern auch mit der Softwareentwicklung in anderen Bereichen nicht mithalten können. Gleichzeitig bilden sie die Schnittstelle zur Hardware eines Rechensystems und damit eine Schranke für die Weiterentwicklung anderer Softwarekomponenten. Mit anderen Worten: Betriebssysteme bilden einen Flaschenhals in der Softwareentwicklung.

Im folgenden werden die Aufgaben, die ein Betriebssystem innerhalb eines Rechensystems erfüllt, erläutert. Anschließend wird, nach einem einführenden Beispiel, eine Kategorisierung der gegenseitigen Einflußnahme in der Entwicklung von Rechensystemen vorgenommen und anhand weiterer Beispiele ein Überblick über die gegenseitige Beeinflussung bei der Weiterentwicklung von Systemkomponenten gegeben, wobei insbesondere auf Auswirkungen bezüglich der Betriebssystementwicklung eingegangen wird. Abschließend wird die bremsende Wirkung der langsamen Betriebssystementwicklung eingehender erörtert.

2.1 Aufgaben eines Betriebssystems

Der Begriff des Betriebssystems ist mit der oben verwendeten Bezeichnung als "Schnittstelle zur Hardware eines Rechensystems" für den Rahmen dieser Arbeit nicht ausreichend definiert. Auch existiert im allgemeinen, nach Silberschatz und Galvin, überhaupt keine vollständige und adäquate Definition des Begriffs. Ebenso gibt es keine allgemein anerkannte Definition, welche Software Bestandteil eines Betriebssystems ist und welche nicht. Betriebssysteme lassen sich sehr viel einfacher über die Aufgaben definieren, die sie erfüllen, als über das, was sie darstel-

len. “It is easier to define operating systems by what they *do*, rather than by what they *are*” [SilGal94].

Im folgenden werden die Aufgaben eines Betriebssystems durch eine Aufteilung in drei Bereiche beschrieben, wobei im wesentlichen nur Aspekte berücksichtigt werden, die im Rahmen dieser Arbeit von Bedeutung sind.

- Abstraktionsbildung:
  Eine der wichtigsten Aufgaben eines Betriebssystems ist das Verbergen der Hardwaredetails eines Rechensystems. Kein Anwender möchte beispielsweise einen Festplattenkontroller direkt ansprechen und selbst bestimmen, auf welchem Plattensektor seine Daten gespeichert werden. Ein Betriebssystem bildet Abstraktionen, um den Umgang mit einem Rechensystem zu erleichtern. Weitere Beispiele für solche Abstraktionen bzw. Bereiche, in denen sie eingesetzt werden, sind:
  - Dateisysteme und E/A-Treiber
  - Programmausführung / Prozesse
  - Zugriff auf Statusinformationen
    (z.B. Cachemiss-Zähler, Zeitmessung)
- Virtualisierung:
  Der Begriff der Abstraktion reicht nicht aus, um die Aufgaben eines Betriebssystems zu beschreiben. So führt beispielsweise die Forderung, ein Rechensystem für mehrere Anwendungen gleichzeitig zu verwenden, zum Konzept der Virtualisierung. Innerhalb einer einzelnen Anwendung soll es selbstverständlich nicht notwendig sein, den Mehrprogrammbetrieb zu berücksichtigen. Diese Forderung wird erfüllt, indem das Betriebssystem jeder Anwendung eine Ausführungsumgebung (Prozeß) zur Verfügung stellt, innerhalb derer von der Aufteilung des Rechensystems abstrahiert werden kann. Eine andere Sichtweise ist, daß das Betriebssystem jede Anwendung auf ihrem eigenen virtuellen Prozessor ausführt, unabhängig von der tatsächlichen Prozessoranzahl des Rechensystems. Ein weiteres Beispiel stellt der virtuelle Speicher dar. Durch Auslagerung von Speicherbereichen auf Sekundärspeicher (im allgemeinen durch Festplatten realisiert) kann eine Anwendung beispielsweise über mehr Speicher verfügen, als in einem Rechensystem tatsächlich installiert ist.
- Schutzumgebung:
  Der Mehrprogramm- bzw. Mehrbenutzerbetrieb eines Rechensystems, aber auch schon die Notwendigkeit einer gegenüber Programmfehlern toleranten Ausführungsumgebung für einzelne Programme, führt zu einer weiteren Anforderung an Betriebssysteme. Betriebssysteme müssen, abhängig von dem Umfang ihrer Aufgaben, eine Schutzumgebung sowohl für die Benutzer des Systems als auch für einzelne ablaufende Programme (Prozesse) untereinander anbieten. Ein Beispiel für die Schutzumgebung auf Benutzerebene ist der Zugriffsschutz bezüglich der gespeicherten Daten verschiedener Benutzer eines Rechensystems. Eine Schutzumgebung auf Prozeßebene muß die Zugriffsrechte eines Prozesses gegenüber anderen und gegenüber Daten des Betriebssystems selbst regeln so-

wie das gesamte Rechensystem gegenüber einem Fehlverhalten einzelner Prozesse schützen. Im Rahmen dieser Arbeit ist nicht nur ein Schutz bezüglich fataler Fehler einzelner Prozesse von Interesse, sondern vor allem ein Schutz der einzelnen Prozessen zugeteilten Ressourcen. Gerade die Eigenschaft, Ressourcen eines Rechensystems fair zu verteilen, kann verloren gehen, wenn einzelnen Benutzern im Rahmen von Systemerweiterungen die Möglichkeit gegeben wird, interne Systemabläufe zu beeinflussen.

2.2 Wechselwirkungen in der Entwicklung von Systemkomponenten

2.2.1 Ein erstes Beispiel: Der Cache

Einer der herausragenden Entwicklungssprünge in der Informationstechnologie besteht in der Weiterentwicklung der Mikroprozessoren. Neben konzeptionellen Entwicklungen, wie z.B. Pipelining und parallelen Ausführungseinheiten, führte auch die Erhöhung der Taktrate im Gigahertzbereich zu einer erheblichen Leistungssteigerung. In anderen Bereichen der Hardwareentwicklung konnten Geschwindigkeitssteigerungen in diesem hohen Maß nicht erreicht werden. Bei Komponenten, die eng mit dem Mikroprozessor gekoppelt sind, führt dies in heutigen Systemen zu großen Problemen. Bussysteme und Speicher sind im Vergleich zu den Prozessoren zu langsam und bremsen diese aus, sobald Zugriffe auf Ressourcen außerhalb des Prozessors erfolgen. Nach Hennessy und Patterson [HenPat96] lag der jährliche Leistungszuwachs der Prozessoren bezüglich der Integerleistung in den letzten Jahren bei 58% während die Speicherzugriffszeiten lediglich um jährlich 7% verbessert werden konnten. Ein Prozessor muß beispielsweise ein Vielfaches der Zeit, die er zur Addition zweier Werte benötigt, darauf warten, daß die Werte vom Speicher in den Prozessor geladen werden. 40 bis 50 Wartezyklen ohne Nutzung sind bei Speicherzugriffen in heutigen Systemen keine Seltenheit. Um solche bremsenden Speicherzugriffe zu vermeiden, werden in heutigen Systemen Speicher mit einer sehr viel kürzeren Zugriffszeit als Puffer zwischen Prozessor und Hauptspeicher gesetzt. Diese sogenannten Cache-Speicher werden genutzt, um aus dem Hauptspeicher geladene Werte zwischenzuspeichern, damit bei einer wiederholten Verwendung schneller auf diese zugegriffen werden kann. Da schnelle Speicher - Cache-Speicher sind im Idealfall innerhalb eines Prozessortakts zugreifbar - nicht in der von heutigen Systemen verlangten Größe realisiert werden können, sind sie nur als Cache einsetzbar, um einen geringen Anteil (oft < 1‰) des Hauptspeichers zwischenzuspeichern. Bei einem Zugriff auf Speicherbereiche, die nicht im Cache liegen, müssen Cacheinhalte verdrängt und gegebenenfalls in den Hauptspeicher zurückgeschrieben werden.

Dieses erste Beispiel zeigt, wie eine Weiterentwicklung einzelner Komponenten im Gesamtsystem Probleme entstehen läßt. In diesem Fall wurde ein Problem, das durch eine unterschiedliche technologische Weiterentwicklung verschiedener Hardwarekomponenten entstanden ist, durch die Einführung einer weiteren Hardwarekomponente, einer Erweiterung der Rechnerarchitektur, gelöst. Das Beispiel zeigt aber auch, daß solche Probleme Auswirkungen auf die Software eines Rechensystems haben können: Es ist offensichtlich, daß die Effizienz des weiteren

Programmablaufs vom aktuellen und zukünftigen Inhalt des Cache abhängig ist. Für eine effiziente Nutzung cache-basierter Systeme ist es folglich notwendig, innerhalb eines Programms über Informationen bezüglich des Zustands des Cache zu verfügen, um diesen beeinflussen zu können. Dies muß durch die Systemsoftware unterstützt werden.

2.2.2 Eine Kategorisierung der Wechselwirkungen

Die Einführung oder Weiterentwicklung von Komponenten eines Rechensystems kann, wie hier an einem einfachen Beispiel gezeigt wurde, zur Notwendigkeit von neuen Entwicklungen in ganz anderen Bereichen führen. Dies gilt sowohl für den Bereich der Hardware, der Software - insbesondere der Systemsoftware - als auch untereinander. Es sind folgende Fälle denkbar:

- Verbesserungen oder Neuentwicklungen von Hardwarekomponenten erfordern neue Entwicklungen in anderen Hardwarebereichen. Als Beispiel kann die oben erwähnte Entwicklung von Cache-Speichern angeführt werden.
- Verbesserungen von Hardwarekomponenten, insbesondere aber architekturelle Neuerungen, erzwingen neue Konzepte in der Softwaretechnik. Als erstes Beispiel dient wiederum der Cache.
- Weiterentwicklungen in der Software können Änderungen in anderen Softwarebereichen erzwingen. Beispielsweise sind nachrichtengekoppelte Systeme für parallele Anwendungen auf Basis von Message-Passing-Bibliotheken nur dann effizient nutzbar, wenn die Systemsoftware eine effiziente Nutzung der Kommunikationshardware zuläßt und nicht durch Standardrealisierungen einen zu hohen Overhead erzeugt.
- Neue Konzepte im Bereich der Software, z.B. neue Programmierparadigmen, können eine Weiterentwicklung der Rechnerarchitektur nach sich ziehen. So legt beispielsweise das Konzept der leichtgewichtigen Prozesse (Threads) nahe, Prozessoren zu entwickeln, die diese direkt unterstützen.

Weiterhin werden die verschiedenen Fälle von Abhängigkeiten anhand von Beispielen näher betrachtet. Die gewählten Beispiele haben nicht den Anspruch, die Vielfalt der möglichen gegenseitigen Beeinflussung abzudecken. Sie sind bewußt aus Bereichen gewählt, in denen ein enger Zusammenhang zu Betriebssystemaspekten zu finden ist. Die Beispiele beziehen sich sowohl auf Konzepte in heutigen Betriebssystemen, als auch auf Entwicklungen, die in kommenden Systemen zu erwarten sind.

2.2.2.1 Auswirkungen von Hardware auf Hardwareentwicklungen

Das als Einstieg gewählte Beispiel des Cache zeigt, wie Entwicklungen im Bereich der Hardware weitere Fortentwicklungen der Rechnerarchitektur nach sich ziehen. Dies ist selbstverständlich nicht anders zu erwarten, und die Auswirkungen von Hardwareentwicklungen auf die restliche Hardware eines Rechensystems sind auch nicht der entscheidende Punkt im Rahmen dieser Arbeit. Von Interesse sind vielmehr die Auswirkungen auf die Softwaretechnik, insbesondere auf die Entwicklung der Systemsoftware, die in den nächsten Abschnitten dieser Kategorisierung behandelt werden. Dennoch werden - der Vollständigkeit halber - hier einige weite-

re Beispiele aufgeführt. Sie sind unter anderem auch deshalb interessant, weil sie zum Teil, wie auch im Fall des Cache, die Begründung für architekturelle Eigenschaften von Rechensystemen darstellen, die aus Sicht der Softwareentwicklung zu nicht zu unterschätzenden Problemen führen.

NUMA-Architekturen

Ebenso wie die Entwicklung des Cache eine Reaktion auf lange Speicherzugriffszeiten darstellt, ist die Einführung von NUMA-Architekturen[1] in Zusammenhang mit der begrenzten Bandbreite von Bussystemen zu sehen. Multiprozessorsysteme, die durch die Nutzung eines gemeinsamen Speichers durch alle Prozessoren realisiert werden (*shared-memory*), sind, soweit die Speicherzugriffe über einen gemeinsamen Bus erfolgen, in der Anzahl der Prozessoren begrenzt. In NUMA-Architekturen wird dieser Engpaß vermieden, indem mehrere Module mit einer aus Sicht der Busbandbreite unproblematischen Prozessoranzahl, eigenem Speicher und eigenem Bussystem gebildet werden. Die einzelnen Module werden über ein Verbindungsnetzwerk gekoppelt, das Zugriffe auf den Speicher anderer Module erlaubt. Der Zugriff auf entfernten Speicher wird im allgemeinen durch Hardware realisiert oder zumindest unterstützt, ist aber deutlich langsamer als ein Speicherzugriff innerhalb eines Moduls. Die unterschiedlichen Zugriffszeiten sind das charakteristische Merkmal von NUMA-Architekturen. Im Fall der Convex/HP[2] SPP-Architektur [Convex93], die im Rahmen des Erlanger ELiTE-Projekts[3] eingesetzt wird, ist ein Zugriff auf entfernte Module um den Faktor vier langsamer.

Direct Memory Access (DMA)

Das letzte Beispiel zeigt, im Gegensatz zum Cache, der als “Workaround” angesehen werden kann, eine konsequente Weiterentwicklung auf. Obwohl Ein-/Ausgabe-Prozessoren (E/A) eingesetzt wurden, um den Hauptprozessor bei der Bedienung von E/A-Geräten zu entlasten, mußten die gesendeten bzw. empfangenen Daten weiterhin vom Hauptprozessor zwischen E/A-System und Hauptspeicher kopiert werden. Gerade bei blockorientierten E/A-Geräten bedeutet dies eine erhebliche Belastung des Hauptprozessors. Daher wurden E/A-Prozessoren entwickelt, die selbständig auf den Hauptspeicher zugreifen können. Auch diese *direct memory access* (DMA) genannte Technik hat zum Beispiel in Zusammenhang mit Caches (siehe Abschnitt 2.2.2.2) Auswirkungen auf die Systemsoftware.

2.2.2.2 Auswirkungen von Hardware auf Softwareentwicklungen

Caches und Systemsoftware

Auch in dieser Kategorie wird das Eingangsbeispiel Cache als erstes genannt, da an diesem die Defizite in der Systemsoftwareentwicklung besonders deutlich dargelegt werden können. In kommerziellen Betriebssystemen werden Caches, die in fast allen heutigen Mikroprozessorarchitekturen zu finden sind, nur insoweit berücksichtigt, wie dies für die Systemkonsistenz not-

1. **N**on **U**niform **M**emory **A**ccess (NUMA) - Systeme mit unterschiedlichen Speicherzugriffszeiten auf verschiedenen Speicherbereichen.
2. Der Parallelrechnerhersteller Convex wurde während der Projektlaufzeit durch Hewlett Packard (HP) aufgekauft.
3. Auf das **E**rlangen **Li**ghtweight **T**hread **E**nvironment (ELiTE) wird in Kapitel 5 näher eingegangen.

wendig ist. Darüberhinaus gehende Unterstützung findet sich nur in Ausnahmen beziehungsweise in Forschungssystemen. Besser sieht es im Bereich der Anwendungsentwicklung aus. Wissen über Caches wird in optimierenden Compilern und in Werkzeugen zur Performanz-Analyse (Profiling) eingesetzt.

Zum Verständnis der folgenden Beispiele werden hier einige Begriffe kurz eingeführt. Für eine genaue Definition und eingehende Abhandlung wird auf Hennessy und Patterson [HenPat96] bzw. andere einschlägige Literatur verwiesen.

- Cachehit/Cachemiss:
  Ein Cachehit liegt vor, wenn sich das referenzierte Speicherwort bereits im Cache befindet. Im anderen Fall spricht man von einem Cachemiss.
- Cacheline:
  Die Abbildung zwischen Hauptspeicher und Cache erfolgt nicht auf Basis einzelner Speicherworte. Mehrere aufeinanderfolgende Speicherworte werden jeweils zu einem Block (Cacheline) zusammengefaßt und bilden die Übertragungseinheit zwischen Hauptspeicher und Cache. Wird auf ein Speicherwort zugegriffen, das sich nicht im Cache befindet, wird die gesamte Cacheline in den Cache übertragen, nachdem zuvor gegebenenfalls eine Cacheline in den Hauptspeicher zurückgeschrieben wurde. Aus Sicht des Prozessors ist der Speicherzugriff beendet, sobald das angesprochene Speicherwort übertragen ist. Das Speichersubsystem überträgt den Rest der Cacheline, während der Prozessor den Programmablauf fortsetzen kann.
- Direkt abgebildete (*direct-mapped*) und assoziative Caches:
  - *direct-mapped* Caches zeichnen sich durch eine eindeutige Abbildung des Hauptspeichers auf den Cache aus. Im Falle eines Cachemiss wird ein Speicherbereich, der auf die gleiche Cacheline abgebildet wird, aus dem Cache verdrängt, auch wenn andere Teile des Cache aktuell nicht verwendet werden. Die Abbildung erfolgt im allgemeinen nach der Regel:

    (*Cacheline-Adresse*) MODULO (*Anzahl der Cachelines im Cache*)

  - Bei voll-assoziativen Caches kann ein Speicherbereich auf jede Cacheline des Cache abgebildet werden. Die Auswahl der Cacheline erfolgt im allgemeinen nach der Zeitdauer seit dem letzten Zugriff auf eine Cacheline (*least-recently-used* - LRU) oder zufällig. Der Aufwand an Hardware für assoziative Caches ist erheblich höher.
  - Einen Mittelweg stellen die mengen-assoziativen Caches dar. Man spricht von einem n-fach mengen-assoziativen Cache, wenn n Cachelines zu jeweils einer Menge zusammengefaßt werden. Wie im Fall des *direct-mapped* Cache wird zunächst über eine MODULO-Rechnung eine Menge bestimmt. Innerhalb dieser Menge kann jede Cacheline verwendet werden. Die Auswahl erfolgt wiederum zufällig oder über die LRU-Strategie. Üblich sind 2 oder 4-fach mengen-assoziative Caches.

- Physikalisch und virtuell indizierte Caches:
    - Zur Bestimmung der Cacheline wird bei physikalisch indizierten Caches von der physikalischen Adresse des referenzierten Speicherwortes ausgegangen. Die Überprüfung des Cache auf einen Treffer kann daher erst nach der Berechnung der physikalischen Adresse erfolgen. Der Vorteil dieser Methode liegt in der systemweiten Eindeutigkeit einer physikalischen Adresse.
    - Wird im Fall von virtuell indizierten Caches die virtuelle Adresse verwendet, kann die Bestimmung der Cacheline und die Umrechnung der virtuellen in eine physikalische Adresse parallel erfolgen. Ein Zugriff auf den Cache ist folglich schneller möglich. Die virtuelle Indizierung birgt jedoch das Problem, daß die gleiche virtuelle Adresse auf verschiedene physikalische Adressen verweisen kann oder eine physikalische Adresse durch verschiedene virtuelle Adressen repräsentiert wird.

Assoziative Caches werden im Rahmen dieser Arbeit nicht weiter betrachtet. Auf Unterschiede bezüglich der Indizierungsart wird in einigen Fällen hingewiesen. Entscheidend für die Betrachtungen im Rahmen dieser Arbeit ist die Abbildung von Speicheradressen auf Cacheadressen bei *direct-mapped* Caches.

Wie bereits erwähnt, können Caches einen inkonsistenten Zustand eines Rechensystems verursachen. Betriebssysteme für cache-basierte Architekturen müssen daher besondere Mechanismen bereitstellen. Beispielsweise muß das Betriebssystem zu bestimmten Zeitpunkten gewährleisten, daß alle modifizierten Daten bestimmter Speicherbereiche in den Hauptspeicher zurückgeschrieben wurden bzw. daß Speicherbereiche in keinem Cache gespiegelt sind. Dies ist unter anderem bei DMA-Zugriffen notwendig, da E/A-Prozessoren im allgemeinen am Cache vorbei auf den Speicher zugreifen. Im Fall von virtuell indizierten Caches kommen weitere mögliche Fälle von Inkonsistenzen hinzu, die von Betriebssystemen vermieden oder behandelt werden müssen. Eine Speicherzelle kann zum Beispiel mehrfach im Cache gehalten werden, wenn sie unter verschiedenen virtuellen Adressen angesprochen wird (*alialising*). Von Wheeler und Bershad wird die Konsistenzproblematik bei virtuell indizierten Caches am Beispiel des HPPA/RISC-Prozessors, der in der Convex/HP SPP-Architektur verwendet wird, umfassend betrachtet [WheBer92]. Ansonsten findet sich in heutiger kommerzieller Systemsoftware kaum etwas, um die Leistungsfähigkeit cache-basierter Systeme auszunutzen. Einige Systeme bieten Schnittstellen, über die Profiling-Werkzeuge auf Cachemiss- und andere Zähler des Rechensystems zugreifen können. In anderen Systemen ist eine Analyse der Cacheausnutzung von Anwendungen nur am Betriebssystem vorbei durch direkte Zugriffe auf die Hardware möglich. Dabei könnte die Leistungsfähigkeit cache-basierter Rechensysteme über eine adäquate Behandlung der Caches durch die Systemsoftware deutlich gesteigert werden.

Im Bereich der Speicherverwaltung sollte vermieden werden, daß Speicherbereiche, auf die innerhalb kurzer Zeiträume gleichzeitig zugegriffen wird, auf gleiche Cachelines abgebildet werden. Insbesondere bei physikalisch indizierten Caches liegt dies im Aufgabenbereich der Systemsoftware, da die Anwendungen selbst keinen Einfluß auf die Lage der Daten im Speicher haben. Bei einer Speicheranforderung können die Speicherseiten (*page*) so gewählt werden, daß sie nicht auf gleiche Cachebereiche - man spricht von Seiten gleicher Farbe (*color*) bezüglich

des Cache - abgebildet werden. Bei diesem Ansatz ergeben sich Probleme, wenn die Speicherverwaltung Seiten aus- und einzulagern (Paging) beginnt. Bei der Einlagerung von Seiten muß deren ursprüngliche Farbe berücksichtigt werden. Weiterhin kann der parallele Zugriff auf Speicherbereiche gleicher Farbe, insbesondere durch verschiedene Anwendungen, im allgemeinen erst zur Laufzeit dynamisch erkannt werden. Ein neuerer Ansatz, das *page-recoloring*, versucht daher, solche Konflikte zu erkennen und gegebenenfalls Speicherbereiche in Seiten anderer Farbe umzukopieren. Bershad et al. haben gezeigt, daß eine solche Strategie sowohl mit spezieller Hardwareunterstützung als auch unter Verwendung von Standardhardware realisiert werden kann [BLRC94a und BLRC94b].

Eine ebenso bedeutende Rolle spielt der Cache bei der Entscheidung, welcher Prozeß oder Thread[4] als nächster ausgeführt werden soll. Wenn Schedulingstrategien den Zustand der Caches nicht berücksichtigen, wird eine effiziente Nutzung cache-basierter Systeme dadurch beeinträchtigt, daß ein Thread, der gerade einen Cacheinhalt aufgebaut hat, unterbrochen und erst fortgesetzt wird, nachdem seine Arbeitsmenge vollständig oder teilweise aus dem Cache verdrängt wurde. Im ungünstigsten Fall wird er auf einem anderen Prozessor und damit auch unter Verwendung eines anderen Cache fortgesetzt. Bei Betriebssystemen für Multiprozessoren sind Schedulingstrategien relativ weit verbreitet, die versuchen, ein zu häufiges Wechseln des Prozessors, auf dem ein Thread ausgeführt wird, zu vermeiden. Solche Strategien werden als Lokalitätsscheduling (*locality-scheduling*) bezeichnet.

Im Rahmen des ELiTE-Projekts werden Strategien entworfen, implementiert und untersucht, die sehr viel weitergehende Informationen über den Systemzustand nutzen, um ein mit cachebasierten Systemen verträgliches Scheduling zu realisieren. Eine Variante besteht darin, Aussagen über den Zustand der Caches aus Daten über den vergangenen Verlauf des Schedulings sowie unter Ausnutzung von Cachemiss- oder Cachehit-Zählern zu gewinnen und darauf basierend die Prioritäten der Threads zu bestimmen [Bellos95, Stecker95 und BelSte96]. Das Zusammenspiel dieser cache-optimierten Strategien mit dem in Kapitel 6 vorgestellten Betriebssystemmechanismus *Sleeping-Threads* wird in Abschnitt 6.3 eingehend untersucht. Es zeigt sich, daß die hier entwickelten Ansätze für ein cache-optimiertes Scheduling zur Lösung eines Problems aus dem Bereich der Speicherverwaltung beitragen können. Ein zweiter Ansatz nutzt dynamisch gewonnene Verwandschaftsbeziehungen zwischen Threads bezüglich der referenzierten Speicherbereiche aus, um zu vermeiden, daß zwischen zwei Threads mit sich überschneidenden Arbeitsmengen und somit potentiell gemeinsamen Cacheinhalten ein dritter Thread ausgeführt wird [Bellos97b]. Erste theoretische Überlegungen zur Ausnutzung solcher Verwandschaftsbeziehungen sind im “Computational Field Model” von Tokoro [Tokoro90] zu finden.

4. Diese Arbeit befaßt sich im wesentlichen mit Systemen, die Threads als Basiseinheit für das Scheduling verwenden. Prozesse stellen lediglich eine Ausführungsumgebung für einen oder mehrere Threads dar. Wegen der besseren Lesbarkeit wird in Fällen, in denen es allgemein um Prozeß- oder Threadscheduling geht, nur der Begriff des Threads verwendet.

NUMA-Architekturen

NUMA-Architekturen sind nicht nur durch unterschiedliche Speicherzugriffszeiten charakterisiert, sondern auch durch längere Zugriffszeiten im Vergleich zu Multiprozessoren der UMA-Kategorie (**U**niform **M**emory **A**ccess), selbst bei Zugriffen auf den lokalen Speicher eines Moduls. Dieser Effekt ist durch die komplexere Speicheransteuerungslogik zu erklären. Cachemisses führen im Fall von Zugriffen auf entfernten Speicher zu erheblich und im Fall von lokalen Zugriffen immerhin noch zu deutlich stärkeren Performanzeinbrüchen als bei UMA-Architekturen. Daher gewinnen alle Techniken zur besseren Nutzung von Caches im Fall von NUMA-Architekturen eine noch größere Bedeutung. Dies gilt insbesondere dann, wenn - wie im Fall der Convex/HP SPP - durch Verwendung von lokalem Speicher als Cache für entfernten Speicher erweiterte Cache-Hierarchien gebildet werden.

Systemsoftware für NUMA-Architekturen muß eine erweiterte Schnittstelle zur Speicherverwaltung zur Verfügung stellen. Der Benutzer muß beeinflussen können, in welchen Bereichen angeforderter Speicher tatsächlich angelegt wird. Weiterhin können verschiedene Speicherklassen angeboten werden, wobei auch Kombinationen von lokalem und entferntem Speicher, wie zum Beispiel die *far-shared* und *block-shared* Speicherklassen der Convex/HP SPP-Architektur [Convex94], sinnvoll sein können.

Ein-/Ausgabe Cache (*buffer cache*)

Um die E/A-Operationen bezüglich der im Vergleich zum Hauptspeicher sehr langsamen blockorientierten Peripheriegeräte[5] und damit die Dateisystemzugriffe zu beschleunigen, werden Teile des Hauptspeichers als Cache für E/A-Blöcke verwendet. Schreiboperationen werden extrem beschleunigt, da sie aus Sicht der Anwendung beendet sind, sobald die Daten in den Cache geschrieben wurden. Für Leseoperationen muß nicht auf das E/A-Gerät zugegriffen werden, soweit die referenzierten Blöcke bereits im Cache zwischengespeichert sind. Die Blöcke des Cache werden nach der LRU-Strategie verwaltet, so daß die am längsten ungenutzten freigegeben werden, wenn neue freie Blöcke notwendig sind. Zusätzlich werden beim Lesen von einem E/A-Gerät eine festgelegte Anzahl von Folgeblöcken ebenfalls in den Cache übertragen, und zwar in der Annahme, daß auch diese in Kürze benötigt werden (*read ahead*).

Schon 1981 legte Stonebraker dar, daß diese weit verbreitete Implementierung von Dateisystemen für Datenbanken wenig geeignet ist und erhebliche Auswirkungen auf deren Performanz haben kann [Stoneb81]. Ein übliches Vorgehen bei Datenbanken ist daher, diese Abstraktionen des Betriebssystems nicht zu nutzen und stattdessen eigene Strategien zu implementieren. Der Zugriff auf exklusiv vom Datenbanksystem genutzte Plattenspeicher erfolgt über alternative Schnittstellen der Betriebssysteme, die ein direktes, ungepuffertes Schreiben und Lesen einzelner Plattenblöcke erlauben. Andere Arbeiten zeigen, daß deutliche Geschwindigkeitssteigerungen bei Dateizugriffen möglich sind, wenn den Anwendungen erlaubt wird, die Cachingstrategien zu beeinflussen [CaFeLi94].

Als Alternative zum Dateizugriff über den *buffer cache* hat sich eine zweite Schnittstelle etabliert. Der Zugriff auf Dateien wird über die virtuelle Speicherverwaltung realisiert, über deren

5. Im wesentlichen sind dies die Plattenspeicher.

Paging-Mechanismus die Plattenblöcke geschrieben und gelesen werden. Die charakteristische Eigenschaft dieser *memory mapped files* ist, daß keine Systemaufrufe zum Schreiben bzw. Lesen benötigt werden, sondern daß dies über normale Speicherzugriffe realisiert wird. Aber auch diese Methode basiert auf dem generellen Mechanismus der Abstraktion des virtuellen Speichers, dessen implementierte Strategien für viele Anwendungsfälle ungeeignet sein können. Auch hier läßt sich zeigen, daß eine Steuerung durch die Anwendungen zu erheblichen Performanzsteigerungen führen kann [KLVA93].

2.2.2.3 Auswirkungen von Software auf Softwareentwicklungen

Prozesse vs. Threads

Der Begriff des leichtgewichtigen Prozesses oder Threads ist aus heutiger Sicht in einem engen Zusammenhang mit Multiprozessorsystemen, also einer Hardwareentwicklung, zu sehen. Dennoch gehört er in diesen Abschnitt der Kategorisierung, da die Abstraktion des Threads schon auf Einprozessorsystemen entwickelt wurde.

Ein Prozeß kann, wie schon in Abschnitt 2.1 dargestellt wurde, als Ausführungsumgebung für ein Programm angesehen werden. Schon bei einem Einprozessorsystem, das im Mehrprogrammbetrieb eingesetzt wird und mehrere Prozesse quasi parallel ausführen kann, liegt es nahe, eine Anwendung durch mehrere Prozesse zu realisieren. Gründe hierfür können zum Beispiel in einer besseren Strukturierung, einer Trennung weitgehend unabhängiger Programmteile, einer besseren Auslastung des Systems oder der Notwendigkeit liegen, trotz blockierender Aufrufe ein Weiterlaufen der Anwendung zu garantieren. Werden solche Anwendungen über mehrere Prozesse realisiert, trifft man jedoch in zwei Bereichen auf potentielle Probleme.

- Prozesse arbeiten unter anderem aufgrund einer gegenseitigen Abschottung in unterschiedlichen Adreßräumen. Daher ist es nicht ohne weiteres möglich, auf Daten eines anderen Prozesses zuzugreifen. Sie müssen entweder über das Dateisystem ausgetauscht oder es müssen explizit Speichersegmente angelegt werden, auf die zwei oder mehr Prozesse Zugriff haben. Insbesondere ist der gemeinsame Zugriff auf Daten unmöglich, die nicht in gemeinsamen Segmenten liegen können. Dies trifft zum Beispiel auf statisch angelegte Daten zu.
- Unter anderem bedingt durch die Umschaltung der Adreßräume sind Wechsel in der Prozeßausführung (Kontextwechsel, *context switch*) sehr langsam. Noch extremer trifft dies bei der Erzeugung neuer Prozesse zu. Dabei ist in vielen Fällen eine gegenseitige Abschottung innerhalb einer Anwendung überhaupt nicht notwendig.

Um diese Probleme zu lösen, läßt man mehrere leichtgewichtige Prozesse (Threads) in einem gemeinsamen Adreßraum laufen. Die Ausführungsumgebung (u.a. Schutzumgebung, Zugriffsrechte, Adreßraum und systemabhängige Datenstrukturen) - oft auch als Task bezeichnet - wird von den eigentlichen Aktivitätsträgern, den Threads, getrennt. Ein herkömmlicher Prozeß kann folglich als eine Task mit genau einem Thread angesehen werden. Der Kontext eines Threads besteht im wesentlichen nur aus Programmzähler, Prozessorstatus, Registersatz und Stack. Eine Umschaltung zwischen Threads des gleichen Adreßraums (der gleichen Task) verursacht daher

weniger Overhead als ein Prozeßwechsel. Der Zugriff auf die gemeinsamen Ressourcen zwischen den Threads einer Task bedarf keiner expliziten Datenübertragung oder spezieller Speichersegmente. Das erst weiter verbreitete Betriebssystem, das diese Abstraktion anbot, ist das an der Carnegie Mellon University entwickelte MACH-System [ABBG+86].

Kernel-Threads vs. User-Level-Threads

Selbst auf Systemen, die Threads als Bestandteil des Betriebssystems (im weiteren als Kernel-Threads bezeichnet) anbieten, kann es sinnvoll sein, Threads im Rahmen eines Laufzeitsystems auf Benutzerebene (User-Level-Threads) zu implementieren. Aus Sicht der Laufzeitumgebung dienen Kernel-Threads oder Prozesse[6] als virtuelle Prozessoren. Analog zum Scheduling von Kernel-Threads auf physikalischen Prozessoren durch das Betriebssystem erfolgt das User-Level-Scheduling auf Basis der virtuellen Prozessoren. Ein Beispiel für einen auf Benutzerebene implementierten Prozeßumschalter stellt die QuickThread-Umgebung [Keppel93 und Reder95] dar, die als Basis für den User-Level-Scheduler des ELiTE-Projekts eingesetzt wird [Stecker95].

Ein tiefergehender Vergleich von Kernel- und User-Level-Threads ist in Kapitel 4 zu finden. Die wichtigsten Punkte müssen jedoch schon im Rahmen dieses Beispiels zur Verdeutlichung aufgezählt werden. Vorteile der User-Level-Threads liegen im fehlenden Overhead des Wechsels in den Kernadreßraum, der zum Beispiel bei der Generierung und bei bestimmten Formen der Synchronisierung sowie bei der Threadumschaltung selbst erforderlich ist. Ein weiterer bedeutender Vorteil liegt in der Flexibilität bezüglich der Schedulingstrategien, die ohne Eingriff in das Betriebssystem variiert werden können. Im Gegensatz zu den kaum beeinflußbaren Strategien des Kernel-Schedulings bieten User-Level-Threadbibliotheken oft mehrere Strategien an. Nachteile liegen vor allem in einer ungenügenden Systemeinbindung. So kann ein User-Level-Scheduler beispielsweise nicht auf Blockierungen im Betriebssystemkern reagieren, was u.a. zu einem Einbruch der Parallelität führen kann. Ebenso kann sich ein konkurrierendes Scheduling bezüglich der Kernel-Threads durch den Betriebssystemkern und der User-Level-Threads durch das Laufzeitsystem störend auswirken.

Eine Vielzahl von Arbeiten hat sich mit der Problematik "Kernel-Threads vs. User-Level-Threads" beschäftigt und Lösungen oder Mittelwege entwickelt [IKNM91, MSLM91, ABLL92, BMVL93, InKaMa93, Koppe94, Koppe95, RieKle96 und Hollma96]. Auch dies macht deutlich, daß weder Kernel-Threads noch User-Level-Threads auf Basis von Kernel-Threads eine gute Abstraktion zur Programmierung *shared-memory*-basierter Multiprozessorsysteme darstellen. Andere Arbeiten befassen sich im Rahmen der Entwicklung neuer Betriebssystemarchitekturen mit Schedulingmechanismen, die sich von dem bislang üblichen Threadscheduling durch den Betriebssystemkern weiter entfernen und weitreichende Teile des Schedulings aus dem Kern auslagern oder unter Kontrolle des User-Level stellen [EnKaTo95 und BSPS+95]. Diese Konzepte werden in den Abschnitten 3.1.3 und 4.5.2 näher betrachtet.

6. Als Basis für User-Level-Threads können ein oder mehrere Prozesse dienen. Dies macht jedoch beim Einsatz auf Multiprozessorsystemen nur in wenigen Spezialfällen Sinn, da eine echte Parallelität innerhalb eines Prozesses nicht möglich ist und bei mehreren Prozessen wiederum das Problem der getrennten Adreßräume besteht. Im weiteren werden User-Level-Threads auf Basis von Kernel-Threads betrachtet.

Die vorliegende Arbeit befaßt sich schwerpunktmäßig mit einem Betriebssystemmechanismus zur Integration des User-Level-Schedulings. User-Level-Threads, die auf Basis der *Sleeping-Threads* realisiert werden, zeigen die angesprochenen Probleme nicht und können über ihre Effizienz bei gleichzeitig guter Systemeinbindung als Basis zur Lösung anderer Problembereiche dienen (siehe Kapitel 6).

Virtuelle Speicherverwaltung / *out-of-core*-Anwendungen

Die Entwicklung von Hardware und Software in den vergangenen Jahrzehnten zeigt, daß es immer Anwendungen gab, die mehr als den verfügbaren Speicher benötigen. Für solche *out-of-core*-Anwendungen mußten im Laufe der Weiterentwicklung der Rechensysteme immer wieder Lösungen gefunden werden. Eine Lösung ist das heute sehr weit verbreitete Konzept des virtuellen Speichers, das - auf den ersten Blick - den zusätzlichen Vorteil bietet, daß die Anwendung von dem tatsächlich installierten Speicher abstrahieren kann. Der Vorteil, bei der Programmierung den zu geringen Speicherausbau nicht berücksichtigen zu müssen, wandelt sich jedoch schnell in einen Nachteil, wenn die Performanz der Anwendungen betrachtet wird. Die Abstraktion muß zwar nicht in jedem Fall zu einem verschwenderischen Umgang mit der knappen Ressource Speicher führen, doch können auch schon eine ungünstige Anordnung von Daten oder unnötig weit gestreute Zugriffsmuster zu einem verstärkten Paging und damit zu einem Einbruch der Performanz führen[7]. Ebenso bedeutend in Hinsicht auf die schlechte Performanz ist, daß die Implementierungen von virtuellem Speicher im allgemeinen nicht für *out-of-core*-Anwendungen optimiert sind. Sowohl die Speicherverwaltung als auch das Scheduling müssen als Betriebssystemmechanismen eine generell verwendbare Basis für eine Vielzahl von Anwendungsszenarien darstellen. Auf allgemein einsetzbaren Rechensystemen ist eine Speicherverwaltung, die den physikalischen Speicher nach einer sinnvollen Strategie zwischen mehreren Anwendungen aufteilt und auch die Anforderungen von interaktiven Anwendungen berücksichtigt, ebenso notwendig. Während rechenintensive, nicht interaktive Anwendungen zu Gunsten anderer größtenteils oder sogar vollständig ausgelagert werden können, sollte für interaktive Anwendungen der Anteil der eingelagerten Speicherseiten zumindest so umfassend sein, daß eine schnelle Reaktionszeit sichergestellt ist. Ein Betriebssystemmechanismus sollte nicht auf die Optimierung bezüglich nur einer Klasse von Anwendungen ausgelegt sein. Ganz im Gegensatz zu dieser Prämisse ging die Entwicklung in die Richtung einer Optimierung zugunsten interaktiver Anwendungen.

Eine effiziente Ausführung von *out-of-core*-Anwendungen kann auf unterschiedliche Weise erreicht werden. Eine Möglichkeit besteht darin, die Speicherverwaltung des Betriebssystems zu umgehen und über einen direkten Zugriff auf Dateisystem oder selbstverwaltete Festplattenbereiche Daten selbst ein- und auszulagern. Dieser Ansatz ist aus Anwendungssicht wenig transparent und kann auf das schon in Abschnitt 2.2.2.2 als Beispiel angesprochene Problem ungeeigneter und unflexibler Dateisystemimplementierungen stoßen. Ein zweiter - aus Sicht der Betriebssystemforschung interessanterer - Ansatz setzt bei einer Verbesserung der Pagingstrategien des virtuellen Speichersystems an. Wie schon in Zusammenhang mit den *memory mapped*

7. Die Analogie zur Cache-Problematik ist nicht überraschend, da der physikalische Speicher auch als ein voll assoziativer Cache für den virtuellen Speicher angesehen werden kann.

files in Abschnitt 2.2.2.2 erwähnt wurde, kann durch steuerndes Eingreifen der Anwendung in die Pagingstrategien oder durch eine Auslagerung der Strategien eine deutliche Leistungssteigerung erreicht werden. Zusammen mit einer automatischen Generierung von frühzeitigen Einlagerungsanforderungen (*pre paging* statt *demand paging*) durch den Compiler kann eine effiziente Ausführung von *out-of-core*-Anwendungen erreicht werden [MoDeKr96]. Weitergehende Konzepte, zum Beispiel eine Verlagerung der virtuellen Speicherverwaltung auf die Benutzerebene [EnKaTo95] oder die Integration von anwendungsspezifischen Speicherverwaltungscodes in den Betriebssystemkern [BSPS+95], wie sie in neueren Betriebssystemarchitekturen verwirklicht sind, können ebenfalls als Basis genutzt werden. Im Rahmen dieser Arbeit wird ein Ansatz vorgestellt, der bei parallelisierbaren Anwendungen eine effizientere Ausführung ohne Modifikationen im Bereich der Speicherverwaltung durch ein an die Pagingstrategien des Betriebssystems angepaßtes Threadscheduling erreicht (siehe Abschnitt 6.3.3).

Anwendungskontrolle über Kommunikationshardware

Das ParaStation-Projekt [BlWaTi98] an der Universität Karlsruhe realisiert den Zugriff auf die Kommunikationshardware eines Workstation-Clusters unter Umgehung des Betriebssystems. Stattdessen wird eine Bibliothek angeboten, die einen direkten Zugriff auf die Kommunikationshardware vom User-Level aus erlaubt und eine Nutzung durch mehrere Anwendungen koordiniert. Die Bibliothek übernimmt - gerade im Fall der Ressourcenverteilung - klassische Aufgaben des Betriebssystems, jedoch ohne einen gegenseitigen Schutz der Anwendungen garantieren zu können. Die Virtualisierung der Kommunikationshardware über diesen Ansatz funktioniert nur so lange zuverlässig, wie alle Anwendungen ausschließlich über Routinen der Bibliothek auf die Hardware zugreifen. Die Vorteile des Verfahrens liegen in einer sehr effizienten Nutzung der Kommunikationshardware für spezifische Aufgaben innerhalb des Parallelrechnersystems ParaStation. Es ist jedoch ungeeignet für generell nutzbare Kommunikationsschnittstellen.

Die experimentelle Betriebssystemarchitektur Exokernel (siehe Abschnitt 3.1.3.1) verfolgt ebenfalls den Ansatz, einer Anwendung einen möglichst direkten Zugriff auf die Hardware eines Rechensystems zu gewähren. Ein Zugriffsschutz, der auf einer sehr tiefen, hardwarenahen Ebene durch den Exokernel realisiert wird, bleibt jedoch erhalten.

2.2.2.4 Auswirkungen von Software auf Hardwareentwicklungen

Hardwareunterstützung für Threads

Eine sehr feingranulare Parallelität von Anwendungen sowie eine bessere Ausnutzung von Rechenkapazitäten kann erreicht werden, wenn die Umschaltung zwischen Threads nicht durch das Betriebssystem realisiert wird, sondern schon der Prozessor zwischen mehreren Threads umschalten kann. Die Threadumschaltung erfolgt nicht nur sehr viel schneller, sie ist auch in Situationen möglich, in denen die Systemsoftware auf herkömmlichen Architekturen nicht eingreifen kann. Eine Threadumschaltung kann zum Beispiel erfolgen, wenn ein Thread auf einen Speicherzugriff warten muß. Der Prozessor kann einen anderen Thread ausführen, statt Warte-

zyklen einzulegen. Dies wiederum hat nicht unerhebliche Auswirkungen auf die Bedeutung von Caches in solchen Architekturen. Auf solche *multithreaded* Prozessoren und Architekturen [IGHS94] wird im Rahmen der Diskussion von Threadschedulingkonzepten in Abschnitt 4.5.3 näher eingegangen. Das seit 1998 kommerziell vertriebene Tera-System ist ein Beispiel für *multithreaded* Architekturen [ACCK+90]. Ein weiterer Ansatz (*simultaneous multithreading*), der die Vorteile von superskalaren und *multithreaded* Prozessoren kombiniert, wird von Tullsen et al. vorgestellt [TuEgLe95].

Hardwaremonitoring

Das Beispiel der Cachemiss-Zähler zeigt, welche Bedeutung eine Analyse der Vorgänge in der Hardware eines Rechensystems für die Performanz des Gesamtsystems oder einer Anwendung haben kann. Von Interesse sind jedoch nicht nur Cachemisses, sondern auch eine Reihe anderer Ereignisse, hauptsächlich aus dem Bereich der Speichersubsysteme. Je nach Intention können dies Zahlen zum Beispiel über Cachehits, Buszugriffe, Zugriffe auf einzelne Speicherbänke oder Zugriffskonflikte sein. Moderne Mikroprozessoren und Rechensysteme stellen daher eine Vielzahl solcher Zähler auf dem Prozessor selbst bzw. in externen Komponenten zur Verfügung. Im allgemeinen sind die Zähler jedoch nur über wenige Register zugreifbar, so daß nur einige Zähler über den gleichen Zeitraum beobachtbar sind. Ein weiteres Problem stellt der Konflikt zwischen einer Nutzung durch Performanzanalysetools, aufgrund derer die Zähler integriert wurden, auf der einen Seite und für eine Laufzeitanalyse durch den Betriebssystemkern oder Laufzeitbibliotheken auf der anderen Seite dar. Eine Erweiterung um zusätzliche Register zum Zugriff auf Ereigniszähler wäre eine sinnvolle Verbesserung der Systemarchitekturen.

2.2.3 Relevanz für die Betriebssystemforschung

Der erste Teil der vorliegenden Arbeit (Kapitel 2, 3 und 4) befaßt sich mit Konzepten, die ein Betriebssystem anbieten muß, um den Anforderungen der Anwendungsprogrammierung auf modernen Multiprozessoren gerecht zu werden und insbesondere eine effiziente Ausnutzung dieser Systeme zu ermöglichen. Es ist offensichtlich, daß hierbei Bereiche aus allen vier in Abschnitt 2.2 genannten Fällen von Interesse sind, da eine adäquate Nutzung der Hardware von modernen Multiprozessoren erreicht werden muß (Fälle 1 und 2), Anforderungen der Anwendungsprogrammierung ihren Niederschlag in adäquaten Betriebssystemkonzepten finden müssen (Fall 3) und Betriebssystemkonzepte häufig Lösungen für Probleme oder Anforderungen darstellen, die sehr viel besser durch neue Hardwarekonzepte zu realisieren sind (Fall 4).

2.3 Auswirkungen auf die Betriebssystementwicklung

Die in Abschnitt 2.2 zur Erläuterung der gegenseitigen Beeinflussung gewählten Beispiele zeigen gleichzeitig eine Reihe von Bereichen auf, in denen heutige kommerzielle Betriebssysteme weder dem Stand der Technik noch den Anforderungen der Anwendungen entsprechen. Dies sind vor allem die Speicherverwaltung auf allen Ebenen (vom Cache bis zum Hintergrundspeicher), die mit sehr ähnlichen Problemen behaftete blockorientierte Ein-/Ausgabe, die Abstrak-

tionen zur parallelen Programmierung und die Ausnutzung von Ereigniszählern in der Hardware. Ebenso deutlich wird der Gegensatz zwischen dem Anspruch des Betriebssystems, generelle Lösungen anzubieten, und den speziellen Anforderungen einzelner Anwendungen. Moderne Betriebssysteme sollten Grundmechanismen anbieten, die einen Kompromiß zwischen genereller Anwendbarkeit und Effizienz darstellen, gleichzeitig aber möglichst einfach erweiterbar und anpaßbar sind, um eine Basis zur effizienten Ausführung von Anwendungen mit speziellen Anforderungen zu bilden.

Monolithische Betriebssysteme bilden eine denkbar schlechte Ausgangsbasis für solche Anforderungen. Prinzipiell sind sie erweiterbar, wenn der Quellcode verfügbar ist. Dies bedeutet jedoch, daß Erweiterungen - mit Ausnahme von frei verfügbaren Systemen - letztendlich nur durch oder mit Unterstützung der Hersteller möglich sind. Eine Ausnahme bilden statisch oder dynamisch ladbare Module, denen enge Grenzen gesetzt sind (siehe Abschnitt 3.1.1). Einen ersten Schritt in Richtung erweiterbarer Systeme stellen Betriebssysteme dar, die auf Mikrokernen (*microkernel*) aufbauen [ABBG+86 und Liedtk96]. Nur grundlegende Betriebssystemdienste sind direkter Bestandteil der Mikrokerne, darüber hinausgehende Funktionalität wird als Prozeß auf Benutzerebene in User-Level-Servern implementiert. Um die Kompatibilität zu bestehenden Systemen zu gewährleisten, wurden Standardbetriebssysteme, wie zum Beispiel UNIX, als User-Level-Server auf Mikrokernen implementiert [GDFR90, RBFG+ und HHLS+97]. In der Praxis führen sie jedoch, wie in Kapitel 3 erläutert wird, zu den gleichen Problemen wie monolithische Systeme.

Betriebssystemarchitekturen, die auf Flexibilität, Erweiterbarkeit und Adaptierbarkeit an Anwendungen ausgelegt sind, bilden einen Schwerpunkt der aktuellen Betriebssystemforschung. Zur Lösung obengenannter Probleme sind diese Systeme besonders gut geeignet, da sie die parallele Integration unterschiedlicher anwendungsspezifischer Erweiterungen erlauben. Beispielsweise ist es möglich, verschiedene Paging-Strategien gleichzeitig auf einem System zu ermöglichen oder anwendungsspezifische Schedulingstrategien zu realisieren, ohne diese als User-Level-Scheduler auf ein wenig geeignetes allgemeines Kernel-Scheduling aufsetzen zu müssen. Zwei Vertreter solcher Forschungssysteme sind das Exokernel-Projekt am MIT [EnKaTo95] und das SPIN-Projekt an der University of Washington in Seattle [BSPS+95].

Ebenso wenig wie Mikrokernsysteme können aktuelle Forschungsprojekte im Bereich der erweiterbaren Betriebssysteme die beherrschende Stellung der UNIX-Systeme außer acht lassen. Um die Chance auf eine weitere Verbreitung zu wahren, müssen sie zumindest eine UNIX-kompatible Schnittstelle anbieten. Sowohl innerhalb des Exokernel-Projekts als auch im SPIN-Projekt wird an UNIX-kompatiblen Erweiterungen gearbeitet [KEGB+97 und Dion96]. In diesem Zusammenhang ist auch einer der Gründe zu sehen, weshalb sich das in Kapitel 5 vorgestellte ELiTE-Projekt mit der Erweiterung kommerzieller Unix-Systeme befaßt.

Welche Betriebssystemarchitektur im Bereich von Arbeitsplatzrechnern am oberen Leistungsspektrum, Hochleistungsservern und technisch/wissenschaftlichem Rechnen die heutige Rolle von UNIX-Systemen einnehmen wird, ist noch nicht vorhersagbar. Bis sich die Lösungskonzepte der aktuellen Forschung niederschlagen, muß über Lösungskonzepte auf Basis der heute verbreiteten Betriebssystemarchitektur UNIX nachgedacht werden, allerdings ohne die Probleme

der Integration in eine seit Jahrzehnten gewachsene Architektur zu vernachlässigen, bei deren Entwicklung Modularität und Erweiterbarkeit keine bedeutende Rolle gespielt haben.

Auf das Microsoft Betriebssystem Windows NT wird im Rahmen dieser Arbeit nicht näher eingegangen. Windows NT stellt aufgrund von Skalierungsproblemen [PerSit96] keine Alternative zu UNIX im Bereich von parallelen Hochleistungsrechnern dar. Bedeutender ist jedoch, daß Windows NT als ein in sich geschlossenes, unter der Kontrolle eines Herstellers stehendes System, für die verschiedenen Formen der Erweiterung, die im Rahmen dieser Arbeit vorgestellt werden, noch ungeeigneter als UNIX-basierte Systeme ist.

Ein weiterer bisher nicht angesprochener Aspekt, der Auswirkungen auf die Betriebssystementwicklung haben sollte, liegt in einem ergonomischen Design von Schnittstellen und Abstraktionen der Betriebssysteme. Es ist nicht allein von Interesse, ob ein Systemdienst verfügbar und effizient ist, ein ebenso wichtiges Kriterium ist die Handhabbarkeit der angebotenen Schnittstelle. Zusammen mit einer geeigneten Abstraktionsbildung, die eine möglichst geringe Kenntnis der internen Abläufe durch die Anwendungsprogrammierer verlangt, hat dies einen bedeutenden Einfluß auf die Kompaktheit, Komplexität und Fehleranfälligkeit der Anwendungsprogramme. Beispiele hierfür werden in Zusammenhang mit Systemabstraktionen zur parallelen Programmierung in den Kapiteln 4 und 6 gegeben.

2.4 Der Flaschenhals Betriebssystem

Die zu Beginn des Kapitels aufgestellte These vom Betriebssystem als Flaschenhals der Softwareentwicklung wird durch die genannten Beispiele belegt. In verschiedenen Bereichen werden Betriebssystemmechanismen nicht genutzt und stattdessen soweit wie möglich durch die Anwendungen unter Ausnutzung von Schnittstellen eines niedrigeren Abstraktionsniveaus umgangen. Auch bieten Betriebssysteme oft generell einsetzbare Mechanismen an, die für spezielle Anwendungen ungeeignet sind. Gleichzeitig sind heutige Betriebssysteme so komplex, daß notwendige Erweiterungen und Verbesserungen nur sehr langsam ihren Weg in kommerzielle Systeme finden können oder aufgrund struktureller Probleme kaum oder nicht mehr integrierbar sind. Ebensowenig können Betriebssysteme als Basis aller anderen Softwareprodukte nicht ohne große Probleme durch neuere Entwicklungen ersetzt werden. Ein Beispiel für die Langlebigkeit einmal etablierter Systeme ist das Betriebssystem UNIX, das trotz vieler Erweiterungen immer noch in wesentlichen Teilen auf mittlerweile 23 Jahre alten Entwicklungen basiert [RitTho74]. Die Vielzahl von Betriebssystemerweiterungen und Lösungen auf Anwendungsebene, an denen gearbeitet wird, zeigt jedoch, daß die heute realisierten Konzepte nicht ausreichen. Die Betriebssystementwicklung hinkt den Anforderungen der Anwendungsprogrammierung hinterher.

Das Fehlen von Konzepten auf der einen und die schwierig zu realisierende Integration neuer Konzepte auf der anderen Seite legen nahe, die Lösung in erweiterbaren Betriebssystemarchitekturen zu suchen. Damit wäre nicht nur die Integration neuer Konzepte gewährleistet, sondern auch eine schnellere Anpassung von Betriebssystemen an Anforderungen aus Software- und Hardwarebereichen möglich.

2.5 Zusammenfassung

Fast alle Entwicklungen, sowohl im Bereich der Hardware als auch der Software, haben Auswirkungen auf Betriebssysteme oder die Systemsoftware im weiteren Sinne. Unter anderem aufgrund der Komplexität der Systemsoftware spiegeln sich jedoch nicht alle Weiterentwicklungen in ausreichendem Maß in der Systemsoftware wider. Der Begriff des "Betriebssystems als Flaschenhals der (Software)entwicklung" beschreibt daher die heutige Situation sehr gut.

Der Trend in der Betriebssystemforschung geht in Richtung erweiterbarer und anpaßbarer Betriebssysteme, um den Anforderungen gerecht zu werden und gleichzeitig die strukturellen Probleme, die zur heutigen Situation geführt haben, zu beseitigen. Aufgrund der marktbeherrschenden Stellung monolithischer Betriebssysteme wie UNIX ist jedoch auch die Integration neuer Konzepte in heutige kommerzielle Betriebssysteme notwendig, um eine bessere Ausnutzung der leistungsfähiger werdenden Hardware zu ermöglichen. Dies im Bereich von modernen parallelen Rechnerarchitekturen zu gewährleisten und gleichzeitig die Grenzen der Anpaßbarkeit solcher Systeme zu untersuchen, ist ein Schwerpunkt des Forschungsprojekts ELiTE.

Dennoch muß an Lösungsansätzen für heutige kommerzielle Betriebssysteme, die auf der Betriebssystemarchitektur UNIX basieren und nur langfristig durch neuere Entwicklungen ersetzt werden können, gearbeitet werden. Weiterentwicklungen im Bereich des Schedulings spielen hierbei eine zentrale Rolle. Dies zeigt sich unter anderem durch die mehrfachen Verweise auf den in Kapitel 6 beschriebenen *Sleeping-Threads* Mechanismus bei verschiedenen in der Kategorisierung der Wechselwirkungen angeführten Beispielen.

Im folgenden Kapitel wird ein Überblick über die Erweiterungsmöglichkeiten in monolithischen Betriebssystemen bis hin zu den Konzepten erweiterbarer Forschungssysteme gegeben. Nach einer Darstellung der für moderne Multiprozessorarchitekturen notwendigen Betriebssystemmechanismen in Kapitel 4 geht der zweite Teil der Arbeit auf das ELiTE-Projekt und die dort implementierten Mechanismen ein.

3 Betriebssystemarchitekturen und ihre Tragfähigkeit

Anhand von Beispielen wurde in Kapitel 2 dargelegt, daß heutige kommerzielle Betriebssysteme in mehreren zentralen Bereichen den Anforderungen der Anwendungsprogrammierung nicht genügen. Nicht nur Verbesserungen bzw. Erweiterungen bezüglich Effizienz oder Funktionalität sind notwendig, sondern auch strukturelle Veränderungen, die eine flexible Anpassung der Systemsoftware an spezielle Anforderungen erlauben.

Nach einem Überblick über verschiedene Betriebssystemarchitekturen wird deren Tragfähigkeit bezüglich notwendiger Erweiterungen und neuer Konzepte erörtert. Ausgehend von diesen Vergleichen wird die im Abschnitt 2.3 begonnene Abwägung zwischen einer modernen Betriebssystemarchitektur im Sinne eines Forschungsbetriebssystems und heutigen kommerziell eingesetzten UNIX-Systemen als Basis für das ELiTE-Projekt weitergeführt.

3.1 Betriebssystemarchitekturen im Überblick

Die Ausrichtung dieser Arbeit auf UNIX-artige Betriebssysteme spiegelt sich in der Auswahl der hier vorgestellten Architekturen und Beispiele wider. Als Beispiel für monolithische Systeme dient UNIX selbst; mit MACH wird ein Mikrokern betrachtet, der hauptsächlich als Basis für UNIX-Implementierungen eingesetzt wird. Auch die gewählten Beispiele aus dem Bereich der Forschungssysteme zeichnen sich dadurch aus, daß sie UNIX-kompatible Erweiterungen anbieten und, wie bereits das Mikrokernsystem MACH, die Ausführung von UNIX-Binärprogrammen[1] erlauben (Binärkompatibilität[2]).

Ein gemeinsames Merkmal aller hier betrachteten Betriebssysteme ist die Ausnutzung verschiedener Privilegierungsstufen der Hardware, um die Integrität des Systems zu gewährleisten. An-

1. Ein für eine bestimmte UNIX-Variante übersetztes und damit ausführbares Programm.
2. Ein binärkompatibles System bietet ein entsprechendes **A**pplication **B**inary **I**nterface (ABI) an. Wird lediglich eine kompatible Programmierschnittstelle angeboten, spricht man von einem **A**pplication **P**rogramming **I**nterface (API). In diesem Fall muß ein Programm als Quellcode vorliegen und neu übersetzt werden.

wendungen werden in einer Privilegierungsstufe, dem Benutzermodus, ausgeführt, in der die Ausführung bestimmter privilegierter Prozessorbefehle nicht möglich ist. Ein Wechsel in den privilegierten Kernelmodus, zum Beispiel zur Ausführung eines Systemaufrufs, ist nur über Prozessorbefehle möglich, die in eine höhere Privilegierungsstufe wechseln und anschließend den Programmablauf an definierten Einsprungadressen fortsetzen. Zur Erhaltung der Integrität eines Betriebssystems muß sichergestellt werden, daß an diesen Adressen nur vertrauenswürdiger Code des Betriebssystems selbst installiert ist. Bevor das Betriebssystem die Kontrolle an eine Anwendung zurück gibt, muß die Privilegierungsstufe wieder zurückgesetzt werden.

Ein Hauptunterscheidungsmerkmal zwischen den im folgenden vorgestellten Betriebssystemarchitekturen besteht in den Grenzen, die sie ziehen. Mit anderen Worten: Welche Teile der Systemsoftware laufen im geschützten Kernelmodus ab, welche im Benutzermodus.

3.1.1 Monolithische Betriebssysteme

Der Begriff des monolithischen Betriebssystems ist in der Kapselung des Betriebssystems in einem Adreßraum im privilegierten Modus begründet. Der geschützte Betriebssystemkern implementiert mit der Speicher- und Prozeß- bzw. Threadverwaltung, den Gerätetreibern sowie dem Dateisystem den Netzwerkdiensten und anderen Elementen alle grundlegenden Systembestandteile und bietet eine einheitliche Systemaufrufschnittstelle über einen Trap-Mechanismus an. Eine Anwendung, die einen Systemaufruf ausführt, muß über spezielle Befehle in den privilegierten Modus und den Kernadreßraum wechseln (Trap). Im allgemeinen werden eine Systemaufrufnummer und gegebenenfalls Aufrufparameter über Register übergeben. Der Systemaufrufhandler des Betriebssystemkerns leitet den Aufruf abhängig von der Systemaufrufnummer an die entsprechende Systemroutine weiter. Aufgrund des monolithischen Systemadreßraums ist dies sehr effizient über Funktionsaufrufe möglich. Lediglich der Adreßraumwechsel jeweils an Beginn und Ende des Systemaufrufs führt zu einem nicht unerheblichen Overhead.

Gerade diese monolithische Struktur ist jedoch einer der Gründe, die zu dem in Kapitel 2 ausgeführten Problem des “Betriebssystems als Flaschenhals” führen. Modifikationen oder Erweiterungen sind in diesen großen, sehr komplexen Programmsystemen nur sehr schwer möglich. Selbst wenn der Betriebssystemkern modular aufgebaut und gut strukturiert ist, sind Modifikationen und Erweiterungen im allgemeinen nur durch den Hersteller möglich, da die Verfügbarkeit des Quellcodes notwendig ist. Ergebnisse beispielsweise von universitären Forschungsprojekten können einerseits aufgrund der fehlenden Verfügbarkeit des Quellcodes nicht verbreitet werden, andererseits fließen sie nicht oder nur sehr langsam in die offiziell vertriebenen Versionen ein. Eine Möglichkeit, monolithische Systeme flexibler zu gestalten, besteht in compilierten Objektmodulen, die zum Betriebssystemkern gebunden werden oder, um einen Neustart des Systems zu vermeiden, dynamisch geladen und wieder entfernt werden können. Für solche Module müssen jedoch Schnittstellen angeboten werden. Die häufigste Anwendung sind Module für Gerätetreiber, da diese oft von den Peripherieanbietern vertrieben werden bzw. ein zu starkes Aufblähen der Systeme verhindert werden soll, indem nur die notwendigen Treiber installiert werden. Einige Systeme ermöglichen auf diese Art die Integration zusätzlicher Systemaufrufe. Hierfür ist jedoch eine tiefgehende Kenntnis der internen Struktur notwendig. Da solche Sy-

stemerweiterungen nur bedingt in andere Bereiche des Systemkerns eingreifen können, sind sie für kleine, vom restlichen System unabhängige Erweiterungen geeignet. Objektmodule können und dürfen nur mit Administratorrechten installiert oder geladen werden, da sie im Kernadreßraum ausgeführt werden und Zugriff auf alle Datenstrukturen des Betriebssystemkerns haben.

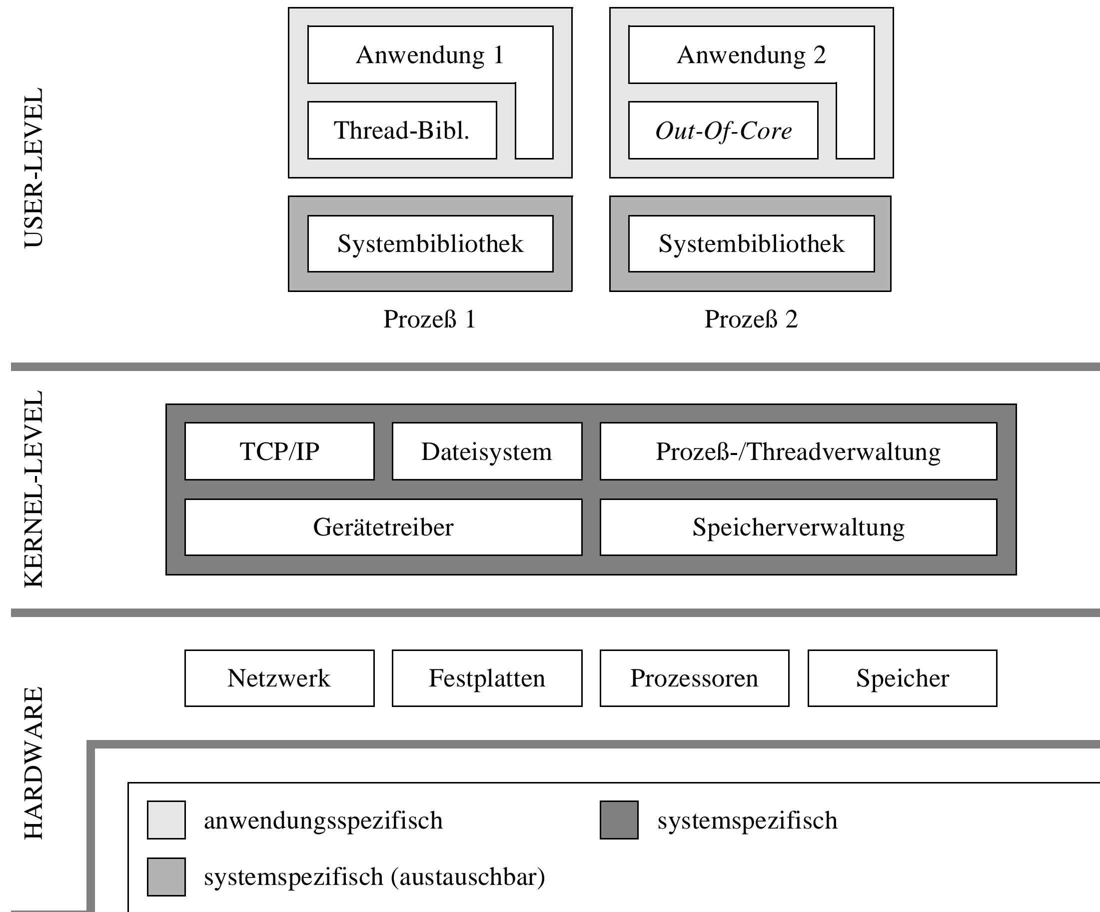

Abb. 3.1: Beispielszenario für monolithische Betriebssysteme (UNIX)

In Abbildung 3.1 wird die Struktur eines monolithischen Betriebssystems am Beispiel von UNIX dargestellt. Die Schnittstellen zu Hardware, Speicher- und Prozeßverwaltung sowie Dateisystem und Netzwerkdiensten sind integraler Bestandteil des Betriebssystemkerns. Anwendungen werden innerhalb eines eigenen virtuellen Adreßraums (Prozeß) im Benutzermodus ausgeführt. Die Nutzung von Systemdiensten erfolgt über einen Transfer des Kontrollflusses in den geschützten Systemkern. UNIX stellt systemnahe Funktionsbibliotheken zur Verfügung, die eine der verwendeten Programmiersprache angepaßte Schnittstelle dem Systemaufrufmechanismus (Trap) sowie weitere auf den Systemdiensten aufbauende höhere Abstraktionen anbieten. Diese Bibliotheken sind systemspezifisch, können aber durch den Benutzer erweitert oder ersetzt werden.

Für jede der hier vorgestellten Betriebssystemarchitekturen wird die Umsetzung des gleichen Szenarios erörtert. Es sollen zwei Anwendungen mit unterschiedlichen speziellen Systemanforderungen ausgeführt werden. Die erste Anwendung benötigt ein bestimmtes, in diesem Rahmen

nicht näher betrachtetes Threadscheduling, die zweite Anwendung hat einen sehr großen Speicherbedarf und muß als *out-of-core*-Anwendung ausgeführt werden. Es wird davon ausgegangen, daß sowohl das Standardscheduling des Systems ungeeignet als auch die Implementierung der virtuellen Speicherverwaltung für *out-of-core*-Anwendungen nicht effizient genug ist. In diesem Abschnitt werden lediglich Lösungen besprochen, die ohne Modifikationen oder Erweiterungen des Grundsystems möglich sind. Auf Lösungsansätze, die an Betriebssystemmodifikationen anknüpfen, wird im weiteren Verlauf der Arbeit eingegangen.

Geht man davon aus, daß am Betriebssystemkern selbst keine Erweiterungen möglich sind, besteht die einzige Möglichkeit, neue, erweiterte oder veränderte Dienste anzubieten, darin, zusätzliche (systemnahe) Funktionsbibliotheken anzubieten oder vorhandene zu ersetzen. Die Lösung bezüglich der ersten Anwendung besteht in einem auf den Kernel-Threads des Betriebssystems aufbauenden Threadscheduling auf Benutzerebene; allerdings mit den in Abschnitt 2.2.2.3 angesprochenen Problemen, die in einer ungenügenden Einbindung in das Gesamtsystem begründet sind. Im Fall einer *out-of-core*-Anwendung kann eine Lösung nur darin bestehen, auf ein Ausnutzen der Abstraktion des virtuellen Speichers zu verzichten und das Ein- und Auslagern der benötigten Daten explizit zu programmieren. Dies ist entweder über das Dateisystem oder durch blockweise Speicherung der Daten auf reservierten Plattenspeicherbereichen möglich. Diese Lösung ist nicht mehr transparent, da vor einem Datenzugriff die richtigen Daten, gesteuert durch die Anwendung, in den Hauptspeicher eingelagert werden müssen. Es können aber Abstraktionen in Form von Bibliotheken angeboten werden, die die Verwaltung sowie die Ein- bzw. Auslagerung der Daten vereinfachen.

3.1.2 Mikrokern-Betriebssysteme

Bei den auf Mikrokernen basierenden Betriebssystemarchitekturen ergibt sich ein geteiltes Bild. Einerseits sind flexiblere Konfigurationen denkbar, die in Zusammenhang mit dem allgemeinen Aufbau dieser Betriebssysteme erläutert werden. Andererseits ergibt sich in der Praxis eine Situation, die mit der monolithischer Systeme durchaus vergleichbar ist. Das konkrete Beispiel einer UNIX-Implementierung auf Basis von MACH wird diese Analogie verdeutlichen.

Der Mikrokern selbst ist mit dem Kern einer monolithischen Betriebssystemarchitektur vergleichbar. Er wird in einem eigenen Adreßraum im privilegierten Modus ausgeführt. Der Mechanismus des Systemaufrufs ist aufgrund der gleichen Hardwaremechanismen analog implementiert. Der wesentliche Unterschied im Vergleich zum monolithischen Betriebssystem liegt im Umfang der im Kern implementierten Dienste. Der Mikrokern sollte nur die unbedingt notwendigen Basisabstraktionen anbieten, sollte kompakt sein und effiziente Kommunikationsmechanismen zur Interaktion mit den im außerhalb des Kerns realisierten höheren Betriebssystemdiensten zur Verfügung stellen. Die Basismechanismen umfassen im wesentlichen die Speicherverwaltung, Dienste zur Manipulation von Tasks und Threads sowie Gerätetreiber. Ein weiterer Dienst, der insbesondere aufgrund der Kommunikation zwischen einzelnen Betriebssystembestandteilen an Bedeutung gewinnt, ist die Interprozeßkommunikation (IPC[3]). Darüber hinaus muß ein Mikrokern ein möglichst allgemeines Sicherheits- und Rechtekonzept anbieten, auf das

3. **I**nter**p**rocess **C**ommunication

die entsprechenden Konzepte der im Benutzermodus realisierten Betriebssysteme abgebildet werden können.

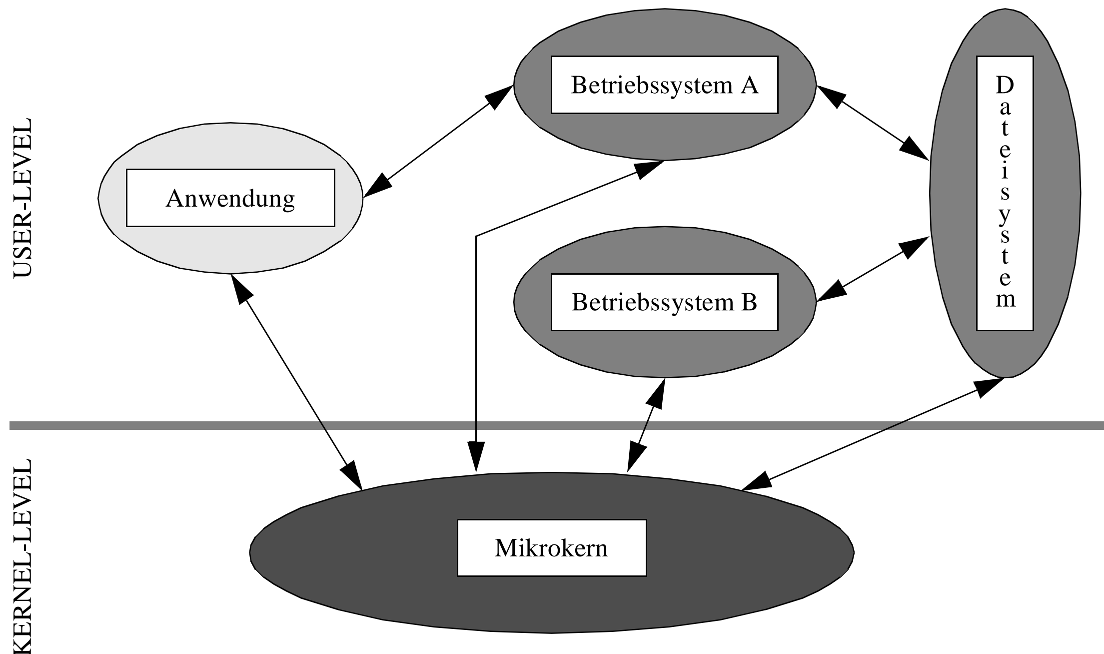

Abb. 3.2: Mikrokern-Betriebssystem (allgemeine Architektur)

Abbildung 3.2 zeigt eine denkbare Konfiguration eines Mikrokernsystems, das zwei verschiedene Betriebssysteme als Benutzerprozesse implementiert. Das Dateisystem ist als eigenständiger Serverprozeß implementiert und wird von beiden Betriebssystemen genutzt, um eine gemeinsame Sicht auf die Benutzerdaten zu ermöglichen. Die Verbindungen zwischen den einzelnen Prozessen symbolisieren die Kommunikationsstrukturen innerhalb des Gesamtsystems. Die Betriebssystemserver A und B kommunizieren jeweils mit dem Dateisystem und dem Mikrokern; das Dateisystem benötigt Dienste des Mikrokerns, um auf die ihm zugeordneten Geräte (Plattenspeicher) zugreifen zu können. In diesem Beispiel ist ein Anwendungsprozeß mit dem Betriebssystem A assoziiert und verfügt folglich über eine Kommunikationsbeziehung zu dem entsprechenden Serverprozeß. Eine direkte Kommunikation mit dem Dateisystem ist in diesem Fall nicht vorgesehen, wäre aber aus Effizienzgründen sicher sinnvoll. Eine direkte Kommunikation zwischen einem Anwendungsprozeß und dem Mikrokern ist, abgesehen von der Verwendung der IPC-Dienste zur Kommunikation mit dem Betriebssystemserver, im allgemeinen nicht unbedingt notwendig. Sie kann aber aus verschiedenen Gründen sinnvoll bzw. Voraussetzung für spezielle Anforderungen sein. Aus Gründen der Effizienz kann der direkte Aufruf von Mikrokerndiensten ermöglicht werden, solange dadurch Betriebssystemserver nicht umgangen werden können. Ein weiterer Grund für direkte Mikrokernaufrufe liegt in der Realisierung von Systemaufrufen bezüglich der Betriebssystemserver. Ein Anwendungsprozeß kann Dienste eines Betriebssystemservers direkt oder indirekt aufrufen:

- Systemaufrufe können direkt über IPC-Nachrichten an einen Betriebssystemserver gesendet werden. In diesem Fall wird nur der IPC-Mechanismus des Mikrokerns benötigt.

- Alle Systemaufrufe werden vom Mikrokern ausgewertet. Falls ein Aufruf nicht für den Mikrokern selbst bestimmt ist, wird er an den entsprechenden Serverprozeß weitergeleitet (*systemcall redirection*), indem der Mikrokern
  - die Ablaufkontrolle an einen speziellen Teil des Anwendungsprozesses, den Emulator [GDFR90], zurück gibt, der dann eine IPC-Nachricht an den Serverprozeß sendet, oder
  - selbst eine IPC-Nachricht an den Serverprozeß sendet.

Indirekte Aufrufe über den Mikrokern sind u.a. dann notwendig, wenn ein auf Basis eines Mikrokerns implementiertes Betriebssystem binärkompatibel zu einer direkt auf der Hardware implementierten Version sein soll. Da das unveränderte Binärprogramm zur Ausführung eines Systemaufrufs einen Trap zum Kontrolltransfer in den privilegierten Modus ausführt und dadurch den Systemaufrufhandler des Mikrokerns aufruft, muß der Mikrokern den Aufruf auf eine der beiden beschriebenen Methoden weiterleiten. Eine Alternative wäre die Ausnutzung von dynamisch gebundenen Systembibliotheken, um die Systemaufschnittstelle anzupassen und direkt IPC-Nachrichten an den Betriebssystemserver zu senden. Bei einer solchen Lösung sind statisch gebundene Programme allerdings nicht mehr unter der Mikrokernversion des Betriebssystems ausführbar.

Auf Mikrokernen basierende Betriebssystemarchitekturen sind aufgrund der Realisierung höherer Systemdienste im Benutzermodus deutlich flexibler als monolithische Betriebssysteme. Untersucht man ihre Eigenschaften jedoch bezüglich der aus den Ausführungen in Kapitel 2 abgeleiteten Forderungen nach Flexibilität, Anpaßbarkeit und Erweiterbarkeit, werden einige Einschränkungen deutlich.

Ein Mikrokern bildet bezüglich der Anpaßbarkeit und Erweiterbarkeit die gleichen Grenzen wie der Kern eines monolithischen Betriebssystems. Im Benutzermodus realisierte Systemdienste sind durch die Funktionalität der Basisdienste eingeschränkt, insbesondere dann, wenn Basisdienste an spezielle Anforderungen angepaßt oder erweitert werden sollen. Kapitel 2 hat jedoch auch gezeigt, daß gerade die generellen Lösungen für grundlegende Dienste wie Scheduling und Speicherverwaltung, also die im Mikrokern realisierten Basisdienste, zu Problemen führen. Trotz der Einschränkungen bezüglich der Basisdienste sind sinnvolle Lösungen für die beiden Beispielfälle auf Ebene der Betriebssystemserver denkbar. Abhängig von dem eingesetzten Mikrokern ist eine unterschiedlich tiefe Einflußnahme auf die Basisdienste Scheduling und Speicherverwaltung möglich. Gegenüber einer Lösung in Form von Bibliotheken kann zumindest eine bessere Integration neuer Dienste erreicht werden. Eine zweite Lösung für den *out-of-core*-Fall wäre ein spezielles Dateisystem, das auf die Anforderungen solcher Anwendungen zugeschnitten ist.

Die Flexibilität eines mikrokern-basierten Systems ist um so größer, je modularer das Gesamtsystem aufgebaut ist. Es liegt also nahe, nicht alle außerhalb des Mikrokerns zu realisierenden Dienste in einem Serverprozeß zu implementieren, sondern mehrere Server für funktional trennbare Teilbereiche zu nutzen (Multiserver-Implementierung). Hier wird jedoch durch die im Vergleich zu Funktionsaufrufen geringere Effizienz der Interprozeßkommunikation eine Grenze

gezogen. Eine zu feine Modularisierung auf Basis der Prozeßabstraktion kann zu einem nicht mehr vertretbaren Betriebssystemoverhead führen.

Betrachtet man reale Implementierungen von mikrokern-basierten Betriebssystemen, zeigen sich noch weitergehende Einschränkungen. Am Beispiel der weit verbreiteten, auch kommerziell genutzten Kombination "UNIX auf Basis von MACH" läßt sich zeigen, daß bezüglich des Beispielszenarios kein Unterschied zu der Realisierung auf Basis eines monolithischen Betriebssystems zu erkennen ist (siehe Abbildung 3.3). Das Betriebssystem wird zwar aus dem Mikrokern und ein oder mehreren Servern gebildet, stellt jedoch ein derart starres Gesamtsystem dar, daß die gesuchten Lösungen wiederum nur auf Anwendungsebene in Form von Bibliotheken möglich sind.

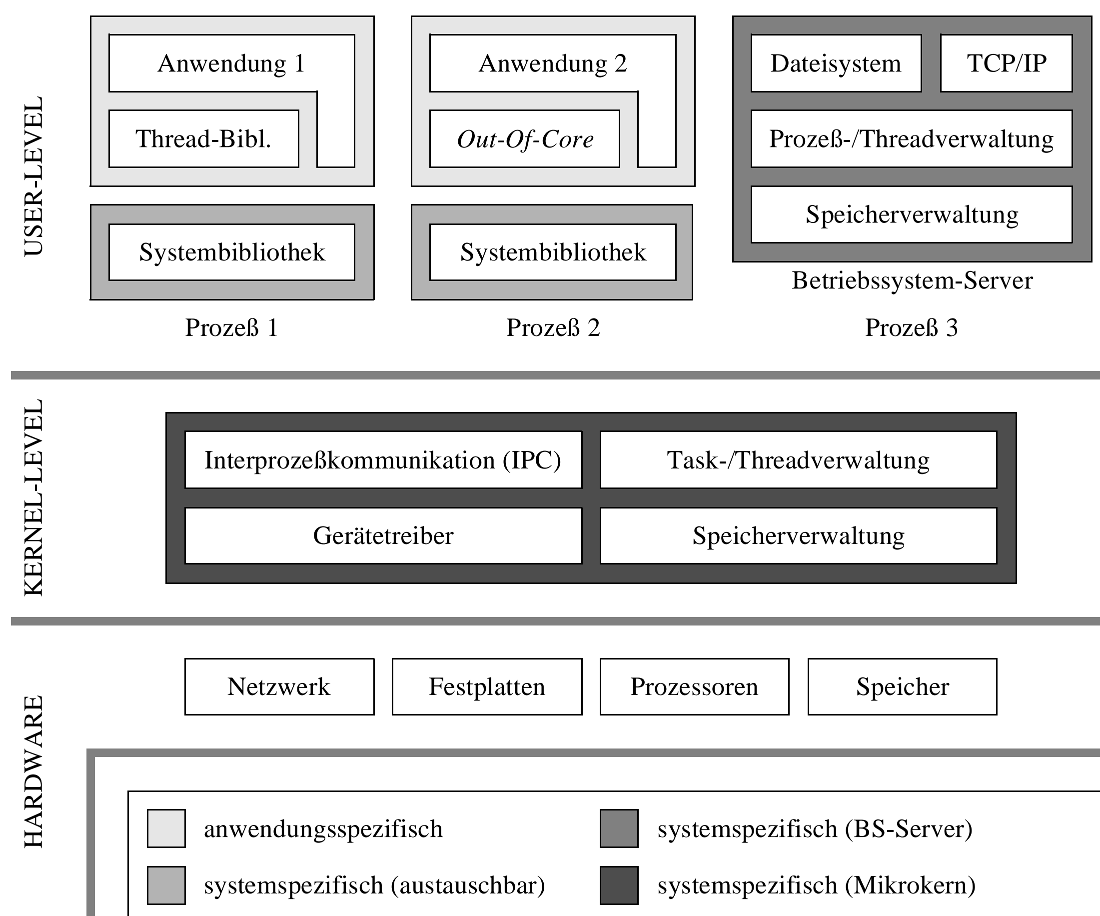

Abb. 3.3: Beispielszenario für Mikrokern-Betriebssysteme (UNIX auf Basis von MACH)

Gründe für diese der Intention von mikrokern-basierten Betriebssystemen eigentlich entgegenlaufende Entwicklung lassen sich in mehreren Bereichen finden. Existierende Betriebssysteme wie UNIX, die auf Basis eines Mikrokerns portiert werden, sind oft nicht entsprechend der neuen Plattform einem Redesign unterworfen, sondern lediglich soweit wie nötig angepaßt. Dadurch entstehen große monolithische Server, die über keine Schnittstellen zur Interaktion mit weiteren Servern (z.B. weiteren Dateisystemtypen) verfügen. Auch wenn Betriebssysteme in Form von mehreren Servern implementiert sind, liegen die Schnittstellen oft nicht offen. Her-

steller von Rechensystemen sind auch in den meisten Fällen überhaupt nicht an einer Erweiterbarkeit ihrer Systeme interessiert, sondern nutzen Mikrokerne aus anderen Gründen als Implementationsbasis, beispielsweise als eine besser geeignete Plattform für NUMA-Systeme. Weitere Probleme können aufgrund von Maßnahmen zur Verbesserung der Effizienz entstehen. Mikrokerne werden an die auf ihnen aufbauenden Betriebssysteme angepaßt oder sogar in grundlegenden Bereichen modifiziert. Im Fall des in Abschnitt 5.2.2.1 näher betrachteten SPP-UX sind diese Anpassungen sehr weitgehend und betreffen selbst die Kommunikationsmechanismen des Mikrokerns und die Privilegierungsstufe der Server, die nicht mehr im Benutzermodus ausgeführt werden. Eine Instanziierung weiterer Server ist z.B. bei vielen MACH/UNIX-Systemen ohne Änderungen im Quellcode überhaupt nicht möglich, da der Start der Server fest in den Initialisierungsroutinen des Systems kodiert ist und Funktionen zur nachträglichen Instanziierung fehlen.

Neben dem Beispiel MACH, als einem der ersten Mikrokerne, soll noch ein zweiter neuerer Mikrokern genannt werden. Der an der Gesellschaft für Mathematik und Datenverarbeitung (GMD) entstandene und am IBM T. J. Watson Research Center fortentwickelte extrem kompakte Mikrokern L4 [Liedtk96] konzentriert sich auf die ursprünglichen Entwurfsziele von Mikrokernarchitekturen. Durch eine sehr effiziente Interprozeßkommunikation und wenige sehr allgemeine Basisdienste werden die Einschränkungen bezüglich eines modularen Aufbaus des Gesamtsystems und der Erweiterbarkeit aufgeweicht. Die prinzipiellen Einschränkungen bleiben jedoch erhalten und die Effizienz des L4 muß durch eine sehr hardwareabhängige Implementierung des gesamten Mikrokerns in Maschinensprache erkauft werden. Eine Portierung des L4 auf andere Architekturen und Erweiterungen um eine beispielsweise in der aktuellen Version noch nicht vorhandene Multiprozessorunterstützung werden dadurch sehr erschwert.

3.1.3 Betriebssysteme in der aktuellen Forschung

Die Forderungen nach Flexibilität und Anpaßbarkeit von Betriebssystemen, insbesondere an anwendungsspezifische Anforderungen, können auf Basis der bisher betrachteten Architekturen nicht oder nicht im genügenden Umfang erfüllt werden. Betrachtet man neuere Entwicklungen in der aktuellen Betriebssystemforschung, lassen sich zwei Themenbereiche erkennen, mit denen sich ein Großteil der Forschungsarbeiten zur Architektur von Betriebssystemen befaßt. Die Objektorientierung stellt einen dieser keinesfalls überschneidungsfreien Bereiche dar, der im Rahmen dieser Arbeit jedoch nicht weiter betrachtet werden soll. Den zweiten Bereich bilden die erweiterbaren Betriebssysteme. Mit der Exokernel-Architektur und dem Betriebssystem SPIN werden im folgenden zwei Systeme mit sehr unterschiedlichen Lösungsansätzen als Beispiele für diesen Forschungsbereich vorgestellt.

3.1.3.1 Exokernel

Im Fall der am MIT entwickelten Exokernel-Architektur handelt es sich nicht um ein konkretes Betriebssystem, sondern um einen neuen Ansatz für anwendungsspezifisch anpaßbare Betriebssysteme. Mit Aegis/ExOS (MIPS), Glaze/PhOS (SPARC) und Xok/ExOS (Intel x86) wurden

auf der Exokernel-Architektur aufbauende Systeme für unterschiedliche Prozessorarchitekturen entwickelt.[EnKaTo95 und KEGB+97]

Die Exokernel-Architektur führt die erstmals von dem Betriebssystem Hydra [WCCJ+74] durchgehend verwendete Idee der Trennung von Mechanismus und Strategie (*separation of mechanism and policy*) konsequent fort. Der Mechanismus wird als Bestandteil der Strategie angesehen und die Trennung erfolgt zwischen dem Schutz von Systemressourcen (*protection*) und deren Verwaltung (*management*) auf einer möglichst hardwarenahen Ebene. Der geschützte Kern einer Exokernel-Architektur implementiert lediglich den Schutz und die Verteilung der Systemressourcen, alle darüberhinausgehenden Betriebssystemdienste werden in Form von Bibliotheken (*library operating system*) realisiert. Jede Anwendung kann ein eigenes Bibliotheksbetriebssystem verwenden.

Je tiefer die Ebene der Rechteüberprüfung angesetzt wird, um so häufiger muß ein Exokernel Zugriffsrechte der laufenden Anwendungen überprüfen, was zu einem beträchtlichen Overhead führen würde. Um dies zu vermeiden, werden sogenannte *secure bindings* eingesetzt. Komplexere Rechteüberprüfungen werden nur zum Zeitpunkt der Bindung durchgeführt, um für eine folgende mehrfache Nutzung auf einfache und effiziente Schutzmechanismen durch Hardware oder den Kern abgebildet zu werden. Ein einfaches Beispiel für ein *secure binding* ist die Abbildung des Zugriffsschutzes auf den Speicherschutzmechanismus der Hardware. Zugriffsrechte beispielsweise für einen Festplattenblock müssen nur zum Zeitpunkt der Abbildung des Blocks auf eine Speicherseite überprüft werden. Die Überprüfung weiterer Zugriffe ist bis zur Auflösung der Bindung nicht mehr notwendig.

Neben dem Schutz der Ressourcen muß ein Exokernel eine faire Verteilung zwischen den Anwendungen bzw. Prozessen gewährleisten. Alle Ressourcen sind explizit durch einen Prozeß anzufordern und werden ebenfalls durch den Prozeß selbst wieder freigegeben. Dadurch liegt die Kontrolle beispielsweise darüber, welcher Prozessor oder welche Speicherseite freigegeben wird, bei der Anwendung bzw. bei dem an die Anwendung gebundenen Bibliotheksbetriebssystem. Der Exokernel selbst stellt nur Freigabeaufforderungen für einen bestimmten Ressourcentyp. Lediglich wenn ein fehlerhaftes Bibliotheksbetriebssystem einer Freigabeaufforderung nicht nachkommt, greift der Exokernel abhängig von der Implementierung ein und beendet den betroffenen Prozeß oder entzieht die benötigten Ressourcen selbständig.

Die entscheidende Fragestellung beim Entwurf eines Exokernels liegt in der Festlegung der Ebene, auf der die Zugriffskontrolle erfolgt, und damit in der Grenzziehung zwischen Kern und Bibliotheksbetriebssystem. Die unterste Ebene, auf der ein Ressourcenschutz durch den Kern noch möglich wird, ist nicht in jedem Fall sinnvoll, wie das dritte der folgenden Beispiele zeigt:

- Virtueller Speicher:
  In diesem Fall ist der Ressourcenschutz direkt auf die Speicherschutzmechanismen des Prozessors abbildbar. Der Exokernel stellt dem Bibliotheksbetriebssystem auf Anforderung physikalische Speicherseiten durch ein entsprechendes Speicher-Mapping zur Verfügung. Durch die Weiterleitung von Seitenfehlern an das Bibliotheksbetriebssystem kann die fehlende Seite im Benutzermodus in eine freie Speicherseite geladen werden. Das geänderte Speicher-Mapping wird über einen Systemaufruf an den Exokernel einge-

tragen. Benötigt der Exokernel physikalischen Speicher für andere Anwendungen, stellt er eine Freigabeanforderung. Es bleibt der Speicherverwaltung auf Benutzerebene überlassen, welche Seite freigegeben wird.

- Prozessorverwaltung und Scheduling:
  Die Prozessorverwaltung ist in erster Linie keine Frage des Ressourcenschutzes, sondern der fairen Verteilung zwischen den laufenden Anwendungen. Das Scheduling des Exokernels basiert auf Zeitscheiben (processor slices), die dem Bibliotheksbetriebssystem durch den Exokernel zugeteilt und wieder entzogen werden. Das Restaurieren und Sichern des Kontextes zu Beginn bzw. am Ende einer Zeitscheibe wird innerhalb des Bibliotheksbetriebssystems realisiert. Um Fairness garantieren zu können, setzt der Exokernel dem Bibliotheksbetriebssystem eine Zeitgrenze, innerhalb der ein Prozessor freigegeben werden muß. In einer Multiprozessorimplementierung wäre es denkbar, daß ein Bibliotheksbetriebssystem entscheiden kann, welchen von mehreren Prozessoren es auf Anforderung freigibt. Zeitscheiben müssen explizit vom Exokernel angefordert werden und können an andere Prozesse bzw. Threads weitergereicht werden.

- Massenspeicher und Dateisysteme:
  Ein naheliegender Ansatz für blockorientierte Massenspeicher wie Festplatten wäre eine Rechteverwaltung auf Blockebene. Eine solche Sicht kann für Plattenbereiche, die beispielsweise zum Paging verwendet werden, noch sinnvoll sein, birgt aber auch die Gefahr eines erheblichen Overheads, da die Zugriffsrechte getrennt von den Blöcken gespeichert werden müssen. Als Basis für Dateisysteme auf Benutzerebene ist diese sehr hardwarenahe Abstraktionsebene nicht geeignet. Ohne eine Unterstützung des Kerns, die über eine Rechteverwaltung auf Blockebene hinausgeht, sind effiziente Dateisysteme, die einen parallelen Dateizugriff durch unabhängige Dateisysteminstanzen erlauben, nicht realisierbar. Die von Kaashoek et al. [KEGB+97] realisierte Lösung lädt einen für jeden neu definierten Dateityp interpretierten Code in den Kern, über den im wesentlichen Zugriffsrechte und zugehörige Blöcke einer Datei bestimmt werden können. Der Kern ist dadurch in der Lage, in einfacher Weise Zugriff auf genau die Blöcke einer Datei zu erlauben, und kann, ohne den strukturellen Aufbau eines Dateityps zu kennen, die Konsistenz von Änderungen an der Dateistruktur überprüfen. Eine solche ladbare Beschreibung der Dateistruktur wird ebenfalls als *secure binding* bezeichnet, da sie u.a. die Abbildung komplexer Zugriffsrechte auf den einfach zu realisierenden Zugriffsschutz von Festplattenblöcken ermöglicht. Weitere Dateizugriffe sind ohne Interaktion mit dem Exokernel möglich, solange die Abbildung zwischen Festplattenblöcken und Speicherseiten erhalten bleibt und an der Dateistruktur keine Änderungen vorgenommen werden.

Die Verwaltung blockorientierten Massenspeichers wird von Kaashoek et al. unter anderem deshalb sehr ausführlich, inklusive der gescheiterten Ansätze, beschrieben, weil sie ein sehr gutes Beispiel für die Komplexität des Problems ist, eine sinnvolle Schnittstelle zwischen Exokernel und Bibliotheksbetriebssystem zu finden. Eine gut entworfene Exokernel-Architektur stellt eine gangbare Basis für die Anforderungen nach Flexibilität und Anpaßbarkeit dar.

Eine denkbare Lösung für das Beispielszenario wird in Abbildung 3.4 dargestellt. Der Exokernel bildet die Schnittstelle zwischen Hardware und Bibliotheksbetriebssystemen. Der Kern stellt die Verteilung und den Schutz der Ressourcen sicher und ermöglicht eine effiziente Nutzung über *secure bindings*. Standardlösungen für Betriebssystemdienste werden in Form einer Bibliothek angeboten, können jedoch vollständig oder teilweise durch andere Lösungen ersetzt werden. *Out-of-core*-Anwendungen können eine Speicherverwaltung mit speziell angepaßten Ein- und Auslagerungsstrategien verwenden. Ein anpaßbares Thread-Scheduling kann auf der Möglichkeit, Zeitscheiben weiterzureichen, aufgebaut werden. Anstatt eine Zeitscheibe direkt vom Kernel anzufordern, kann sie bei einem Schedulerprozeß oder Thread angefordert werden, der seinerseits mehrere Zeitscheiben vom Kernel anfordert und diese weiterverteilt.

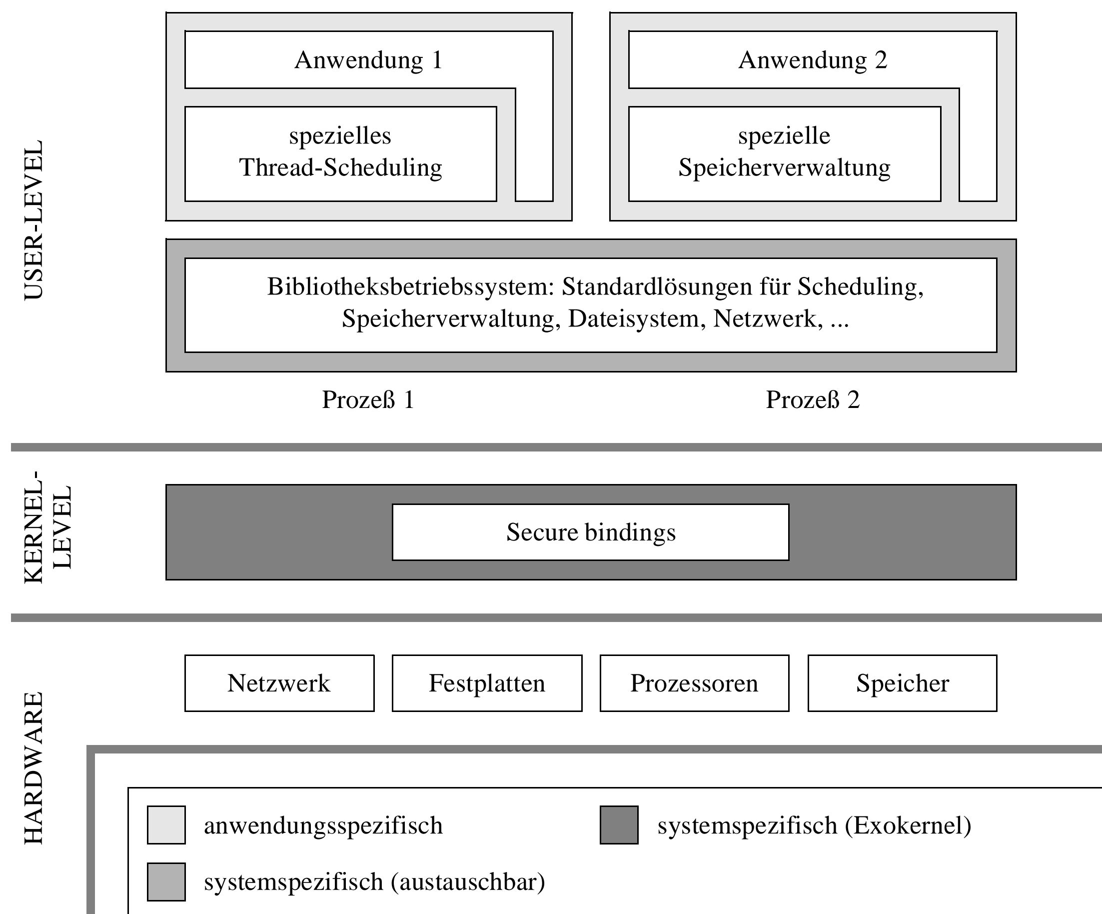

Abb. 3.4: Beispielszenario für Exokernel-Betriebssysteme

3.1.3.2 SPIN

Im Gegensatz zu der Exokernel-Architektur, die die Trennlinie zwischen Kern- und Benutzermodus verschiebt und im Betriebssystemkern im wesentlichen nur den Zugriffsschutz auf Systemressourcen realisiert, implementiert das erweiterbare Betriebssystem SPIN [BSPS+95] weiterhin alle grundlegenden Betriebssystemdienste im privilegierten Betriebssystemkern. Um eine Erweiterbarkeit des Systems und eine Anpassung an Anwendungsanforderungen zu ermöglichen, erlaubt SPIN das Laden von nicht vertrauenswürdigem Benutzercode in den Be-

triebssystemkern, wo er mit Systemprivilegien ausgeführt wird. Während beim Entwurf eines Exokernel eine sinnvolle Trennlinie zwischen Ressourcenschutz und Ressourcenverwaltung gefunden werden muß, ist bei einem erweiterbaren Betriebssystem wie SPIN sicherzustellen, daß mit Systemprivilegien ausgeführter Benutzercode die Systemintegrität nicht verletzen kann.

SPIN ist zum größten Teil in der Programmiersprache Modula-3 [CDGJ+92] geschrieben. Da SPIN das Laden von zusätzlichen Modulen durch nicht privilegierte Benutzer nur akzeptiert, wenn zuvor die Module durch einen vertrauenswürdigen Modula-3 Compiler übersetzt und signiert wurden, kann die Integrität des Systems trotz des mit Systemprivilegien ausgeführten Benutzercodes garantiert werden. Die wichtigsten Eigenschaften von Modula-3, die eine solche Garantie ermöglichen, sind das Modul- und Interface-Konzept sowie die Typsicherheit. Das erste stellt sicher, daß eine Interaktion mit anderen Teilen des Betriebssystems nur über die vom Basissystem angebotenen Schnittstellen möglich ist, das zweite verhindert die Dereferenzierung eines Zeigers für einen nicht kompatiblen Datentyp und damit unzulässige Speicherzugriffe. Diese durch Sprachkonzepte realisierten Sicherheitsmechanismen können schon bei der Übersetzung durch den Compiler überprüft werden, zur Laufzeit ist lediglich eine Überprüfung der Feldgrenzen bei berechneten Indizes nötig. Eine weitere wichtige Eigenschaft von Modula-3 ist die Speicherverwaltung, die sicherstellt, daß keine Speicherbereiche, die noch durch gültige Zeiger referenziert werden, freigegeben und wiederverwendet werden. Der Entwurf von SPIN stützt sich darüberhinaus auf weitere Eigenschaften von Modula-3. Unter anderem sind dies Modula-3 Threads und die Ausnahmebehandlung.

Eine Erweiterung in SPIN wird durch eine Behandlungsroutine (Handler) für ein bestimmtes Ereignis (Event) realisiert. Ein Event kann beispielsweise ein Seitenfehler, aufgrund dessen in einen Handler des Speicherverwaltungssystems verzweigt wird, oder ein programmgesteuerter Trap zur Realisierung eines Systemaufrufs sein. Eine Systemerweiterung wird installiert, indem ein neuer oder zusätzlicher Handler für einen Event eingetragen wird. Ein Handler kann mit einem sogenannten Guard assoziiert werden, der eine Bedingung ausdrückt, unter der der Handler ausgeführt wird. Über das Laden neuer Module und das Installieren von Handlern kann ein laufendes SPIN-System dynamisch erweitert oder modifiziert werden. Da der SPIN-Kernel preemptiv ist, kann ein benutzerinstallierter Handler das System nicht monopolisieren.

Neben den Mechanismen für eine Erweiterbarkeit des Systems müssen Schnittstellen für anpaßbare oder erweiterbare Betriebssystemdienste angeboten werden. Dabei unterscheidet SPIN zwischen verschiedenen Erweiterungsmöglichkeiten. Betriebssystemdienste können durch eine neue Implementierung teilweise oder vollständig ersetzt bzw. um zusätzliche Funktionalität erweitert werden. Aufgrund der globalen Auswirkungen solcher Eingriffe muß diese Art der Systemerweiterung an Systemverwalterprivilegien gebunden sein. Normale nichtprivilegierte Systembenutzer haben die Möglichkeit, anwendungsspezifische Erweiterungen zu installieren. Solche Erweiterungen werden über Guards realisiert, die eine Ausführung des Handlers zum Beispiel nur im Kontext des installierenden Prozesses zulassen und nicht - wie ebenfalls möglich - von dem installierten Modul bereit gestellt, sondern von einem vertrauenswürdigen primären Handler des betroffenen Events installiert werden.

SPIN bietet eine Vielzahl von Schnittstellen zur Erweiterbarkeit und Anpaßbarkeit des Systems. Ein Beispiel stellen die Netzwerkdienste dar. Es können Handler für verschiedene Protokolle installiert werden, deren jeweilige Ausführung über Guards gesteuert wird. Auf einer höheren Ebene sind beispielsweise Guards und Handler für bestimmte TCP- oder UDP-Ports möglich, um extrem effiziente HTTP-Server im Systemkern zu realisieren. Die für das Beispielszenario notwendigen Anpassungen von Speicherverwaltung und Thread-Scheduling können realisiert werden, indem anwendungsspezifische Handler für Events der Speicher- bzw. Threadverwaltung eingetragen werden. Die vertrauenswürdigen primären Handler stellen eine faire Verteilung der Ressourcen sicher und leiten Events an die anwendungsspezifischen Handler weiter.

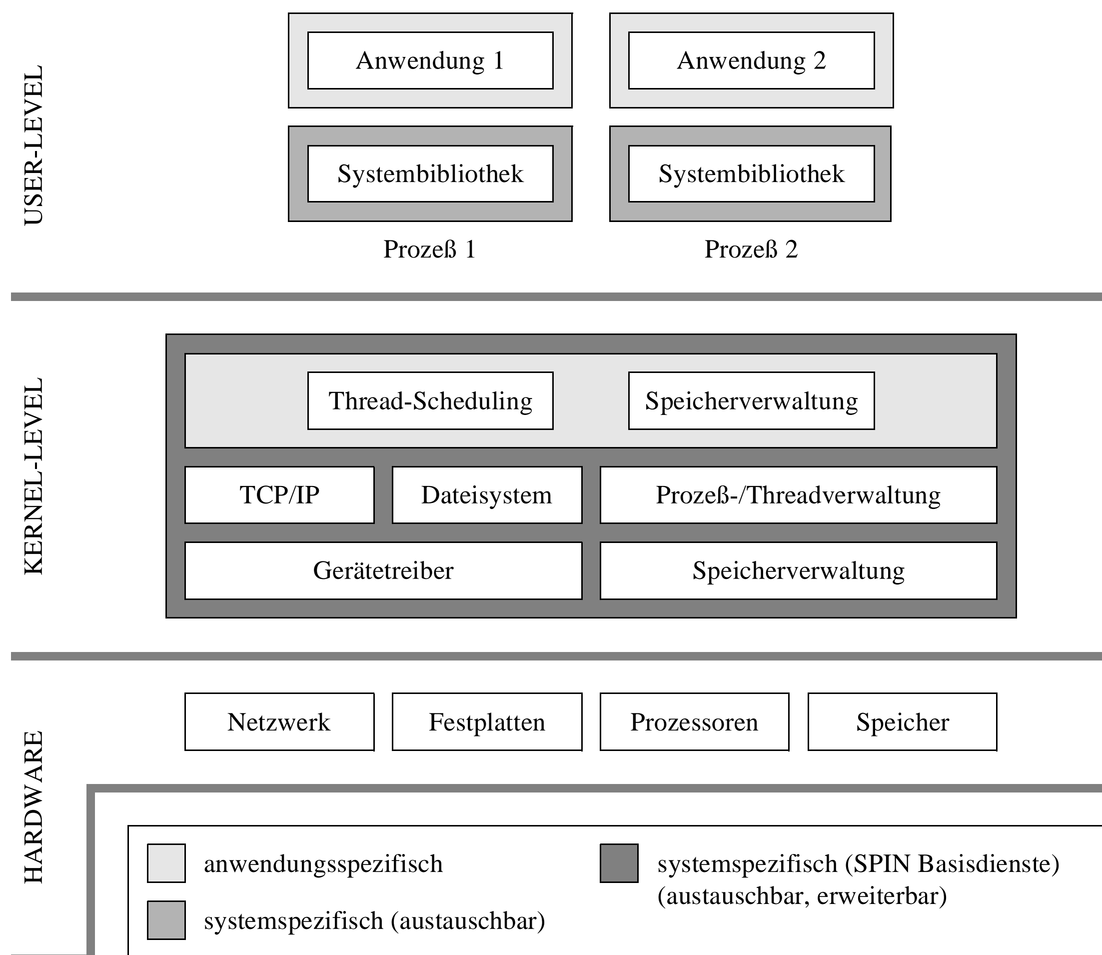

Abb. 3.5: Beispielszenario für erweiterbare Betriebssysteme (SPIN)

Abbildung 3.5 zeigt, wie eine Lösung für das Beispielszenario in einem System wie SPIN aussehen könnte. Auf den ersten Blick zeigt sich eine Ähnlichkeit zu der Struktur des monolithischen UNIX in Abbildung 3.1. Es besteht eine klare Trennung zwischen den Anwendungen im Benutzermodus und dem Betriebssystemkern im privilegierten Kernelmodus, jedoch mit dem Unterschied, daß Benutzercode in den Kern geladen werden kann. Die speziellen Anforderungen der zwei Beispielanwendungen an Threadscheduling und Speicherverwaltung können als anwendungsspezifische Betriebssystemerweiterungen dynamisch geladen werden. Sie werden im Kernadreßraum mit entsprechenden Privilegien ausgeführt. Wie im Fall des Exokernel-Sze-

narios kann die anwendungsspezifische Speicherverwaltung auf Basis der ihr zugeteilten Speicherseiten speziell an *out-of-core*-Anwendungen angepaßte Ein- und Auslagerungsstrategien implementieren. Einem anwendungsspezifischen Scheduler wird vom zentralen Systemscheduler über ein *resume*-Event ein Prozessor übergeben, auf dessen Basis der Scheduler seine Threads ausführen kann. Über ein *checkpoint*-Event signalisiert der Systemscheduler, daß der Prozessor wieder freigegeben werden muß.

3.2 Modernisierung vs. neue Architekturen

Die zwei Beispielanwendungen der Fallstudie gehören einer Klasse von Problemen an, die sich durch spezielle Anforderungen an einzelne Betriebssystemdienste auszeichnen. Sie sind eine ideale Anwendung für Betriebssystemarchitekturen, die eine Anpassung dieser Dienste an Anwendungsanforderungen erlauben. Diese Fragestellungen sind daher für den Teilbereich des ELiTE-Projekts, der sich mit einer Unterstützung paralleler Anwendungen, insbesondere bezüglich des Threadschedulings, durch die Systemsoftware (dem Schwerpunkt dieser Arbeit) befaßt, von großer Bedeutung. Erweiterbarkeit und Adaptierbarkeit spielen folglich bei der Bewertung der vorgestellten Architekturen eine wichtige Rolle. Andere Kriterien für eine Bewertung der Architekturen ergeben sich aus Anforderungen, die sich auf ein globales Verhalten der Systeme beziehen.

3.2.1 Erweiterbarkeit und Adaptierbarkeit

Werden die vier vorgestellten Lösungsansätze für das Beispielszenario betrachtet, dann findet man eine Bestätigung zweier Aussagen des vorausgehenden Kapitels:

- Die rein auf Benutzerebene realisierten Lösungsansätze für UNIX-verwandte Betriebssysteme stellen, unabhängig davon, ob sie auf monolithischen Systemen oder auf Mikrokernarchitekturen basieren, keine befriedigende Lösung dar.
- Erweiterbare Architekturen können die in Kapitel 2 abgeleiteten Forderungen nach einer weitgehenden Adaptierbarkeit erfüllen. Sie bieten generell einsetzbare Standarddienste an, die aufgrund spezieller Anwendungsanforderungen angepaßt, erweitert oder ersetzt werden können.

Die Annahme, daß kommende Betriebssysteme auf solchen oder ähnlich erweiterbaren und adaptierbaren Architekturen basieren werden, liegt folglich nahe. Auch die in Abschnitt 2.3 aufgestellte These, daß weiterhin an einer Modernisierung UNIX-basierter Betriebssysteme, insbesondere um eine bessere Adaptierbarkeit an spezielle Anwendungen zu ermöglichen, gearbeitet werden muß, kann aufgrund der genannten Beispiele erhärtet werden. Zu dem Problem, daß neue Betriebssystemarchitekturen sich gegen das marktbeherrschende UNIX etablieren müssen, kommt hinzu, daß die beiden ausgewählten Beispiele Exokernel und SPIN, die noch dazu zu den in der Fachwelt bekannteren Projekten zählen, erst in den Anfängen stecken. Insbesondere gilt dies, wenn man Multiprozessorsysteme betrachtet. An einer SPIN Implementierung für Intel-basierte symmetrische Multiprozessoren wird gearbeitet [Spin98], es sind jedoch

noch keine Ergebnisse veröffentlicht. Mit Glaze/PhOS liegt eine Exokernel Implementierung für das am MIT entwickelte SPARC-basierte experimentelle Multiprozessorsystem FUGU [MKAK94] vor. Auch über diese Implementation ist keine Veröffentlichung zu finden.

3.2.2 Weitere Kriterien

Ein anderer Bereich des ELiTE-Projekts befaßt sich mit Schedulingstrategien im Betriebssystemkern, die globale Auswirkungen über Anwendungsgrenzen hinweg haben. Im Rahmen der Kategorisierung in Kapitel 2 wurde bereits das Beispiel des *Follow-On* Scheduling genannt, das Verwandschaftsbeziehungen bezüglich der referenzierten Speicherbereiche ansonsten unabhängiger Threads ausnutzt, um eine sinnvolle Ausführungsreihenfolge zu ermitteln. Ein weiteres Beispiel sind Bandbreitengarantien bezüglich des Speicherzugriffs von Anwendungen aus dem Multimediabereich [Bellos97a und Bellos98]. Auch hier muß in den Ablauf anderer laufender Anwendungen eingegriffen werden, um die garantierte Bandbreite einer Anwendung sicherzustellen.

Solche anwendungsübergreifenden Anforderungen an die Systemsoftware können nicht durch anwendungsspezifische Erweiterungen realisiert werden. Da sie globale Auswirkungen haben, kann die Umsetzung nicht im Rahmen einer Anwendung, etwa durch in das System geladenen Benutzercode, erfolgen, sondern muß im Verantwortungsbereich eines vertrauenswürdigen Systemdienstes liegen. Entscheidend ist folglich, wie leicht Basisdienste des geschützten Bereichs der Systemsoftware erweitert, angepaßt oder ausgetauscht werden können. Die beiden vorgestellten erweiterbaren Betriebssysteme zeigen bezüglich dieser Anforderung unterschiedliche Eigenschaften. Die Implementierungen der Exokernel Architektur bieten solche Möglichkeiten nicht. Basisdienste, die beispielsweise die Ressourcenverteilung zwischen verschiedenen Anwendungen realisieren, sind fester Bestandteil des Exokernel und können nur durch die Generierung eines neuen Kerns ersetzt werden. SPIN ermöglicht es dagegen, mit entsprechenden Rechten auch Basismodule zur Laufzeit auszutauschen. Da die Schnittstellendefinition der einzelnen Module jedoch festliegt, sind Erweiterungen auf Basis dieser Eigenschaft auch nur in begrenztem Umfang möglich. Für weitergehende Modifikationen ist auch in diesem Fall die Generierung eines neuen Kerns notwendig.

Für diesen Bereich zu integrierender neuer Betriebssystemmechanismen zeigt der Unterschied zwischen UNIX-basierten Systemen und modernen erweiterbaren Architekturen also nicht in jedem Fall so große Auswirkungen wie im Fall der anwendungsspezifischen Erweiterungen. Man kann jedoch annehmen, daß Erweiterungen in modernen Architekturen einfacher zu realisieren sind, da diese im allgemeinen besser strukturiert und modularer als langsam gewachsene und sehr komplexe Kerne von UNIX-basierten Systemen aufgebaut sind.

3.2.3 Die Rolle heutiger Betriebssystemarchitekturen

UNIX-basierte Betriebssysteme werden für eine noch nicht absehbare Zeit weiterhin im Bereich von Rechensystemen des oberen Leistungsspektrum eine dominante Rolle spielen, da sich bisher noch keine Neuentwicklung abzeichnet, die als Basis für eine ernstzunehmende Konkur-

renz im kommerziellen Bereich dienen könnte. Der heutige Status der Projekte, die sich mit erweiterbaren Betriebssystemen befassen, verdeutlicht die in Abschnitt 2.3 begründete Notwendigkeit, Lösungskonzepte auf Basis von UNIX-Architekturen möglichst weitgehend zu entwikkeln. Ansonsten wird in den nächsten Jahren eine adäquate Ausnutzung der immer weiter steigenden Leistungsfähigkeit moderner Rechnerarchitekturen nicht erreicht werden können. Das Fehlen geeigneter Systemsoftware wird, wie schon heute, durch einen überhöhten Einsatz sehr leistungsfähiger Hardware kompensiert werden[4].

3.3 Zusammenfassung

Moderne erweiterbare Betriebssystemarchitekturen können eine Basis bilden, auf der eine Reihe von in Kapitel 2 angesprochenen Problemen lösbar erscheinen. Diese Systeme haben jedoch den Status von Forschungsprojekten noch nicht überwunden. Auf der anderen Seite stehen kommerziell verfügbare und weit verbreitete monolithische sowie mikrokern-basierte UNIX-Systeme, auf denen, ohne Änderungen am Betriebssystemkern selbst, notwendige Erweiterungen nicht oder nur in unbefriedigendem Umfang realisiert werden können. Andere Aspekte moderner Systemsoftware finden auch in erweiterbaren Architekturen keine Basis und können hier nur als integraler Bestandteil des Betriebssystemkerns realisiert werden.

Erweiterbare Betriebssystemarchitekturen sind besonders gut in Bereichen einsetzbar, in denen spezielle Lösungen für einzelne Anwendungen gesucht sind. Gegenüber heutigen kommerziellen Betriebssystemen bieten sie hingegen kaum Vorteile, wenn die Integration neuer globaler Mechanismen oder Strategien in ein bereits existierendes System gefragt ist.

Die geringe Bedeutung, die Fragestellungen bezüglich Multiprozessoren und paralleler Programmabläufe in Projekten über erweiterbare Betriebssystemarchitekturen bisher haben, läßt befürchten, daß auch hier noch keine zukunftsweisenden Konzepte zu finden sind. Mit speziellen Anforderungen, die moderne Multiprozessorsysteme an die Systemsoftware stellen, befaßt sich das folgende Kapitel.

4. Dieser Effekt ist im übrigen auch im Bereich der Anwendungssoftware zu finden, wo alte Algorithmen und Programme auf immer leistungsfähigerer Hardware ausgeführt werden, statt in eine Reimplementierung gegebenenfalls auf Basis neuer Algorithmen zu investieren, wodurch in vielen Fällen eine Leistungssteigerung im vergleichbaren Umfang möglich wäre.

4 Spezielle Probleme auf Multiprozessorsystemen

Die bisher angesprochenen Probleme bzw. die von heutigen Betriebssystemen nicht erfüllten Anforderungen lassen sich weitgehend in zwei Bereiche einteilen. Zum einen sind dies Fragestellungen aus dem Bereich der Prozeß- bzw. Threadverwaltung sowie des Schedulings, zum anderen der effiziente Umgang mit Speicher und Caches. Beide Themenbereiche sind für Betriebssysteme auf Multiprozessorarchitekturen von noch entscheidenderer Bedeutung. Für den Bereich der Threads liegt dieser Bedeutungszuwachs auf der Hand, da es gerade eines der Ziele von Multiprozessorsystemen ist, eine schnellere Abarbeitung von Anwendungen durch die parallele Ausführung mehrerer Threads zu erreichen.

Im Kapitel 4 wird zunächst dargelegt, daß ein Bedeutungszuwachs auch im Bereich der Speicherverwaltung und der Caches vorliegt. Anschließend werden einige, insbesondere für Multiprozessorsysteme wichtige Fragestellungen bezüglich der Prozeß- bzw. Threadverwaltung erörtert. Mit einer tiefergehenden Diskussion der in Abschnitt 2.2.2.3 angesprochenen Problematik "Kernel-Threads vs. User-Level-Threads" wird eine Grundlage für die Beschreibung der *Sleeping-Threads* im weiteren Verlauf dieser Arbeit geschaffen. Abschließend werden andere Arbeiten aus dem Bereich einer besseren Integration von User-Level-Threads in die Systemsoftware vorgestellt.

4.1 Speicherverwaltung / Caches

Multiprozessoren, insbesondere NUMA-Architekturen, zeichnen sich im allgemeinen durch größere Latenzzeiten beim Speicherzugriff aus. Der Einsatz von Caches kann daher noch stärkere Auswirkungen haben. Aufgrund der Speicherhierarchien von NUMA-Architekturen können bzw. müssen Techniken zur Cache-Verwaltung auf weiteren Ebenen eingesetzt werden, um zum Beispiel den langsamen Zugriff auf entfernte Speicherbereiche anderer Module durch Caching im lokalen Speicher zu vermeiden. Neben diesem allgemeinen Bedeutungszuwachs von Caches aufgrund eines komplexeren Speichersubsystems, wirft der Einsatz von Caches in Multiprozessorsystemen jedoch auch neue Fragestellungen auf.

Die meisten Multiprozessorarchitekturen setzen getrennte Caches für die einzelnen Prozessoren ein. Systeme mit gemeinsamen Caches sind sehr selten und beschränken sich auf geringe Prozessoranzahlen. Dadurch ergeben sich neue Formen von möglichen Cacheinkonsistenzen, da Speicherbereiche gleichzeitig in verschiedenen Caches abgebildet sein können. Die entsprechenden Konsistenzprotokolle sind meist in Hardware realisiert und erfordern in bestimmten Fällen Maßnahmen durch die Systemsoftware. Diese Form der Unterstützung durch das Betriebssystem ist jedoch für Betrachtungen im Rahmen dieser Arbeit nicht von Bedeutung, da sie als notwendige Funktion in allen hier betrachteten Systemen realisiert ist. Interessanter sind Effekte, die die prinzipielle Funktion eines Systems nicht beeinträchtigen, wohl aber entscheidenden Einfluß auf seine Effizienz haben.

Ein konkurrierender Zugriff auf Daten der gleichen Cacheline durch mehrere Prozessoren, von denen zumindest einer schreibend zugreift, kann nicht nur in Anwendungen fatale Folgen für die Effizienz haben. Auch bei Datenstrukturen der Systemsoftware sollte dies nach Möglichkeit vermieden werden. Die User-Level-Threadbibliothek *m(icro)-threads* [Stecker95], auf die in der Vorstellung des ELiTE-Projekts noch näher eingegangen wird, zeigt, welches Ausmaß solche Effekte annehmen können. Ein weiteres Beispiel für ein nach Möglichkeit zu vermeidendes Ereignis ist die Fortsetzung eines Threads mit aufgebautem Cacheinhalt auf einem anderen Prozessor. Auch hier kann die *m-threads* Bibliothek als ein Beispiel für ein Lokalitätsscheduling genannt werden.

Trotz des Einsatzes von Caches, NUMA-Architekturen und Techniken zur Verbesserung der Speicherbandbreite kann es bei Multiprozessorsystemen zu Engpässen bei Speicherzugriffen kommen, insbesondere wenn große Speicherbereiche schnell referenziert werden, so daß der Einsatz von Caches nicht zum Tragen kommt. Ein typisches Beispiel hierfür sind Multimediaanwendungen. Dies kann dazu führen, daß eine zeitkritische Anwendung - trotz einzeln für sich gesehen ausreichender Rechenzeitgarantien - seitens des Systemschedulings nicht schnell genug ausgeführt werden kann, weil sie durch Speicherzugriffe anderer Prozessoren ausgebremst wird. Auch diese Problematik wird innerhalb des ELiTE-Projekts behandelt (siehe Abschnitt 5.3.2.2). Ebenso wie im Fall der Caches kann davon ausgegangen werden, daß andere, schon bei Einprozessorsystemen eingesetzte Techniken zum effizienten Umgang mit Speichern auf Multiprozessorsystemen eine größere Bedeutung haben, da hier im allgemeinen Speicherzugriffe teurer als im Einprozessorfall sind und die Speichergröße oft nicht linear mit der Prozessoranzahl wächst.

In diesem Rahmen soll die Betrachtung der Speicher- und Cache-Problematik an diesem Punkt genügen, da der Schwerpunkt der vorliegenden Arbeit im Bereich des Schedulings liegt. Durch die engen Wechselwirkungen zwischen Speicher und Caches auf der einen und Scheduling auf der anderen Seite, die in den bisherigen Ausführungen immer wieder deutlich geworden sind, werden verschiedene Aspekte in den folgenden Kapiteln weiterhin eine Rolle spielen. Einige Aspekte werden auch in Zusammenhang mit der Beschreibung des gesamten ELiTE-Projekts in Kapitel 5 noch einmal aufgegriffen.

4.2 Prozesse, Threadverwaltung und Threadscheduling

Die Gründe die zur Entwicklung des Konzepts der Threads geführt haben, sind schon in Abschnitt 2.2.2.3 ausführlich dargelegt worden. Die Vorteile, Parallelität innerhalb eines Prozesses (Task) und nicht durch mehrere Prozesse zu realisieren, liegen - kurz zusammengefaßt - in einer besseren Effizienz und einer einfacheren Nutzung gemeinsamer Speicherbereiche.

Die Aufteilung eines Prozesses in mehrere Threads führt zu einer Klasse von Problemen, die darin begründet liegt, daß die Systemsoftware nunmehr den Thread als Basiseinheit des Schedulings ansieht und Verwandschaftsverhältnisse im Sinne einer Zugehörigkeit zum selben Prozeß nicht berücksichtigt. Ein Prozeß, der beispielsweise zehn Threads startet, kann folglich im Extremfall die zehnfache Rechenkapazität eines Systems nutzen als ein Prozeß mit nur einem Thread. Es ist sehr einfach möglich, ein Rechensystem durch eine auf mehreren Threads basierende Anwendung zu monopolisieren und für andere Anwendung bzw. Benutzer unbenutzbar zu machen. Der gleiche Effekt ist zwar auch bei einer auf mehreren Prozessen basierenden Anwendung möglich, ist aber offensichtlicher - und auch dem Benutzer oft bewußter - als im Fall von Threads. Eine Beschränkung der in Anspruch genommenen Kapazität einer Anwendung mit mehreren rechenintensiven Threads durch einen problembewußten Benutzer ist ohne Systemunterstützung nur sehr schwer zu realisieren. Die Systemsoftware von Multiprozessorsystemen sollte die Möglichkeit anbieten, in die Verteilung der Rechenkapazitäten die bisher verursachte Last verwandter Threads einfließen zu lassen, also eine Rechenzeitverteilung auf Basis von Prozessen ermöglichen. Auch eine Gruppenbildung von Threads durch den Benutzer wäre eine sinnvolle Erweiterung. Uns ist kein System im UNIX-Umfeld bekannt, daß solche oder ähnliche Dienste anbietet.

Ein anderes Problem dieser Art tritt bei parallelen Anwendungen auf, die sich durch eine ausgeprägte Interaktion zwischen den einzelnen Threads auszeichnen. Solche Anwendungen laufen nur dann effizient ab, wenn ihre Threads gleichzeitig auf mehreren Prozessoren abgearbeitet werden, da ansonsten ein Thread immer wieder unterbrochen werden muß, bis seinem Kommunikationspartner ein Prozessor zugeteilt wird. Im ungünstigsten Fall verbraucht ein Thread seine gesamte Zeitscheibe mit aktivem Warten auf die Antwort eines Threads, der erst anschließend ausgeführt wird. Ein Lösungsansatz sind verschiedene Formen von Gangscheduling [FeiRud90 und Ouster82], mit deren Hilfe versucht wird, einer zusammengehörigen Gruppe von Threads möglichst gleichzeitig die entsprechende Anzahl von Prozessoren zuzuteilen. Solche Strategien können bei entsprechender Konfiguration so weit gehen, daß Anwendungen mit einem hohen Bedarf an Prozessoren zeitweise überhaupt nicht ausgeführt werden, wenn die entsprechende Prozessoranzahl nicht gleichzeitig verfügbar ist.

Eine weitere Möglichkeit besteht in einer Zuweisung von Prozessoren an Anwendungen bzw. Prozesse. Diese Prozessor- oder CPU-Scheduling genannte Technik wird beispielsweise für den MACH CPU-Server [Black89, Black90a und Black90b] eingesetzt. Die Aufgabe des CPU-Servers besteht in einer fairen Verteilung der Prozessoren zwischen den einzelnen Anwendungen, die über den Server Prozessoren für eine exklusive Nutzung anfordern können. Der CPU-Server kann Prozessoren wieder entziehen und an andere Anwendungen vergeben.

Einen einfacheren, ebenfalls auf - allerdings statischer - Zuweisung exklusiv nutzbarer Prozessoren basierenden Ansatz stellt die Partitionierung großer Multiprozessorsysteme dar. Diese weit verbreitete Methode, die zum Beispiel auf Convex/HP SPP Systemen [Convex93] und den Systemen von Kendall Square Research [KSR91] eingesetzt wird, hat Vor- und Nachteile. Die in vielen Fällen exklusiv von einer Anwendung genutzten Partitionen ermöglichen die optimale Adaption einer Anwendung an eine durchgehend verfügbare feste Anzahl von exklusiv nutzbaren Prozessoren. Eine statische Partitionierung kann, insbesondere wenn auch der verfügbare Speicher fest auf die Partitionen aufgeteilt ist, in bestimmten Anwendungsprofilen zu einem günstigen Gesamtdurchsatz des Rechensystems führen. Ein solcher Effekt ist besonders dann wahrscheinlich, wenn hauptsächlich sehr rechen- und speicherintensive und nicht interaktive Probleme bearbeitet werden. In anderen Fällen kann eine Partitionierung zu Einbrüchen im Gesamtdurchsatz oder sogar zu teilweise leerlaufenden Systemen führen.

Eine eingehende Abhandlung des Gang- bzw. CPU-Schedulings und der Partitionierung in Zusammenhang mit NUMA-Architekturen kann bei Brecht [Brecht94] nachgelesen werden. Im Rahmen der vorliegenden Arbeit ist diese Problematik nur am Rande von Interesse. Sie ist jedoch insofern wichtig, als der *Sleeping-Threads* Mechanismus und andere im weiteren vorgestellte Arbeiten zur besseren Integration von User-Level-Threads solche Mechanismen für einen effizienten Ablauf voraussetzen oder selbst teilweise realisieren.

Um eine bessere Integration von User-Level-Threads mit den übrigen Bestandteilen der Systemsoftware zu motivieren, ist eine tiefergehende Erörterung einiger Aspekte von Kernel-Threads auf der einen und User-Level-Threads auf der anderen Seite notwendig.

4.3 Kernel-Threads

Neben den konzeptionellen Vorteilen und der besseren Effizienz, die Kernel-Threads gegenüber einer parallelen Programmierung auf Basis von Prozessen auszeichnen, sind eine Reihe von Einschränkungen zu erwähnen, die letztlich dazu geführt haben, auch auf Systemen, die Threads als Betriebssystemdienst anbieten, Threadumschaltung und Schedulingstrategien außerhalb des Kerns auf Benutzerebene zu implementieren.

- Effizienz der Threadumschaltung:
  Im Gegensatz zu einer Threadumschaltung auf Benutzerebene ist jeder Threadwechsel einer Kernel-Thread-Implementierung mit den Kosten für einen Einsprung in den Systemkern verbunden. Der Threadscheduler des Kerns, der im allgemeinen komplexer als ein User-Level-Scheduler ist, muß darüberhinaus eine Reihe notwendiger Routinen oder Abfragen durchlaufen. Beispielsweise muß bei jeder Threadumschaltung überprüft werden, ob nicht auch ein Prozeßwechsel vorliegt und somit eine Adreßraumumschaltung erfolgen muß. Aufgrund der generellen Einsetzbarkeit des Betriebssystemdienstes Thread müssen in jedem Fall alle Register gesichert werden, unabhängig davon, ob eine Anwendung beispielsweise die Floating-Point-Register verwendet. Eine entsprechende Parametrisierung durch die Anwendung wäre zwar möglich, ist aber in keinem uns bekannten System realisiert.

- Integration verschiedener Schedulingstrategien:
  Die Schedulingstrategien für Kernel-Threads sind nicht variierbar. Der Thread-Scheduler des Systemkerns verwaltet Threads über Anwendungsgrenzen hinweg und muß daher eine für das gesamte System einheitliche Strategie implementieren. Diese Restriktion kann im geringen Maß aufgeweicht werden, wie zum Beispiel durch die Scheduling-Klassen des Betriebssystems Solaris der Firma Sun. Auch unter Solaris ist dies nur mit restriktiven Nebenbedingungen möglich, da ein sinnvolles Zusammenspiel zwischen den Scheduling-Klassen gewährleistet werden muß. Scheduling-Klassen sind als Bestandteil des Betriebssystemkerns nicht ohne weiteres hinzufügbar. Für anwendungsspezifische Schedulingstrategien ist dieser Ansatz nicht geeignet, er wird u.a. eingesetzt, um Threads mit (eingeschränkter) Echtzeit-Priorität ausführen zu können.

- Interaktion mit der Anwendung:
  Der Scheduler des Kerns bestimmt den nächsten auszuführenden Thread über berechnete Prioritäten, in die die innerhalb eines bestimmten Zeitintervalls bereits zugeteilte Rechenzeit als einer der Hauptfaktoren negativ eingeht. Dadurch kann eine faire Verteilung der Rechenzeit gewährleistet werden. Für ein Scheduling über Anwendungsgrenzen hinweg ist dies ein sinnvoller Ansatz. Für das Scheduling innerhalb einer Anwendung ist Fairness nicht in jedem Fall notwendig und sinnvoll, eventuell sogar kontraproduktiv. Um solche Faktoren in das Scheduling einfließen zu lassen, sind Informationen über die Anwendung notwendig, die einem anwendungsunabhängigen Scheduler nicht verfügbar sind, sondern nur durch die Anwendung selbst - zum Beispiel über eine Steuerung der Schedulingentscheidungen - eingebracht werden können.

- Allgemeine Optimierungen:
  Auch allgemeine Optimierungen, die unabhängig von einem konkreten Wissen über eine Anwendung möglich sind, können nicht ohne weiteres in eine Schedulingstrategie für Kernel-Threads integriert werden. Es muß sichergestellt sein, daß dies nicht zur Bevorzugung einer Anwendung führen kann. Eine Berücksichtigung zum Beispiel des Cache-Zustands, wie im Rahmen des ELiTE-Projekts durch die *m-threads* realisiert, wäre nicht in einer vergleichbaren Form möglich, da Anwendungen mit hoher Speicherlokalität und folglich einer geringen Zahl von Cachemisses bevorzugt würden. Wie ein Cache-Affinity-Scheduling für Kernel-Threads aussehen könnte, legt Tucker näher dar [Tucker93].

Für einige der aufgezählten Einschränkungen könnten durchaus Lösungen innerhalb eines Schedulings im Betriebssystemkern realisiert werden, wenn eine Verteilung der Prozessorkapazitäten auf Basis von Prozessen und eine Gewichtung der Threads eines Prozesses ohne Beeinträchtigung prozeßfremder Threads integriert würde. Im Hinblick auf die in Kapitel 2 und 3 beschriebene langsame Fortentwicklung von Betriebssystemen war und ist jedoch eine Realisierung auf der Ebene von User-Level-Threads der gangbarere Weg, insbesondere wenn man bedenkt, daß jede einzelne neue Strategie und jede Schnittstelle zwischen Scheduler und Anwendung wiederum in den Betriebssystemkern integriert werden müßten. Einer Verbesserung der Effizienz der Threadumschaltung sind bei Kernel-Thread-Implementierungen im Vergleich zu einer Umschaltung auf Benutzerebene immer engere Grenzen gesetzt.

4.4 User-Level-Threads

4.4.1 Vorteile gegenüber Kernel-Threads

Die Vorteile des User-Level-Schedulings lassen sich von den im vorausgehenden Abschnitt erläuterten Einschränkungen bezüglich der Kernel-Threads ableiten.

Aufgrund des Wegfalls der Kosten für einen Einsprung in den Betriebssystemkern und der Behandlung unterschiedlicher Prozesse sowie einer Sicherung lediglich der für eine bestimmte Anwendung notwendigen Register ist eine erheblich effizientere Threadumschaltung realisierbar. Darüber hinaus wird ein effizienterer Ablauf dadurch erreicht, daß auch potentiell blockierende Synchronisationsmechanismen, wie zum Beispiel Semaphore, im User-Level implementiert werden. Weiterhin können die Datenstrukturen einer User-Level-Threadverwaltung besser in Richtung eines effizienten Ablaufs optimiert werden, als dies im Betriebssystemkern im allgemeinen realisiert ist.

Mit vergleichsweise geringem Aufwand können beliebige, an eine Anwendung optimal angepaßte Schedulingstrategien implementiert werden. Dies gilt ebenso für Formen der Interaktion zwischen Anwendung und Scheduler, mit denen durch Ausnutzung anwendungsinterner Informationen das Scheduling weiter optimiert werden kann. Solche anwendungsspezifische Information oder andere Faktoren (zum Beispiel der Cache-Zustand), die eine faire Verteilung der Prozessorkapazitäten potentiell beeinträchtigen, können - soweit eine Anwendung selbst dies erlaubt - im vollen Umfang und folglich mit maximaler Wirkung in die Scheduling-Entscheidungen eingehen. Eine faire Verteilung zwischen verschiedenen Anwendungen wird durch den vom User-Level-Scheduling völlig unabhängigen Scheduler des Betriebssystemkerns sichergestellt.

Einer der wichtigsten Vorteile eines effizienten User-Level-Schedulings ist, daß der geringe Overhead einen massiven Einsatz von Threads erlaubt. Anwendungen können ihrer logischen Struktur nach auf Threads abgebildet werden und müssen nicht aus Gründen der Effizienz in eine der Rechnerarchitektur angepaßte Struktur gepreßt werden. Die Anzahl der genutzten Threads kann beispielsweise an der Struktur einer rechenintensiven Anwendung anstatt an der Anzahl der Prozessoren ausgerichtet werden. Dadurch kann ebenfalls eine bessere Portierbarkeit auf andere Rechensysteme erreicht werden.

4.4.2 Probleme des User-Level-Schedulings

Die Unabhängigkeit zwischen den Schedulinginstanzen auf Benutzerebene und im Systemkern ist aber auch eine der entscheidenden Ursachen für Probleme, zu denen ein User-Level-Scheduling führt. Anders formuliert bedeutet dies, daß die Probleme des User-Level-Schedulings, von denen im folgenden die wichtigsten beschrieben werden, zum größten Teil in einer ungenügenden Einbindung der Threadverwaltung auf Benutzerebene in die restlichen Teile der Systemsoftware begründet sind.

4.4.2.1 Scheduling auf zwei Ebenen (Two-Level-Scheduling)

Die Schedulingstrategien eines reinen User-Level-Schedulings können den Programmablauf nur optimieren, indem sie festlegen, welcher User-Level-Thread zu welchem Zeitpunkt auf welchem virtuellen Prozessor (bzw. Kernel-Thread) ausgeführt wird. Der User-Level-Scheduler hat jedoch keinerlei Einfluß darauf, ob ein virtueller Prozessor zu einem bestimmten Zeitpunkt überhaupt vom Betriebssystem ausgeführt wird und an welchen physikalischen Prozessor er gebunden ist. Im allgemeinen kann keine Aussage darüber getroffen werden, ob zwei User-Level-Threads, die aus Sicht des User-Level-Schedulers gleichzeitig ablaufen sollen, dies tatsächlich tun oder ob ein User-Level-Thread, der aufgrund einer guten Cache-Ausnutzung immer auf dem gleichen Prozessor ausgeführt werden soll, nicht doch den Prozessor wechselt. Ein unkoordiniertes Zusammenspiel zwischen User-Level- und Kernel-Scheduler kann den potentiellen Gewinn eines optimierten User-Level-Schedulings verhindern oder sogar ins Negative umkehren.

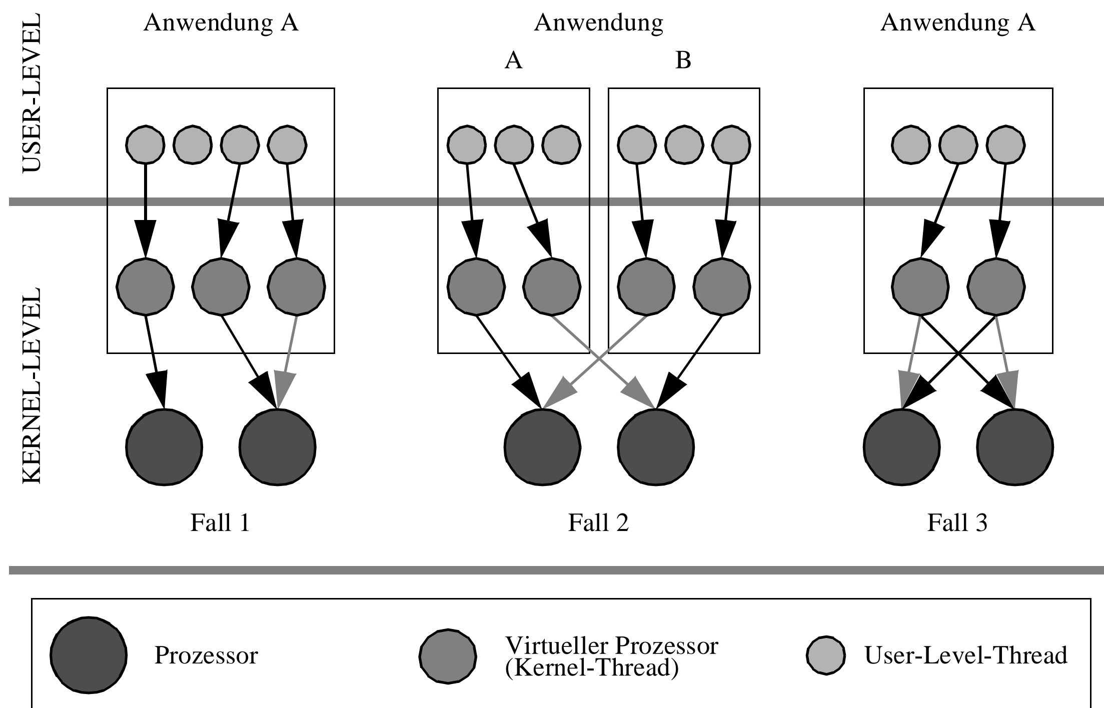

Abb. 4.1: Two-Level-Scheduling

In Abbildung 4.1 sind verschiedene Fälle aufgezeigt, in denen sich das unkoordinierte Scheduling auf Benutzer- und Systemebene (Two-Level-Scheduling) störend auf das User-Level-Scheduling auswirken kann. In Fall 1 nutzt der User-Level-Scheduler mehr virtuelle Prozessoren als der Anwendung physikalische Prozessoren zur Verfügung stehen. Der Kernel-Scheduler muß zwischen den - als virtuelle Prozessoren genutzten - Kernel-Threads umschalten. Zusätzlich zu den bereits genannten Problemen führt dies zu einem erhöhten Overhead durch das Kernel-Scheduling. Ein User-Level-Scheduler sollte folglich nur soviele virtuelle Prozessoren verwenden, wie physikalische verfügbar sind.

Die gleichen Probleme treten auf, wenn nicht in einer Anwendung, sondern erst durch die gleichzeitige Ausführung mehrerer Anwendungen mehr Kernel-Threads als Prozessoren ge-

nutzt werden (Fall 2). Da ein nicht in das Gesamtsystem integrierter User-Level-Scheduler sich nicht an die Anzahl der tatsächlich nutzbaren Prozessoren anpassen kann, bleibt als einzige Lösung die statische Partitionierung des Systems, wenn der potentielle Gewinn durch ein optimiertes User-Level-Scheduling ausgenutzt werden soll. Ein Gang-Scheduling auf Ebene der Kernel-Threads könnte zwar sicherstellen, daß eine Anwendung nur dann ausgeführt wird, wenn die benötigte Prozessoranzahl gleichzeitig verfügbar ist, löst jedoch nur das Problem einer gleichzeitigen Ausführung bestimmter User-Level-Threads. Ein auf gute Cache-Ausnutzung optimiertes User-Level-Scheduling zum Beispiel kann auf dieser Basis nicht effizient ablaufen, da ansonsten auch garantiert werden müßte, daß dieselben Prozessoren verwendet werden. Weiterhin kann ein Gang-Scheduling, das nicht nur versucht, möglichst viele Threads einer Anwendung gleichzeitig auszuführen, sondern dies für bestimmte Threads sogar garantiert, durch Leerlauf einzelner Prozessoren zu einer geringen Auslastung des Gesamtsystems führen.

Auch eine statische Partitionierung löst nicht in jedem Fall alle Probleme. Wenn die Schedulingstrategie des Systemkerns nicht in Richtung einer Prozessoraffinität optimiert ist, können kurzzeitige Unterbrechungen, etwa aufgrund eines Pagefaults, sehr schnell die Zuordnung zwischen Kernel-Threads und Prozessoren ändern (Fall 3). Andere Quellen eines Schedulings, das potentiell die Zuordnung ändern kann, sind sehr kurz laufende Systemthreads wie zum Beispiel Interrupt-Threads auf Solaris-Systemen[1] oder Server-Threads unter MACH.

4.4.2.2 Blockierungen im Betriebssystemkern

Blockiert ein Kernel-Thread im Systemkern, wird der freiwerdende Prozessor genutzt, um einen anderen ablauffähigen Kernel-Thread auszuführen. Ein User-Level-Scheduler verhält sich analog, wenn ein Thread innerhalb eines Synchronisationsaufrufs der Threadbibliothek blockiert wird. Der virtuelle Prozessor kann einen anderen User-Level-Thread zur Ausführung bringen. Führt ein User-Level-Thread einen blockierenden Systemaufruf durch oder blockiert aufgrund eines Pagefaults, verhält sich das System anders. Nicht der User-Level-Thread, der im Systemkern überhaupt nicht bekannt ist, wird blockiert, sondern der Kernel-Thread, der als virtueller Prozessor im Kontext des User-Level-Threads läuft. Ein User-Level-Scheduler kann aufgrund seiner ungenügenden Systemintegration diese Situation nicht einmal erkennen und die Anwendung wird für die Dauer der Blockierung auf einer verminderten Zahl von virtuellen Prozessoren fortgesetzt. Abbildung 4.2[2] verdeutlicht dieses Problem.

Vereinfachend wird angenommen, daß User-Level-Threads immer ablauffähig sind, da eine Blockierung innerhalb des User-Level-Schedulers kein Problem darstellt und damit in diesem Zusammenhang uninteressant ist. Alle User-Level-Threads befinden sich folglich im Zustand **R** (*runnable*). Die Bindung eines User-Level-Threads, der aktuell ausgeführt wird, an einen virtuellen Prozessor wird durch einen Pfeil symbolisiert. Es wird wiederum vereinfachend angenommen, daß virtuelle Prozessoren (Kernel-Threads) im Normalfall immer entweder im Benutzer-

1. Interrupts werden unter Solaris nicht in jedem Fall im Kontext des unterbrochenen Threads behandelt, sondern können auch in Threads umgesetzt werden, die unabhängig von dem unterbrochenen Thread die Interrupt-Behandlung übernehmen.
2. Die gleiche Darstellungsform wird später auch zur Erläuterung des *Sleeping-Threads* Mechanismus verwendet. Dies erklärt die für den hier vorliegenden einfachen Fall übertrieben komplexe Darstellung.

oder im Kernelmodus ablaufen, daß ihnen also immer ein Prozessor zur Verfügung steht. Dies entspricht einer - wie auch immer realisierten - idealen Umsetzung der Forderung aus Abschnitt 4.4.2.1, daß die Anzahl der virtuellen Prozessoren der Anzahl der physikalischen entsprechen soll. Ein virtueller Prozessor bzw. Kernel-Thread kann also nur aufgrund eines Verhaltens des ausgeführten User-Level-Threads (Systemaufruf oder Pagefault) in einen anderen Zustand wechseln. Die Bindung an den entsprechenden Zustand wird durch einen weiteren Pfeil dargestellt. Da der Zustand "ablaufend im Kern" nicht von Interesse ist, wird nur eine Bindung an den Zustand **B** (blockiert) dargestellt. User-Level-Threads werden durch kleine, virtuelle Prozessoren durch große Kreise und Kernzustände durch Quadrate repräsentiert.

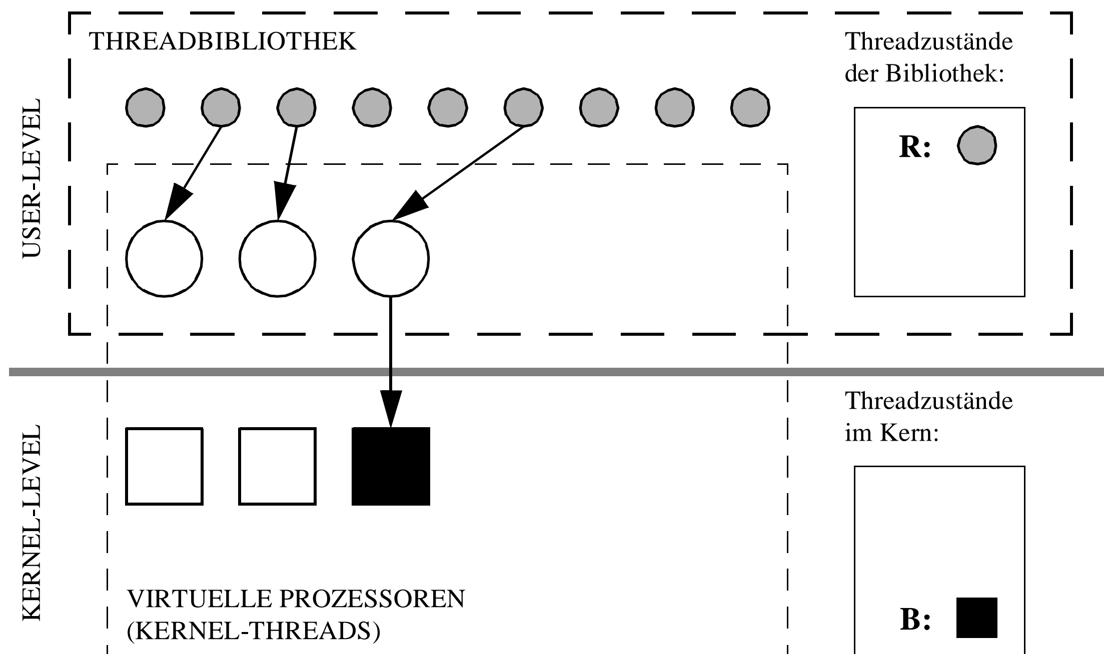

Abb. 4.2: Parallelitätseinbruch durch Blockierungen im Kern

In dem dargestellten Beispiel läuft eine Anwendung mit neun User-Level-Threads auf Basis eines Schedulers, dem drei Prozessoren zugewiesen sind. Um einen optimalen Ablauf der Anwendung zu gewährleisten, nutzt der Scheduler drei Kernel-Threads als virtuelle Prozessoren. Führt, wie in diesem Fall, der sechste User-Level-Thread einen blockierenden Systemaufruf aus oder läuft auf einen Pagefault, wird der entsprechende virtuelle Prozessor im Kern blockiert und die Anwendung kann nur noch zwei Prozessoren nutzen. Die Parallelität der Anwendung bricht folglich ein, obwohl ein Prozessor verfügbar ist und lauffähige User-Level-Threads zur Ausführung anstehen. Ist die "1 zu 1"-Abbildung zwischen virtuellen und physikalischen Prozessoren durch eine statische Partitionierung[3] realisiert, bricht gleichzeitig die Gesamtauslastung des Rechensystems ein, da der frei werdende Prozessor auch nicht für andere Anwendungen eingesetzt werden kann, sondern leerläuft. Gleiches gilt, wenn ein System exklusiv für eine Anwendung genutzt wird.

3. Eine andere Lösung ist, wie in Abschnitt 4.4.2.1 begründet, bei einem reinen User-Level-Scheduling nicht sinnvoll realisierbar.

Die einzige Weg, einen solchen Parallelitätseinbruch zu vermeiden, ist eine Erhöhung der Anzahl der virtuellen Prozessoren, bis sichergestellt ist, daß immer eine genügend große Anzahl lauffähig ist, um alle Prozessoren auslasten zu können. Dies führt jedoch zu einem Two-Level-Scheduling und damit zu einem Effizienzeinbruch auf einer anderen Ebene. Im Extremfall kann dies zu einer Pervertierung des User-Level-Schedulings führen, indem jeder User-Level-Thread an seinen eigenen virtuellen Prozessor gebunden ist. Es ist folglich nicht möglich, auf Basis eines nicht in das restliche System eingebundenen User-Level-Schedulings die Forderungen nach einem effizienten Ablauf auf der einen und nach einer optimalen Auslastung der verfügbaren Rechenleistung auf der andern Seite gleichzeitig zu erfüllen.

4.4.2.3 Systembedingte Verklemmungen

Blockierungen im Systemkern können in Zusammenhang mit User-Level-Threadbibliotheken jedoch nicht nur zu dramatischen Performanzeinbrüchen, sondern auch zu einem Stillstand der gesamten Anwendung führen, wenn alle virtuellen Prozessoren dauerhaft blockiert sind und ein User-Level-Thread, der diese Blockierungen aufheben könnte, deshalb nicht ausgeführt werden kann. Potentiell dauerhafte Blockierungen können nur aufgrund bestimmter Systemaufrufe auftreten. Dazu gehören u.a. Synchronisationsdienste des Systemkerns, wie zum Beispiel Semaphor- oder Mutexoperationen, die in Zusammenhang mit User-Level-Threadbibliotheken überhaupt nicht verwendet werden sollten[4], File-Locking oder bestimmte Formen von E/A-Operationen. Mit diesen Diensten muß daher bei einer Programmierung auf Basis von User-Level-Threadbibliotheken vorsichtig umgegangen werden.

Diese Art von Verklemmung ist nicht auf Programmierfehler in der Anwendung zurückzuführen, wie dies zum Beispiel bei Verklemmungen aufgrund einer falschen Reihenfolge in der Belegung von Locks der Fall wäre. Das eigentliche Problem liegt in der Implementierung des User-Level-Schedulings, die nicht sicherstellen kann, daß ein ablauffähiger Thread tatsächlich ausgeführt wird. Eine Anwendung, die unter einem Kernel-Scheduling korrekt ausgeführt wird, kann unter bestimmten Umständen verklemmen, wenn ein User-Level-Scheduler eingesetzt wird.

Hier sind folglich auch Fragen der Ergonomie des Programmiermodells der User-Level-Threads angesprochen. Das Verhalten einer User-Level-Threadbibliothek entspricht nicht den Erwartungen, die ein Anwender im allgemeinen an das Programmiermodell Threads stellt. Vorausgesetzt, daß dies einem Anwender überhaupt bewußt ist, führt die aufwendige Behandlung von Programmstellen, die zu solchen Problemen führen können, zu einem komplexeren Programmtext und stellt damit eine neue potentielle Fehlerquelle dar. Das nicht der allgemeinen Vorstellung entsprechende Programmiermodell führt nicht erst im Fall einer Verklemmung zu Problemen. Eine nicht den speziellen Eigenschaften des User-Level-Schedulings angepaßte Anwendungsprogrammierung kann aufgrund lang andauernder vermeidbarer Blockierungen zum Leerlauf von Prozessoren führen. Der normale Anwender ist sich solcher Probleme oft nicht bewußt.

4. Stattdessen bieten User-Level-Threadbibliotheken eigene Synchronisationsdienste an, die Threads auf Ebene des User-Level-Schedulers blockieren.

Die Gefahr von Verklemmungen in User-Level-Thread Implementierungen kann über den Begriff der Verklemmungsfreiheit formal gefaßt werden. Nach Hofmann liegt eine verklemmungsfreie Implementierung genau dann vor, wenn eine Verklemmung der Implementierung schon in der Spezifikation erkannt werden kann [Hofman91]. Faßt man die Formulierung einer parallelen Anwendung auf Basis von Threads als Spezifikation und die Realisierung auf Basis eines User-Level-Schedulings als Implementierung auf, bedeutet dies, daß ein User-Level-Scheduler keine verklemmungsfreie Implementierung des Programmiermodells Threads darstellt.

Das Problem der potentiellen Verklemmungsgefahr wird in kaum einer Implementierung berücksichtigt. Eine positive Ausnahme stellt das User-Level-Scheduling des Betriebssystems Solaris dar. Im Zusammenspiel zwischen User-Level-Bibliothek und Kern wird im Fall einer Verklemmung, daß heißt, wenn alle virtuellen Prozessoren einer Anwendung blockiert sind, über ein spezielles Signal einer der blockierten Kernel-Threads unterbrochen, damit er innerhalb der Signalbehandlungsroutine einen weiteren virtuellen Prozessor erzeugen kann. Der neue virtuelle Prozessor kann, falls vorhanden, lauffähige User-Level-Threads ausführen. Kommt es erneut zu einer Blockierung, wiederholt sich das Verfahren [LewDan96]. Eine Verklemmung aufgrund der Einschränkungen des User-Level-Schedulings kann damit aufgelöst werden. Das Verfahren setzt allerdings erst nach dem Erkennen einer Verklemmung und damit erst nach einem bereits andauernden Parallelitätseinbruch ein.

4.4.2.4 Signale, unterbrechendes Scheduling und Zeitscheibenverfahren

Der Signalmechanismus des Betriebssystems UNIX setzt auf dem Prozeßkonzept auf, was zu grundsätzlichen Problemen führt, wenn mehrere Kernel-Threads in einem Prozeß koexistieren. Das Problem liegt in der Bestimmung des Kernel-Threads, dem das Signal zugestellt und in dessen Kontext die Signalbehandlung durchgeführt wird. Im allgemeinen werden zwei Klassen von Signalen gebildet, für die unterschiedliche Lösungen realisiert sind. Bei synchronen Signalen, die synchron zum Ablauf eines Threads ausgelöst werden (zum Beispiel aufgrund einer Speicherschutzverletzung), ist sehr einfach zu realisieren, daß sie dem auslösenden Thread zugestellt werden, da dieser gerade ausgeführt wird. Asynchrone Signale hingegen, die durch abgeschlossene E/A-Operationen, Timer oder Benutzerinteraktion ausgelöst werden, sind mit dem Prozeß, nicht mit einzelnen Threads, assoziiert. Einige Systeme erlauben thread-spezifische Signalmasken, die das Sperren von Signalen für bestimmte Threads und dadurch eine eingeschränkte Steuerung erlauben. In anderen Systemen ist die Auswahl durch die Implementation des Kerns festgelegt.

Am Beispiel des ALARM-Signals läßt sich zeigen, zu welchen Problemen ein Zusammenspiel der Thread- und Signalmechanismen führen kann. Das ALARM-Signal wird häufig verwendet, wenn potentiell unendlich blockierende Systemaufrufe nach einiger Zeit abgebrochen werden. Bevor ein Thread einen solchen Aufruf ausführt, kann er einen Alarm-Timer setzen, um nach dessen Ablauf durch das ALARM-Signal unterbrochen zu werden. Da das ALARM-Signal ein asynchrones Signal ist, kann nicht sichergestellt werden, daß der richtige Thread unterbrochen wird. Über thread-spezifische Signalmasken läßt sich dieses Problem lösen, solange nur ein Thread Timer verwendet. Mehrere Threads müßten in jedem Fall kooperieren, da die Alarmti-

mer im allgemeinen nicht thread-spezifisch sind und nur für einen Prozeß gesetzt werden können.

Aus Sicht einer User-Level-Thread-Implementierung ergibt sich ein weiterer Aspekt. Da User-Level-Threads dem Systemkern nicht bekannt sind, kann auch der traditionelle Signalmechanismus nicht für User-Level-Threads erweitert oder angepaßt werden. Ein Signalmechanismus kann nur innerhalb der Threadbibliothek selbst realisiert werden und wird im allgemeinen nur programmgesteuert innerhalb des gleichen Prozesses aktivierbar sein. Dies ist einfach implementierbar, wenn es ausreicht, daß Signale grundsätzlich erst beim nächsten Aufruf einer Schedulerroutine behandelt werden. Im anderen Fall muß als Voraussetzung die Möglichkeit bestehen, einen laufenden Kernel-Thread unterbrechen zu können.

Diese Möglichkeit, einen Kernel-Thread und damit einen virtuellen Prozessor programmgesteuert unterbrechen zu können, ist auch Voraussetzung für unterbrechende Schedulingstrategien auf Benutzerebene. Eine Unterbrechung wäre durch ein Signal realisierbar, das direkt einem Kernel-Thread zustellbar ist und ansonsten innerhalb der Anwendung nicht benötigt wird. Innerhalb der Signalbehandlung können die entsprechenden Schedulingroutinen aufgerufen und damit gegebenenfalls ein Threadwechsel eingeleitet werden. Zur Implementation eines Zeitscheibenverfahrens wäre darüberhinaus ein thread-spezifischer Timer notwendig, der nach Ablauf nicht dem Prozeß, sondern dem Thread, der den Timer aufgezogen hat, ein Signal zustellt[5]. Es muß ebenfalls berücksichtigt werden, ob der Einsatz des Signalmechanismus, insbesondere bei komplexen Implementierungen auf NUMA-Prozessoren, nicht zu zeitaufwendig ist, um ein effizientes Zeitscheibenverfahren zu realisieren.

4.5 Lösungsansätze

Die Probleme, die bei einer Implementierung von Threads außerhalb des Betriebssystemkerns auftreten, liegen in der unzureichenden Integration mit dem Systemkern begründet. Eine bessere Integration ist jedoch auf Basis der vom Betriebssystemkern angebotenen Dienste nicht möglich. Anderson et. al. drücken dies mit der These “kernel threads are the *wrong abstraction* on which to support user-level management of parallelism” treffend aus [ABLL92]. Im folgenden wird eine Auswahl der Arbeiten vorgestellt, die sich mit dem Problem der Integration des User-Level-Schedulings befassen. Anschließend werden im Vergleich dazu die Thread-Scheduling-Konzepte erweiterbarer Betriebssysteme und als weitere Alternative das Thread-Scheduling auf Hardware-Ebene betrachtet.

5. Theoretisch ist eine Implementierung auch mit einem prozeß-spezifischen ALARM-Timer möglich, wenn der unterbrochene Thread vor einem Einsprung in die Schedulingroutinen allen anderen virtuellen Prozessoren explizit ein Signal zustellt. Dies stellt aber bezüglich der Effizienz keine befriedigende Lösung dar.

4.5.1 Integration von User-Level-Threads

4.5.1.1 *Scheduler Activations*

Scheduler Activations [ABLL92] bilden eine Alternative zu Kernel-Threads als Basis für die Implementierung von User-Level-Threads. Ursprünglich in das Betriebssystem Topaz des DEC SRC Firefly Multiprozessors integriert, wurden sie in einer weiteren Arbeit auf MACH 3.0 portiert [BMVL93].

Scheduler Activations stellen eine spezielle Form von Kernel-Threads dar, die mit einem User-Level-Scheduler interagieren. Ereignisse wie Blockierungen, Deblockierungen oder eine Änderung der Anzahl der zugeteilten Prozessoren werden an den User-Level-Scheduler über Upcalls weitergereicht. Im Fall einer Blockierung beispielsweise erzeugt der Systemkern eine neue *Scheduler Activation*, die im User-Level den Upcall-Handler des User-Level-Schedulers ausführt. Der User-Level-Scheduler kann nun auf die Blockierung reagieren und die neue *Scheduler Activation* nutzen, um einen alternativen User-Level-Thread auszuführen. Im Fall einer Deblockierung setzt der Kern eine *Scheduler Activation* nicht von sich aus fort, sondern informiert wiederum über einen Upcall den User-Level-Scheduler. Zusammen mit einem CPU-Scheduling, über das die verfügbaren Prozessoren den verschiedenen User-Level-Schedulern zugewiesen werden, löst dieser Mechanismus neben der Vermeidung von Parallelitätseinbrüchen das Problem des Two-Level-Schedulings.

Der Upcall als einzige Interaktionsmöglichkeit des Kerns mit dem User-Level-Scheduler führt jedoch zu einem neuen Problem. Im Fall einer Blockierung steht ein freier Prozessor zur Ausführung des Upcalls zur Verfügung. Im Fall einer Deblockierung oder dem Entzug eines Prozessors muß dagegen eine andere *Scheduler Activation* der gleichen Anwendung unterbrochen werden, um den Upcall durchzuführen. Hier ist eine neue Art eines Schedulings auf Kern-Ebene entstanden, die das User-Level-Scheduling stört. Ein konzeptionelles Problem zeigt sich bezüglich der MACH-Version der *Scheduler Activations*. Durch das Serverkonzept ist die Ausführung eines Systemaufrufs von der aufrufenden Anwendung abgekoppelt. Der aufrufende Thread blockiert in jedem Fall und wird durch eine neue *Scheduler Activation* ersetzt. Der durch den Aufruf aktivierte Serverthread benötigt jedoch ebenfalls einen Prozessor. Dieses Problem kann über ein CPU-Scheduling eingedämmt werden, führt jedoch zu vielen zusätzlichen Verschiebungen in der Prozessorzuteilung oder zur Notwendigkeit einer festen, oft leerlaufenden Partition für Serverthreads.

4.5.1.2 XERO Threads

Die Kernel Threads des experimentellen Betriebssystems XERO [IKNM91] sind bezüglich ihres Verhaltens bei Blockierungen mit den *Scheduler Activations* vergleichbar. Ein neu erzeugter Kernel-Thread setzt an einer definierten Stelle im User-Level-Scheduler auf und kann als Ersatz für den blockierten virtuellen Prozessor genutzt werden. Eine Deblockierung wird im Gegensatz zu den *Scheduler Activations* nicht an den User-Level-Scheduler weitergereicht. Der Kern setzt einen deblockierten Thread von sich aus fort, wodurch die Anzahl der virtuellen Prozessoren nach einer Blockierung bis auf ein einstellbares Maximum erhöht wird. Der User-Level-

Scheduler muß eine zu hohe Anzahl von virtuellen Prozessoren selbst erkennen und kann gegebenenfalls überflüssige Kernel-Threads terminieren. Der Mechanismus begrenzt folglich nur einen potentiellen Parallelitätseinbruch, ist aber nicht geeignet, um ein Two-Level-Scheduling zu vermeiden.

Die XERO Kernel-Threads unterstützen ein Zeitscheibenverfahren im User-Level-Scheduling. Die Länge der Zeitscheiben ist jedoch fest an die der Kernel-Threads gebunden und damit nicht flexibel einstellbar.

4.5.1.3 Unstable Threads

Innerhalb derselben Arbeitsgruppe entstanden ebenfalls auf Basis des Betriebssystems XERO die *Unstable Threads* [InKaMa93]. Ein herkömmlicher Kernel-Thread wird von einer Anwendung über einen Systemaufruf erzeugt, entsprechend der Strategien des Kerns ausgeführt und durch einen weiteren Systemaufruf terminiert. Während seiner Lebensdauer steht er der Anwendung beispielsweise als virtueller Prozessor für ein User-Level-Scheduling, abgesehen von nicht erkennbaren Unterbrechungen aufgrund des Kernel-Schedulings, durchgehend zur Verfügung. Ein *Unstable Thread* wird vom Kern generiert, wenn der von einer Anwendung eingestellte gewünschte Parallelitätsgrad nicht erreicht ist. Der Kern kontrolliert, wann ein *Unstable Thread* für die Anwendung verfügbar ist, und kann die Anzahl der aktiven *Unstable Threads* einer Anwendung an die Anzahl der verfügbaren Prozessoren anpassen. Ein *Unstable Thread* wird durch den Kern terminiert, wenn die gewünschte Parallelität einer Anwendung herabgesetzt wird. Die Instabilität ist in dem Sinn zu verstehen, daß ein Thread für längere Zeit durch den Kern suspendiert werden kann, weil für eine Anwendung nicht genügend Prozessoren verfügbar sind. Die Kommunikation zwischen Kern und User-Level-Scheduler erfolgt über einen gemeinsam genutzten Speicherbereich. Für jeden *Unstable Thread* wird eine Datenstruktur angelegt, die den aktuellen Zustand des Threads widerspiegelt und zur Speicherung des Registerzustands des Threads im Fall einer Suspendierung durch den Kern dient. Auf Basis dieser Information kann der User-Level-Scheduler Threads, deren virtuelle Prozessoren suspendiert sind, auf anderen fortsetzen. Durch das Konzept der *Unstable Threads* werden überflüssige Kontextwechsel aufgrund eines Two-Level-Schedulings weitgehend vermieden. Blockierungen werden in diesem Ansatz nicht behandelt.

4.5.1.4 First-Class User-Level-Threads

Die Kernel-Thread-Schnittstelle von Psyche [MSLM91], einem Betriebssystem für den BBN Butterfly Plus NUMA-Multiprozessor, ist explizit auf die Nutzung durch eine User-Level-Threadbibliothek und nicht für eine direkte Programmierung paralleler Anwendungen ausgelegt. Dadurch soll ein *First-Class* Status, eine Systemeinbindung - vergleichbar mit der von Kernel-Threads - für User-Level-Threads erreicht werden. Bezüglich der Propagierung von Blokkierungen und Deblockierungen in den User-Level durch Softwareinterrupts (bzw. Upcalls) sind *First-Class User-Level-Threads* und *Scheduler Activations* vergleichbar. Den Besonderheiten der NUMA-Architektur des BBN Butterfly Plus wird dadurch Rechnung getragen, daß ein virtueller Prozessor immer auf dem gleichen physikalischen Prozessor läuft und Softwareinterrupts immer dem betroffenen virtuellen Prozessor zugestellt werden. Es wird folglich nicht, wie

im Fall der *Scheduler Activations,* ein weiterer virtueller Prozessor unterbrochen, um eine Deblockierung zu propagieren. Weitere Arten von Softwareinterrupts erlauben unterbrechende Strategien bzw. Zeitscheibenverfahren im User-Level.

Psyche propagiert nicht alle Blockierungen in den User-Level. Insbesondere auf Pagefaults kann im User-Level-Scheduler nicht reagiert werden. Der hauptsächlich verwendete blockierende Systemaufruf ist ein RPC[6]-ähnlicher Mechanismus. Der aufrufende Thread wird blokkiert und die Ausführung des Aufrufs an einen Serverthread delegiert. Der User-Level-Scheduler kann daher im Fall eines Softwareinterrupts den Kontext des blockierten Threads sichern und auf Basis des gleichen virtuellen Prozessors (Kernel-Threads) einen weiteren User-Level-Thread ausführen und den wartenden Thread später erneut aufsetzen. Wird der Serverthread auf einem anderen Prozessor ausgeführt, wird sofort ein Softwareinterrupt erzeugt. Im lokalen Fall wird die Blockierung erst später nach bestimmten Regeln propagiert, um den virtuellen Prozessor des Servers anstelle der aufrufenden Anwendung laufen zu lassen. Durch diese Unterscheidung wird das Problem der *Scheduler Activations* bezüglich MACH vermieden (vergleiche Abschnitt 4.5.1.1).

Im Gegensatz zu *Scheduler Activations* oder *Unstable Threads* ist es nicht möglich, einen User-Level-Thread auf einem anderen virtuellen Prozessor fortzusetzen, wenn der assoziierte virtuelle Prozessor vom Kern aufgrund des CPU-Schedulings zwischen mehreren Anwendungen unterbrochen wird. Der betroffene Thread wird erst fortgesetzt, wenn dem virtuellen Prozessor erneut eine CPU zugewiesen wird. Um dies im Fall von wichtigen Threads zu verhindern, muß ein User-Level-Scheduler vermeiden, daß Threads, beispielsweise eines bestimmten Prioritätsbereiches, auf einem virtuellen Prozessor ausgeführt werden, der suspendiert wird. Dies wird über einen Softwareinterrupt ermöglicht, der zu einer einstellbaren Zeit vor einer Suspendierung zugestellt wird[7]. Der Nachteil dieser Lösung liegt darin, daß der User-Level-Scheduler im Vorfeld aktiv werden muß, um eine spätere Schedulingentscheidung zu ermöglichen. Es muß also im voraus die Bedeutung eines Threads bestimmt werden. Darüber hinaus muß sichergestellt sein, daß der eingestellte Zeitrahmen in jedem Fall ausreicht.

4.5.1.5 Hierarchische Scheduler

Ein anderes Konzept wird mit der Realisierung hierarchicher Scheduler [RieKle96] für das Betriebssystem Spring [MGHK+94] verfolgt. Die eigentliche Threadumschaltung verbleibt im Betriebssystemkern, lediglich die Schedulingstrategien werden in den User-Level verlagert. Der Vorteil eines Schedulings von Kernel-Threads durch im User-Level implementierte Strategien liegt in der Vermeidung vieler Probleme einer User-Level-Abstraktion von Threads, der Nachteil im Overhead des Systemeinsprungs für die Threadumschaltung und in einer generell einsetzbaren und damit komplexeren Implementierung der Umschaltung. Weiterhin ist eine Optimierung der Threadumschaltung selbst nicht möglich. Durch mehrere Hierarchieebenen ist ein sehr flexibles anwendungsspezifisches Scheduling möglich. Innerhalb einer Anwendung sind unterschiedliche Strategien gleichzeitig realisierbar. Dies ermöglicht zum Beispiel eine Unter-

6. Remote Procedure Call
7. Der *Two-Minute-Warning* Interrupt kann u.a. auch genutzt werden, um Probleme im Zusammenhang mit Locking zu verringern.

scheidung zwischen rechenintensiven Threads und Threads eines graphischen Ausgabesystems. Der Systemscheduler, die Wurzel der Hierarchie, verbleibt aus Sicherheitsgründen im Systemkern und garantiert eine faire Rechenzeitverteilung zwischen den verschiedenen Schedulern und direkt verwalteten Threads.

4.5.1.6 Zusammenfassung

Betrachtet man die Gemeinsamkeiten der vorgestellten Arbeiten, so wird deutlich, daß ein Thread-Scheduling im User-Level Betriebssystemmechanismen benötigt, die eine Interaktion zwischen den Schedulinginstanzen in und außerhalb des Betriebssystemkerns ermöglichen.

Die fünf angeführten Beispiele zeigen, daß über einen solchen Ansatz die Probleme des User-Level-Schedulings durchaus lösbar sind bzw. anwendungsspezifische Strategien für ein Kernel-Thread-Scheduling realisiert werden können. Es sind jedoch Modifikationen und Erweiterungen in Kernbereichen der verwendeten Betriebssysteme notwendig, so daß die entwickelten Mechanismen keine Verbreitung finden konnten. Solche Konzepte müßten folglich von den Herstellern in deren Betriebssysteme integriert werden. Obwohl die vorgestellten Arbeiten teilweise sieben Jahre zurückliegen, ist keine der Ideen in kommerzielle UNIX-Derivate eingeflossen. Die Betriebssysteme der großen Hard- oder Softwarehersteller setzen weiterhin auf das problembeladene Konzept der Kernel-Threads als einziges vom Betriebssystem unterstütztes Konzept für parallele Abläufe innerhalb eines Adreßraums.

4.5.2 Thread-Scheduling-Konzepte erweiterbarer Betriebssysteme

Die im vorangehenden Abschnitt beschriebenen Lösungsansätze, die eine bessere Integration des User-Level-Schedulings auf Grundlage einer Interaktion zwischen Betriebssystemkern und User-Level ermöglichen, sind selbstverständlich auch auf Basis der Kernel-Thread-Konzepte moderner erweiterbarer Betriebssystemarchitekturen implementierbar. In vielen Fällen wird dies aufgrund einer besseren Strukturierung neuerer Betriebssysteme sogar einfacher zu realisieren sein. Von Interesse ist jedoch die Frage, ob nicht schon die Erweiterungsmechanismen solcher Architekturen geeignet sind, um bezüglich Effizienz und Flexibilität vergleichbare Lösungen zu entwickeln. Als Beispiele dienen erneut die Betriebssysteme SPIN und Exokernel, deren Threadkonzepte im folgenden näher betrachtet werden.

4.5.2.1 SPIN

Entsprechend der Konzeption des Betriebssystems SPIN (siehe Abschnitt 3.1.3.2) werden anwendungsspezifische Scheduler als Benutzercode in den Systemkern geladen. Ereignisse wie zum Beispiel Blockierungen und Deblockierungen von Threads werden durch Signale an den zuständigen Scheduler realisiert, der - etwa durch Umschaltung auf einen anderen Thread der Anwendung - entsprechend reagieren muß. Der globale Systemscheduler wird nur aktiv, wenn ein Prozessor einer anderen Anwendung und damit einem anderen Scheduler zugeteilt werden soll. SPIN erlaubt dadurch flexible Schedulingstrategien für Kernel-Threads, bietet aber keine Mechanismen für eine Systemanbindung von User-Level-Threads an.

Solche Mechanismen können jedoch auf der Ebene eines anwendungsspezifischen SPIN-Schedulers realisiert werden, der die notwendigen Kommunikationsmechanismen mit einem User-Level-Scheduler implementieren kann. Die Voraussetzungen, um beispielsweise Blockierungen im User-Level behandeln zu können, sind dadurch gegeben, daß Blockierungen durch den anwendungsspezifischen Scheduler und nicht durch den Systemscheduler implementiert werden. Für eine Propagierung in den User-Level ist nur noch ein entsprechender anwendungsspezifischer Scheduler notwendig. Eine Systemintegration von User-Level-Threads ist folglich im Rahmen der Erweiterungsmechanismen von SPIN zumindest zum Teil realisierbar.

4.5.2.2 Exokernel

Unter der Exokernel Architektur (siehe Abschnitt 3.1.3.1) wird das Sichern und Restaurieren des Prozessorkontextes in ein im User-Level ablaufendes Bibliotheksbetriebssystem verlagert; gleiches gilt für Blockierungen und Deblockierungen. Ebensowenig wie SPIN unterstützt die Exokernel Architektur Konzepte für ein integriertes User-Level-Scheduling direkt. Ein entsprechendes Bibliotheksbetriebssystem kann jedoch entsprechende Ereignisse an einen User-Level-Scheduler propagieren und durch eine geeignete Schnittstelle dem User-Level-Scheduler Möglichkeiten zur Steuerung einräumen. Im Gegensatz zu SPIN, das die Integration eines anwendungsspezifischen Schedulers, der solche Mechanismen unterstützen könnte, in einfacher Weise erlaubt, ist bei einem Exokernel jedoch die weitgehende Reimplementierung des Schedulers eines Standard-Bibliotheksbetriebssystems notwendig.

4.5.2.3 Neue Wege des Thread-Schedulings?

Beide Systeme, SPIN und Exokernel, legen den Schwerpunkt auf die Erweiterbarkeit. Den in Kapitel 2 entwickelten Anforderungen an Flexibilität, Adaptierbarkeit und Erweiterbarkeit werden sie damit, auch bezüglich des Schedulings, zum größten Teil gerecht. Der weitgehende Einfluß auf das Thread-Scheduling, den beide Systeme ermöglichen, ersetzt einerseits eine Realisierung im User-Level, andererseits kann durch die in beiden Systemen sehr unterschiedliche Einbeziehung des Kerns die Effizienz eines User-Level-Schedulings nicht erreicht werden. Im Fall von SPIN werden anwendungsspezifische Strategien in den Kern geladen, wodurch der Overhead für den Einsprung in den Kern bestehen bleibt. Die Exokernel-Architektur verlagert zwar die Sicherung und Restaurierung des Threadkontextes in ein Bibliotheksbetriebssystem im User-Level, das Scheduling wird jedoch weiterhin im Kontext des Kerns durchgeführt. In beiden Systemen können die Erweiterungsmechanismen allerdings genutzt werden, um die Integration eines echten User-Level-Schedulings zu erleichtern.

Die Problematik des Schedulings, insbesondere in Multiprozessorimplementierungen, spielt zur Zeit in beiden Systemen eine noch untergeordnete Rolle. Beide Systeme dienen hauptsächlich der Evaluierung des jeweiligen Erweiterungskonzepts, auf die sich folglich auch die Veröffentlichungen konzentrieren. Eine Einschätzung, inwieweit die Schedulingkonzepte sich zu einer für Multiprozessorsysteme tragfähigen Lösung entwickeln können, ist auf Basis des verfügbaren Materials kaum möglich.

4.5.3 Threadscheduling auf Hardware-Ebene

Eine Alternative zum durch Software realisierten Threadscheduling auf Ebene des Kernel- oder User-Level stellen die sogenannten *multithreaded* Prozessoren dar. Threads werden direkt durch die Hardware unterstützt, indem ein Prozessor mehrere unabhängige Threadkontexte verwalten kann. Für jeden Threadkontext sind eigene Registersätze, Stackpointer und Programmzähler verfügbar, so daß der Prozessor auf Instruktionsebene zwischen den geladenen Threads umschalten kann. Dadurch wird eine Threadumschaltung möglich, wenn beispielsweise ein Thread beim Lesen eines Speicherwortes blockiert wird. In konventionellen Systemen geht die Wartezeit auf das Speichersubsystem in Form von Wartezyklen verloren. Um eine Umschaltung zwischen Threads verschiedener Prozesse zu ermöglichen, können Prozessoren ebenfalls mehrere Prozeßkontexte in Form unabhängiger Registersätze für die Speicherverwaltungseinheit unterstützen.

Multithreaded Prozessoren sind besonders gut geeignet, um Latenzzeiten bei Speicher- oder anderen Zugriffen außerhalb des hochgetakteten Prozessors zu verbergen. Aufgrund der möglichen Nutzung von Wartezyklen durch andere Threads sinkt die Bedeutung der Caches. Das Tera-System beispielsweise verzichtet vollständig auf Caches und ermöglicht ein Verbergen der Latenzzeiten durch die Unterstützung von 128 Threads pro Prozessor. Damit ist trotz einer durchschnittlichen Latenz von 70 Zyklen eine Auslastung des Prozessors zu 100% theoretisch erreichbar. Erreichbar ist eine solche Auslastung allerdings nur bei einer hinreichenden Parallelität innerhalb einer oder mehrerer - unterstützt durch 16 Prozeßkontexte pro Prozessor - Anwendungen.

Moderne superskalare Prozessoren ermöglichen die parallele Ausführung mehrerer Instruktionen eines Threads durch mehrere Funktionseinheiten und Pipelining. Aufeinanderfolgende Instruktionen können parallel ausgeführt werden, solange nicht die gleichen Funktionseinheiten benötigt werden oder Datenabhängigkeiten zwischen einzelnen Instruktionen eine sequentielle Ausführung erzwingen. Daher können parallele Funktionseinheiten im allgemeinen nicht voll ausgenutzt werden. Auch das Tera-System bietet diese Form der parallelen Abarbeitung von maximal drei Instruktionen neben der Unterstützung mehrerer Threads an. Die parallele Ausführung ist jedoch nur innerhalb eines Threads möglich. Prozessoren, die eine parallele Ausführung von Instruktionen auch verschiedener Threads ermöglichen (*simultaneous multithreading*) wurden von Tullsen et al. simuliert. Die Ergebnisse zeigen deutliche Leistungssteigerungen gegenüber superskalaren und *multithreaded* Architekturen [TuEgLe95].

Multithreaded Architekturen haben drastische Auswirkungen auf die Schedulingverfahren in Betriebssystem und User-Level. Statt zu bestimmen, welcher Thread auf einem Prozessor ausgeführt wird, muß entschieden werden, welche Prozeß- und Threadkontexte in einen Prozessor geladen werden. Schedulingverfahren für *multithreaded* Architekturen werden unter anderem von Fiske et al. [FisDal95], die Schedulingstrategien des Tera-Systems von Alverson et al. [AKKM+95] vorgestellt.

4.6 Zusammenfassung

Moderne Multiprozessorsysteme stellen besondere Anforderungen an zentrale Bereiche der Systemsoftware. Nicht nur Prozeß- und Threadscheduling müssen den Anforderungen von Multiprozessorsystemen gerecht werden, auch in anderen Bereichen muß über neue Konzepte nachgedacht werden. Einen der wichtigsten Bereiche neben Prozeß- und Threadscheduling selbst stellt die Speicher- und Cacheverwaltung dar.

Um leistungsfähige Multiprozessorsysteme adäquat ausnutzen zu können, ist insbesondere die Realisierung effizienter Mechanismen zur parallelen Ausführung einzelner als auch mehrerer Anwendungen wichtig. Bei speichergekoppelten Systemen steht das Konzept der Threads im Mittelpunkt des Interesses. In heutigen Systemen finden sich im wesentlichen zwei Realisierungen von Threads: Threads auf Systemebene (Kernel-Threads) zeichen sich durch eine gute Systemintegration aus, während Realisierungen außerhalb des Systemkerns (User-Level-Threads) sehr viel effizienter sind. Für einen großen Teil der Anwendungen wäre ein Mittelweg die bessere Lösung. Obwohl eine bessere Systemintegration effizienter User-Level-Threads, wie eine Vielzahl von Arbeiten in den letzten Jahren zeigt, durch die Integration neuer Schedulingkonzepte in den Systemkern möglich wäre, bieten heutige kommerzielle Betriebssysteme weiterhin Kernel-Threads als einziges thread-basiertes Parallelisierungskonzept mit Systemunterstützung an.

Erweiterbare Betriebssystemarchitekturen haben den Status von Forschungsprojekten noch nicht überwunden. Scheduling, insbesondere auf Multiprozessorsystemen, spielt in diesem Forschungsbereich eine noch untergeordnete Rolle. Die Entwicklung spezieller Rechnerarchitekturen zur direkten Unterstützung von Threads steckt ebenfalls noch in den Anfängen und leistungsfähige Systeme sind erst seit kurzem verfügbar. Solche Hardware birgt die Gefahr, daß sie nicht mehr allgemein einsetzbar ist, weil sie für nicht, schlecht oder nicht in geeigneter Form parallelisierbare Anwendungen keine akzeptable Rechenleistung erbringt.

Die Zukunft des Thread-Schedulings für leistungsfähige Multiprozessorsysteme ist noch offen. Geeignete Anwendungen für ein effizientes, aber nicht integriertes User-Level-Scheduling oder für Kernel-Threads wird es weiterhin geben. Welche Ansätze sich zu einer Alternative entwikkeln, ist nicht absehbar. Im Bereich des anwendungsspezifischen Schedulings können dies sowohl integrierte User-Level-Scheduler[8] als auch an Anwendungen adaptierbare effiziente Schedulingkonzepte auf Systemebene sein. Weitgehend unabhängig von dieser Fragestellung werden neue Schedulingaspekte Eingang in die globalen Strategien finden. Zunächst parallel wird sich ebenfalls der Bereich der *multithreaded* Architekturen weiterentwickeln.

8. Nachträgliche Systemerweiterungen zur Integration des User-Level-Schedulings werden sicher keine Zukunftsaussicht haben. Denkbar sind aber Systeme, die Integrationskonzepte als integralen Basisdienst anbieten.

5 Das ELiTE-Projekt

Das ELiTE[1]-Projekt ist eine Zusammenfassung verschiedener Forschungsarbeiten im Bereich des Threadschedulings für speichergekoppelte Multiprozessoren am Lehrstuhl für Betriebssysteme der Friedrich-Alexander Universität Erlangen-Nürnberg. Zielsetzung des Projekts ist die Untersuchung des Verhaltens einzelner und insbesondere des Zusammenspiels verschiedener Mechanismen und Strategien des Thread-Schedulings sowohl im Kernel- als auch im User-Level. Die Schwerpunkte liegen dabei zum einen auf einem Threadscheduling, das die gegenseitigen Abhängigkeiten zwischen Scheduling und Speicher- bzw. Cachearchitekturen berücksichtigt, und zum anderen in der Integration von Kernel- und User-Level-Scheduling.

Nach einer Vorstellung der Hard- und Softwareplattformen des ELiTE-Projekts, wird in diesem Kapitel zunächst ein kurzer Überblick über ELiTE und die einzelnen Teilprojekte gegeben. Als Grundlage für die in Kapitel 6 und 7 behandelten Teilprojekte aus dem Bereich der Systemintegration des User-Level-Schedulings und effizienter Unterbrechungsmechanismen wird anschließend das im engen Zusammenhang mit dieser Arbeit stehende Teilprojekt *m-threads* eingehender beschrieben.

5.1 Moderne Schedulingverfahren für UNIX-Systeme

Die Entscheidung für kommerzielle standardisierte UNIX-Plattformen als Basis für das ELiTE-Projekt ergibt sich als konsequente Folge aus der Situationsanalyse in den vorausgehenden Kapiteln.

Experimentelle erweiterbare Betriebssystemarchitekturen zeigen in einigen Bereichen keinen entscheidenden konzeptionellen Vorteil, in anderen könnte die Erweiterbarkeit zu einer konzeptionell sauberen Integration von Schedulingmechanismen genutzt werden. Der Vorteil gegenüber einer Integration in die Standardbetriebssysteme der innerhalb von ELiTE eingesetzten Re-

1. ELiTE - **E**rlangen **Li**ghtweight **T**hread **E**nvironment.

chensysteme ist jedoch nicht entscheidend. Auf der Seite der Standardsysteme überwiegen letztlich rein praktische Argumente.

Weder die vorgestellten noch andere erweiterbare Betriebssystemarchitekturen sind für die im ELiTE-Projekt eingesetzten Multiprozessorsysteme verfügbar. Die Portierung eines erweiterbaren Systems auf eine neue Architektur - insbesondere eine Multiprozessorarchitektur - stände in keinem Verhältnis zum Umfang des eigentlichen Projekts und ist mit den verfügbaren Ressourcen nicht realisierbar. Aufgrund von Kooperationen mit den Hardwareherstellern ist jedoch der Quellcode der Standardsysteme verfügbar und damit die Voraussetzung zur Integration neuer Schedulingverfahren gegeben. Die Einschränkungen bezüglich einer möglichen Verbreitung und weiteren Nutzung der Ergebnisse gegenüber frei verfügbaren Forschungssystemen müssen hingenommen werden.

UNIX wird, vor allem im Bereich von Hochleistungsservern, noch über mehrere Jahre eine beherrschende oder zumindest bedeutende Rolle spielen. Auch aus diesem Grund ist es wichtig, UNIX-Systeme weiterzuentwickeln und zu modernisieren. Möglich ist dies bei Betriebssystemarchitekturen wie UNIX letztendlich nur in den Entwicklungsabteilungen. Forschungsarbeiten auf Basis von UNIX-Systemen sind jedoch eine wichtige Triebfeder für eine Weiterentwicklung. Das ELiTE-Projekt zeigt, daß auch in heutige kommerzielle Betriebssysteme eine moderne Schedulingarchitektur integriert werden kann.

5.2 Plattformen

Im Rahmen des ELiTE-Projekts stehen zwei speichergekoppelte Multiprozessorsysteme zur Verfügung. Die Convex/HP SPP Architektur ist ein NUMA-System mit einem möglichen Ausbau von bis zu 128 Prozessoren. Die Systemsoftware basiert auf dem MACH Mikrokern, der eine Anpassung des darauf aufbauenden UNIX-Betriebssystems an die NUMA-Architektur erlaubt. Das zweite System ist ein moderat paralleles Multiprozessorsystem mit uniformem Speicherzugriff der Firma Sun (Ultra Enterprise 3000) unter dem Betriebssystem Solaris. Beide Systeme werden im folgenden beschrieben. Auf das Convex/HP System und SPP-UX wird als Grundlage für die folgenden Kapitel detaillierter eingegangen, bezüglich des Sun Systems werden nur die wichtigsten Eigenschaften zur Abgrenzung angeführt.

5.2.1 Hardware

5.2.1.1 Convex/HP SPP 1000/1600

Die Basiseinheit des SPP Systems [Convex93] bildet der sogenannte Hypernode. Ein Hypernode, im Prinzip ein symmetrischer achtfach Multiprozessor, wird aus vier Prozessor-Speicher-Modulen und einem I/O Subsystem gebildet, die über einen nichtblockierenden fünffach Crossbar gekoppelt sind. Die Latenzzeit für einen Speicherzugriff innerhalb eines Nodes ist mit 500[2] Nanosekunden (nsec) uniform, unabhängig davon, ob ein Prozessor auf den Speicher des eige-

2. Alle angegebenen Werte gelten für den SPP 1000.

nen oder eines fremden Moduls zugreift. Ein Prozessor-Speicher-Modul setzt sich aus zwei mit 100 MHz getakteten HP PA-RISC 7100 Prozessoren mit jeweils 1 MByte virtuell indizierten, *direct-mapped* Daten- und Instruktionscache und bis zu 512 Mbyte Speicher zusammen. Die beiden Prozessoren sind über eine Agent genannte Schnittstelle an den Crossbar angebunden, der Speicher und eine spezielle Schnittstelle zur Koppelung mehrerer Hypernodes (CTI[3]) über eine Speicherschnittstelle (CCMC[4]), die die notwendigen Cachekonsistenzprotokolle realisiert.

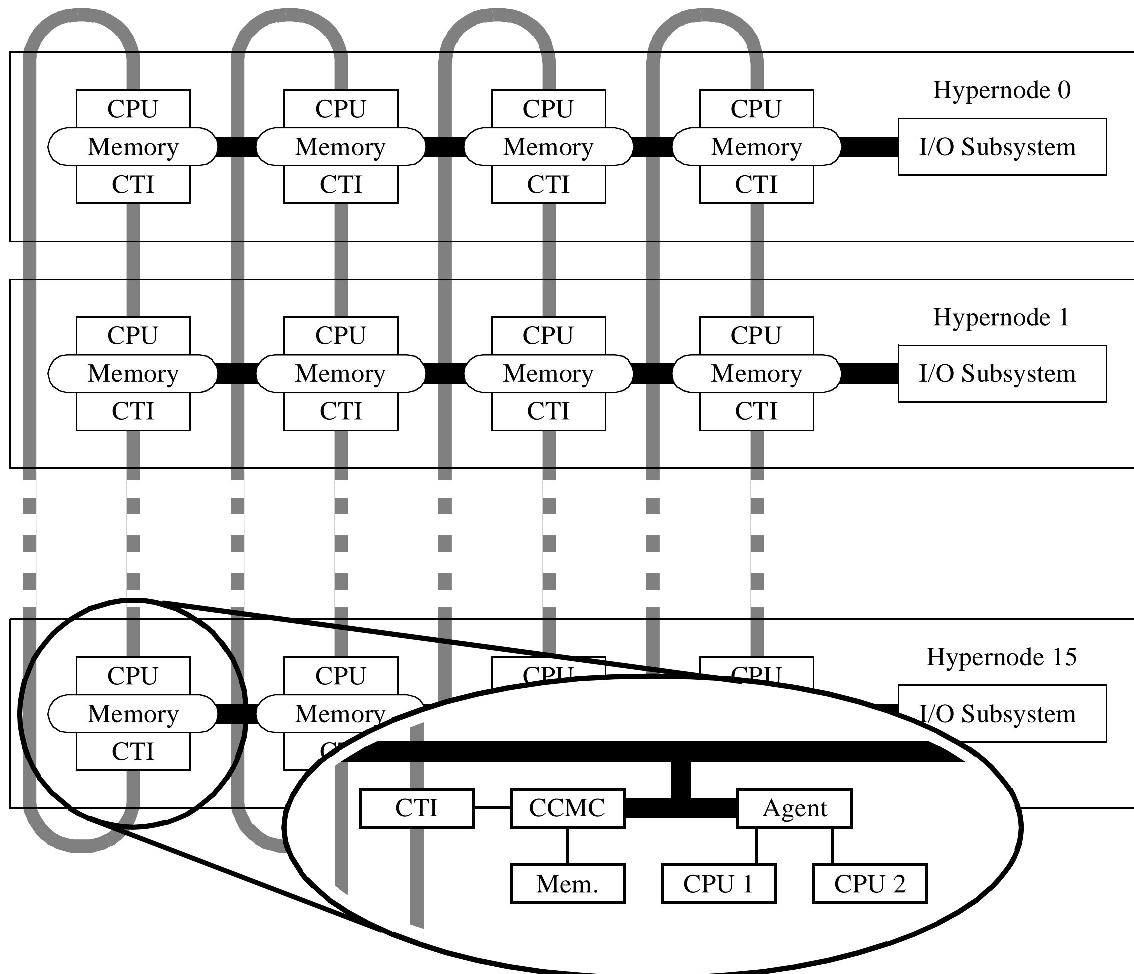

Abb. 5.1: Aufbau der Convex/HP SPP Architektur

Bis zu 15 Hypernodes können über vier parallele CTI-Ringe zu einem NUMA System gekoppelt werden. Die vierfach ausgelegten Ringe dienen nicht zur Verbesserung der Ausfallsicherheit, sondern zur Erhöhung des Durchsatzes beim Zugriff auf entfernten Speicher. CTI ist eine Implementierung des IEEE SCI[5] Standards für Ringnetzwerke zum cachekohärenten Zugriff auf entfernten Speicher. Jeder der vier CTI-Ringe hat eine Datenübertragungsrate von 600 MByte/sec, die Latenzzeit ist mit 2000 nsec deutlich höher als bei einem lokalen Speicherzugriff innerhalb eines Hypernodes. Um die mittlere Speicherzugriffszeit zu vermindern, wird ein

3. **C**onvex **T**orodial **I**nterface
4. Cache **C**oherent **M**emory **C**ontroller
5. **S**calable **C**oherent **I**nterface

Teil des lokalen Speichers jedes Hypernodes als physikalisch indizierter CTI-Cache (Netzwerkcache) verwendet. Die Architektur des SPP Systems ist in Abbildung 5.1 dargestellt.

Der theoretische Maximalausbau eines SPP Systems mit 15 Hypernodes liegt bei 128 Prozessoren und 32 GByte Speicher. An der Universität Erlangen-Nürnberg stehen ein SPP 1000 mit einem Hypernode (8 CPUs, 512 MByte Speicher) sowie ein im Laufe des ELiTE-Projekts auf SPP 1600[6] aufgerüstetes System mit sechs Hypernodes (48 CPUs, 6 Gbyte Speicher) zur Verfügung. Das Multinode-System wird im Produktionsbetrieb am Rechenzentrum der Universität eingesetzt und steht daher für Entwicklungsarbeiten im Rahmen des ELiTE-Projekts nur eingeschränkt zur Verfügung.

Das SPP System zeigt eine für NUMA Architekturen typische Cache- und Speicherhierarchie. Die Latenzzeiten von 10, 500 und 2000 nsec für den Prozessorcache, lokalen und entfernten Speicher entsprechen einem, 50 und 200 Prozessorzyclen (bei 100 MHz). Aufgrund der komplexen Speicheranschaltung ist schon die Latenzzeit für lokale Speicherzugriffe mit 50 Zyklen relativ hoch, bei Zugriffen auf entfernten Speicher steigt sie noch einmal um das vierfache. Ein hoher Nutzungsgrad der Caches innerhalb und zwischen den Hypernodes ist daher für die Leistungsfähigkeit des Systems sehr wichtig. Für Arbeiten in diesem Bereich ist die SPP Architektur sehr interessant.

5.2.1.2 Sun Ultra Enterprise 3000

Im Gegensatz zur NUMA-Architektur des SPP Systems stellt der Sun Ultra Enterprise Server [Sun96] eine UMA-Architektur dar. Er ist daher am besten mit einem einzelnen SPP-Hypernode vergleichbar. Den Grundbaustein des Systems bildet ebenfalls ein Modul mit zwei Prozessoren (Ultra Sparc I) und maximal 2 Gbyte Speicher. Statt eines Crossbars wird zur Koppelung der einzelnen Module ein paketvermittelter (*packed-switched*) Bus mit einem maximalen Durchsatz von 2,6 Gbyte/sec eingesetzt. Statt CPU/Speicher-Modulen können auch I/O-Module mit je zwei I/O-Bussen eingesetzt werden.

Der Maximalausbau eines Sun Ultra Enterprise 3000 Systems (vier Einschübe) liegt also bei 6 CPUs und 6 Gbyte Speicher mit nur einem I/O-Modul. Das 6000er Modell verfügt über 16 Einschübe. Das Enterprise 3000 System des Lehrstuhls für Betriebssysteme war bei den Testreihen mit zwei bzw. vier Prozessoren ausgestattet.

Ein weiterer Unterschied zum SPP System liegt in der Cache-Architektur. Jeder Prozessor verfügt über zwei Cache-Ebenen. Der interne *first-level* Cache besteht aus je 16 Kbyte Daten- und Instruktionscache. Der Datencache ist *direct-mapped*, der Instruktionscache zweifach mengenassoziativ ausgelegt. Ein 512 Kbyte großer externer *direct-mapped* Cache dient als *second-level* Cache für Daten und Instruktionen. Alle Caches sind physikalisch indiziert.

6. Die Hauptunterschiede zum SPP 1000 sind der HP PA-RSIC 1200 Prozessor und die höhere Taktrate von 120 MHz.

5.2.2 Systemsoftware

5.2.2.1 SPP-UX

SPP-UX, das Betriebssystem des SPP, basiert auf OSF/1 AD. OSF/1 AD ist eine Multiserver UNIX Implementierung der Open Software Foundation auf Basis von MACH 3.0. Die modulare Struktur von OSF/1 erlaubt die Realisierung eines Betriebssystems, das an die Hypernode-Struktur der NUMA-Architektur SPP angepaßt ist und auch auf größeren Konfigurationen keine relevanten Leistungseinbrüche zeigt. Auf jedem Hypernode laufen eine unabhängige Instanz des MACH-Mikrokerns und ein oder mehrere Server, die zusammen das auf dem MACH-Kern aufsetzende UNIX-System realisieren. Die UNIX-kompatible Schnittstelle des Systems baut auf dem Konzept des Emulators auf (siehe Abschnitt 3.1.2).

Als MACH-basiertes System bietet OSF/1 Kernel-Threads an. Im Rahmen der Anpassung an die Hypernode-Struktur besteht in SPP-UX jedoch die Einschränkung, daß Kernel-Threads nicht über Hypernodegrenzen hinweg migrieren können. Der Grund für diese Einschränkung liegt in der Repräsentation eines UNIX-Prozesses, der mehrere Hypernodes umspannt. Ein UNIX-Prozeß setzt sich aus je einer Task eines unabhängigen Mach-Kerns auf jedem umspannten Hypernode zusammen.

Neben der Anpassung an die Hypernode-Struktur wurde OSF/1 in weiteren Bereichen modifiziert oder erweitert. Die wichtigsten Modifikationen und Erweiterungen von SPP-UX gegenüber OSF/1 werden im folgenden beschrieben.

Die Programmierschnittstelle (API) von OSF/1 wurde durch eine HP-UX[7] kompatible Schnittstelle mit Erweiterungen für die parallele Programmierung ersetzt. Die binäre Anwendungsschnittstelle (ABI) ist ebenfalls, bis auf geringfügige Ausnahmen, HP-UX kompatibel. Fast alle HP-UX Programme sind damit unverändert auch unter SPP-UX ablauffähig.

SPP-UX unterstützt eine statische Partitionierung des Systems. Einer Partition (Subcomplex) können Prozessoren verschiedener Hypernodes zugeteilt werden. Im Fall einer Partition, die sich über mehrere Hypernodes erstreckt, muß zusätzlich konfiguriert werden, wieviel Speicher jedes beteiligten Hypernodes für den über Hypernodegrenzen zugreifbaren globalen Speicher der Partition genutzt wird. Auf den restlichen Speicher kann nur innerhalb des jeweiligen Nodes zugegriffen werden. Diese ermöglicht einerseits eine eingeschränkte Speicherreservierung, falls sich mehrere Partitionen über einen Hypernode erstrecken, andererseits werden durch die statische Aufteilung zwischen lokalem Speicher eines Hypernodes und global zugreifbarem Speicher zusätzliche Kapazitätsgrenzen gezogen. Die Implementierung der Prozessorzuteilung baut auf den Prozessormengen (Processorsets) von MACH auf.

Um die Effizienzprobleme der nachrichtenbasierten Kommunikation zwischen Mikrokern, Servern und Emulator zu verringern, waren Änderungen an der Betriebssystemarchitektur notwendig. Im Gegensatz zu einem herkömmlichen MACH-basierten UNIX-System werden nicht nur der Kern, sondern auch die Server und der Emulator unter einer höheren Privilegierungsstufe[8]

7. HP-UX ist das Betriebssystem der HP PA-RISC basierten UNIX Rechner von Hewlett Packart.
8. Die HP PA-RISC Architektur erlaubt mehrere Privilegierungsstufen. Die höchste Privilegierungsstufe bleibt weiterhin dem Kern vorbehalten.

als eine Anwendung ausgeführt. Aufgrund des daraus resultierenden Schutzes des Emulators vor Zugriffen der Anwendung und der Vertrauenswürdigkeit von Server und Emulator konnten verschiedene Optimierungen vorgenommen werden [BiGoWa94a].

Funktionalität und Datenstrukturen sind soweit wie möglich aus dem Server in den Emulator verlagert, um zeitaufwendige Kontrolltransfers in den Server zu vermeiden. Durch den privilegierten Status des Emulators ist dies in einem weitaus größeren Umfang als bisher möglich.

Der Authentifizierungsmechanismus von MACH beruht auf dem Recht, Nachrichten an bestimmte Ports senden bzw. Nachrichten für Ports empfangen zu dürfen. Auf Basis eines solchen Konzepts war es möglich, Betriebssystemfunktionalität in einem ansonsten nicht weiter privilegierten Serverprozeß zu implementieren. Aufgrund der Vertrauenswürdigkeit von Server und Emulator kann auf eine Authentifizierung über den Nachrichtenmechanismus verzichtet und ein effizienterer Kommunikationsmechanismus für den Kontrolltransfer realisiert werden. SPP-UX ersetzt den MACH Nachrichtenmechanismus für die Kommunikation zwischen den einzelnen Betriebssystembestandteilen weitgehend durch einen effizienten RPC[9]-Mechanismus. Der RPC wird über einen Trap ausgelöst. Innerhalb des Traphandlers wird eine effiziente in Assembler implementierte Threadumschaltung auf einen freien Thread des aufgerufenen Servers oder des MACH Kerns durchgeführt, wobei die notwendigen Argumente in Registern übergeben werden. Der aufrufende Thread (Client-Thread) blockiert, bis der Server-Thread die gewünschte Funktion ausgeführt hat. Die Übergabe der Ergebniswerte erfolgt wiederum über Register im Rahmen einer erneuten Threadumschaltung. Aufrufe über Hypernodegrenzen hinweg können ebenfalls über den RPC-Mechanismus realisiert werden [BiGoWa94b]. In diesem Fall wird ein sehr effizienter, von der SCI-Hardware unterstützter Nachrichtenmechanismus [Convex93] zur Übertragung der Argumente an den anderen Hypernode genutzt.

5.2.2.2 Solaris

Das Betriebssystem Solaris 2.6 basiert auf System V Release 4 [GooCox93] und ist um Kernel-Threads erweitert. Solaris ist also im Gegensatz zu dem mikrokern-basierten SPP-UX eine monolithische UNIX-Architektur. Die Schedulingklassen von Solaris, eine weitere Erweiterung gegenüber dem Basissystem, erlauben in eingeschränkter Form eine einfache Integration alternativer Schedulingstrategien in den Betriebssystemkern. Dieser Mechanismus wird innerhalb des ELiTE-Projekts für die Implementierung des *Follow-On* Schedulings und der Speicherbandbreitenreservierung genutzt.

5.3 ELiTE Teilprojekte

5.3.1 Anwendungsinternes Scheduling

Der Bereich des anwendungsinternen Schedulings im ELiTE-Projekt umfaßt eine an moderne Multiprozessorsysteme angepaßte Threadverwaltung im User-Level und Betriebssystemmecha-

9. **R**emote **P**rocedure **C**all

nismen, die eine bessere Integration des User-Level-Schedulings mit dem Betriebssystem erlauben.

5.3.1.1 *m(icro)-threads*

Die *m-threads* [Stecker95] sind eine auf dem *Quick-Thread* Threadumschalter [Keppel93 und Reder95] basierende User-Level-Threadbibliothek, die speziell für numerische Anwendungen auf NUMA-Prozessoren entwickelt wurde. Bezüglich der Softwarearchitektur der *m-threads* Bibliothek sind die dezentralen Datenstrukturen zu erwähnen, die einen einer cache-basierten Architektur angemessenen Umgang mit internen Verwaltungsinformationen erlauben. Die Softwarearchitektur der *m-threads* ist an die hierarchische Architektur von NUMA-Rechnern angepaßt. Die Schedulingstrategien der *m-threads* Bibliothek nutzen Cachemiss-Zähler der Hardware zur Laufzeit aus, um den Ablauf einer Anwendung hinsichtlich der Wiederverwendung von Cacheinhalten zu optimieren. Es stehen Versionen für beide Plattformen des ELiTE-Projekts (Sun UltraSparc und Convex/HP SPP) zur Verfügung. Struktur und Strategien der *m-threads* Bibliothek werden in Abschnitt 5.4 näher beschrieben.

5.3.1.2 *Sleeping-Threads*

Sleeping-Threads [Koppe94, Koppe95, und Hollma96] sind eine Erweiterung des Threadschedulingmechanismus des Betriebssystemkerns, die eine bessere Integration von User-Level-Threadbibliotheken ermöglicht. Die *Sleeping-Threads* basieren auf innerhalb des Betriebssystemkerns in Reserve gehaltenen Threads und einer Interaktion zwischen den beiden Schedulinginstanzen. Dies ermöglicht innerhalb einer User-Level-Threadbibliothek, auf Blockierungen von Threads zu reagieren und damit eine Abstraktion des virtuellen Prozessors zu erreichen, die dem Verhalten von Betriebssystemkern verwalteter physikalischer Prozessoren näherkommt. Auf Basis der *Sleeping-Threads* kann eine Großzahl der Probleme des User-Level-Schedulings vermieden werden. Der *Sleeping-Threads* Mechanismus wurde in das SPP-UX Betriebssystem integriert, die *m-threads* Bibliothek an den Integrationsmechanismus angepaßt. Kapitel 6 befaßt sich mit diesem Projekt.

5.3.1.3 Unterbrechungsmechanismen

Für das MACH-basierte Betriebssystem SPP-UX wurde ein Unterbrechungsmechanismus entwickelt, der in dieser komplexen Betriebssystemarchitektur eine effiziente benutzergesteuerte Unterbrechung bestimmter Kernel-Threads erlaubt. **E**LiTE **B**ou**nc**e (EBNC) [Reder96] erreicht eine im Vergleich mit dem in SPP-UX verfügbaren Signal-Mechanismus sehr gute Effizienz, indem auf überflüssigen Ballast verzichtet und nur eine minimale, aber für die angedachten Einsatzbereiche ausreichende Funktionalität realisiert wird. EBNC bildet eine Basis für unterbrechende, insbesondere zeitscheibenbasierte Schedulingstrategien im User-Level. Die Realisierung eines unterbrechenden User-Level-Schedulings auf Basis von EBNC wurde bisher nicht realisiert. Der Unterbrechungsmechanismus EBNC wird in Kapitel 7 diskutiert.

5.3.2 Anwendungsübergreifendes Scheduling

Im zweiten Bereich des ELiTE-Projekts werden Strategien untersucht und implementiert, die Aspekte von Cache- und Speichernutzung in das Threadscheduling im Betriebssystemkern einbeziehen. Im Gegensatz zum User-Level-Scheduling mit den entsprechenden Integrationsmechanismen werden hier Threads verschiedener unabhängiger Anwendungen einbezogen. Dieser Projektbereich wurde auf Basis des Solaris Betriebssystems von Sun verwirklicht. Dieses Teilprojekt wird im Rahmen der vorliegenden Arbeit nicht weiter betrachtet. Eine umfassende Diskussion der Thematik findet sich in [Bellos98].

5.3.2.1 *Follow-On* Scheduling

Eine potentielle wiederholte Nutzung bereits geladener Cacheinhalte ist nicht nur beim Scheduling zwischen Threads einer gleichen Anwendung anzustreben. Verwandtschaftsbeziehungen bezüglich der Nutzung gemeinsamer Speicherbereiche auch zwischen ansonsten unabhängigen Threads können ausgenutzt werden, um den Gesamtdurchsatz eines Rechensystems zu erhöhen. Verwandtschaftsbeziehungen bezüglich der Speicherzugriffe können aus unterschiedlichen Ursachen hervorgehen. Denkbar sind u.a. die mehrfache Ausführung unabhängiger Instanzen eines Programms, gemeinsame dynamisch gebundene Bibliotheken (shared libraries) oder ein ähnliches Profil bezüglich der Nutzung von Systemroutinen.

Beim *Follow-On* Scheduling [Bellos97b] werden zunächst solche Verwandschaftsbeziehungen auf der Ebene von Speicherseiten durch eine Analyse der TLB[10]-Misses der einzelnen Threads ermittelt. Soweit im Rahmen der Fairness vertretbar, wird die Ausführungsreihenfolge derart modifiziert, daß ab einem bestimmten Grad verwandte Threads nacheinander ausgeführt werden, um eine potentielle Nutzung des Cacheinhalts zu ermöglichen.

5.3.2.2 Speicherbandbreitenreservierung

In Zusammenhang mit der gewachsenen Bedeutung eines effizienten Umgangs mit den Ressourcen Speicher und Cache bei modernen Multiprozessorsystemen wurde in Abschnitt 4.1 bereits der Effekt der *memory preemption* genannt. Eine speicherintensive Anwendung kann in einer Multiprozessorumgebung trotz garantierter Prozessorzeit nicht schnell genug ausgeführt werden, weil Anwendungen auf anderen Prozessoren das Speichersystem zu stark belasten. Dieses Problem wird durch eine im Kernel-Scheduling integrierte Speicherbandbreitenreservierung angegangen [Bellos97a und Bellos98]. Die reservierte Bandbreite einer Anwendung wird garantiert, indem Threads anderer Anwendungen ohne Reservierung in ihren Speicherzugriffen entsprechend gebremst werden. Dies wird durch eine in ihrer Iterationsanzahl konfigurierbaren Schleife im TLB-Handler realisiert.

10. Translation lockaside buffer - Hardware-Cache für Pagetable-Einträge. Im TLB sind die, aus Sicht der Hardware, aktuell gültigen Abbildungen zwischen virtuellen und physikalischen Adressen eingetragen. Ist für eine referenzierte Speicherseite kein Eintrag enthalten (TLB-Miss), wird dieser durch einen in Hard- oder Software realisierten Handler eingetragen. Eine Analyse der TLB-Misses ist nur bei Softwarehandlern möglich.

5.3.3 Zusammenfassung

Abbildung 5.2 zeigt die Zusammenhänge zwischen den einzelnen Bestandteilen des ELiTE-Projekts. Das Kernel-Scheduling wurde um den Mechanismus des *Follow-On* Schedulings zur allgemeinen Durchsatzverbesserung und die Bandbreitenreservierung für speicherintensive Anwendungen mit schwachen Echtzeitbedingungen erweitert. Im User-Level Bereich des ELiTE-Projekts ist die Thread-Bibliothek *m-threads* angesiedelt, die eine effiziente Programmierung unter Berücksichtigung der Cache-Architektur moderner Multiprozessorsysteme erlaubt. Die Probleme, die in Zusammenhang mit User-Level-Scheduling auftreten, werden durch spezielle Betriebssystemmechanismen (*Sleeping-Threads* und EBNC), auf denen die *m-threads* Bibliothek aufbaut, vermieden. Die *Sleeping-Threads* erlauben eine mit Kernel-Threads vergleichbare Systemintegration; die effizienten EBNC-Timer ermöglichen unterbrechende oder zeitscheibenbasierte Schedulingstrategien im User-Level. Eine Integration unterbrechender Strategien in die *m-threads* Bibliothek ist nicht Bestandteil dieser Arbeit.

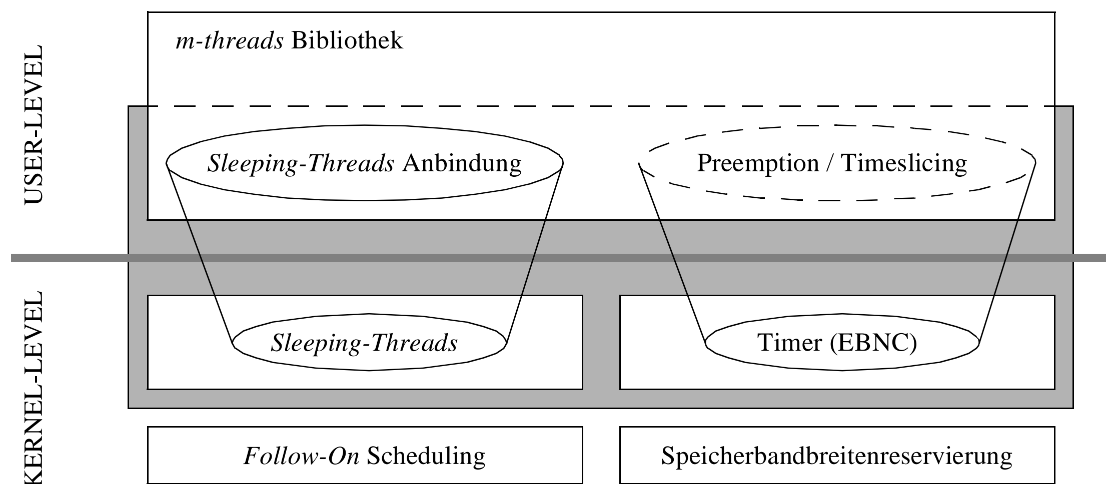

Abb. 5.2: Das ELiTE-Projekt

Die ELiTE-Umgebung ist jedoch nicht in dem hier beschriebenen Umfang einsetzbar, da nicht alle Mechanismen durchgehend auf einer Plattform realisiert sind. Die *m-threads* Bibliothek ist unter SPP-UX und Solaris verfügbar, *Sleeping- Threads* und EBNC nur unter SPP-UX, *Follow-On* Scheduling und Speicherbandbreitenreservierung sind nur unter Solaris realisiert. Die Gründe hierfür liegen unter anderem in der eingeschränkten Verfügbarkeit der SPP-Architektur für Entwicklungsarbeiten. Für die Idee des *Follow-On* Schedulings waren die physikalisch indizierten Caches der Sun-Architektur eine wichtige Voraussetzung. Der notwendige Implementierungsaufwand, um alle Teilprojekte auf beiden Architekturen zu verwirklichen, übersteigt die im Rahmen des ELiTE-Projekts verfügbaren Ressourcen.

Im Rahmen dieser Arbeit werden Integrations- und Unterbrechungsmechanismen (*Sleeping Threads* und EBNC) sowie die Anpassung der *m-threads* Bibliothek behandelt. Die *m-threads* Bibliothek selbst, das *Follow-On* Scheduling und die Speicherbandbreitenreservierung sind Bestandteil einer zweiten Dissertation, die im Rahmen des ELiTE-Projekts entstanden ist [Bellos98].

5.4 Struktur und Affinitätsmodelle der *m(icro)-threads* Bibliothek

Im folgenden wird die *m-threads* Bibliothek soweit beschrieben, wie es zum Verständnis der Integration von *m-threads* und *Sleeping-Threads* notwendig ist. Für eine ausführliche Darstellung wird auf die im weiteren genannten Literaturstellen verwiesen.

5.4.1 Struktur der *m-threads*

Die in Abbildung 5.3 dargestellte Struktur der *m-threads* ist in idealer Weise an die Struktur von NUMA-Architekturen, die aus Blöcken von symmetrischen UMA-Multiprozessoren (Nodes) aufgebaut sind, angepaßt. In der Struktur ist deutlich die Architektur des SPP Systems (Subcomplex, Hypernode, CPU) wiederzuerkennen. UMA-Architekturen, wie die Sun Ultra-Server, können als ein NUMA-System mit nur einem Node angesehen werden.

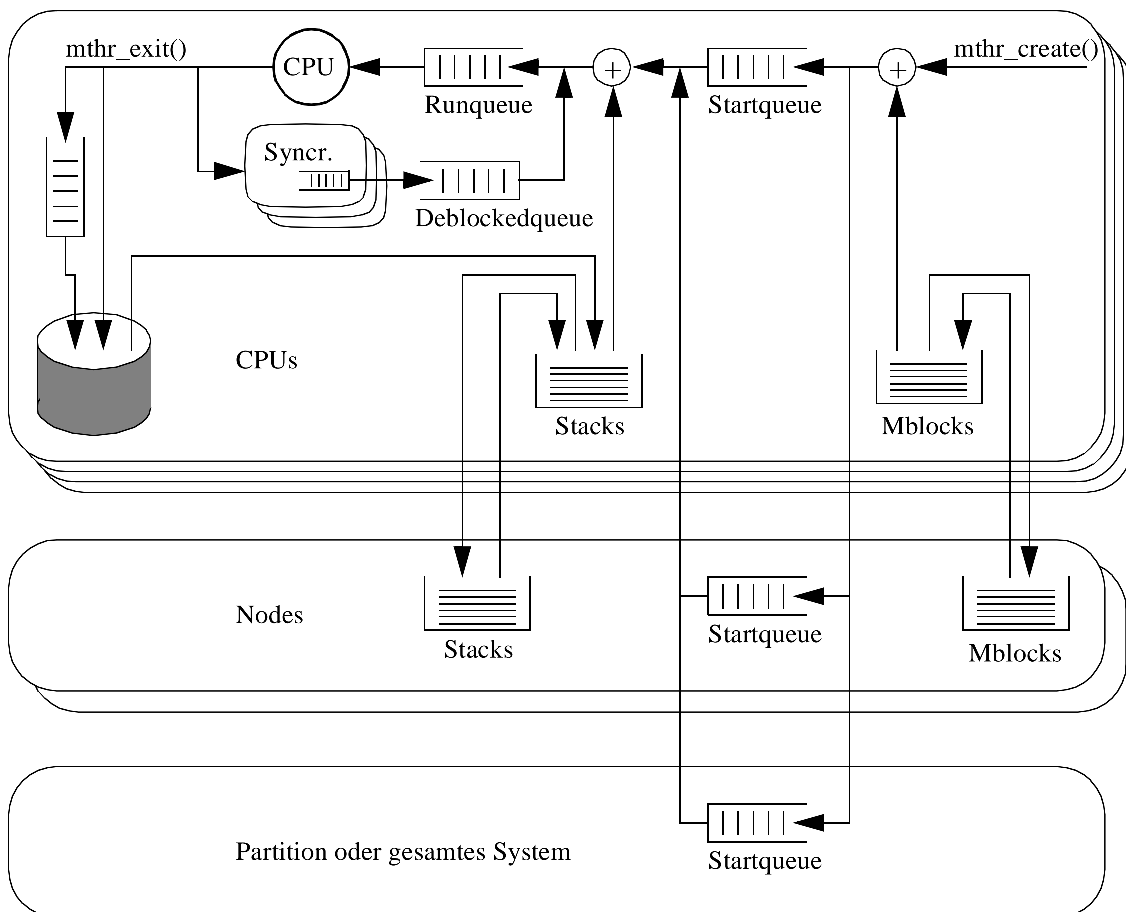

Abb. 5.3: Struktur der *m-threads* Bibliothek

Die Datenstrukturen sind dahingehend ausgelegt, daß sie nach Möglichkeit nur von jeweils einer CPU referenziert werden und somit Cache-Konflikte weitestgehend vermieden werden. Im folgenden werden die Auswirkungen einiger dezentraler Datenstrukturen beschrieben:

- Startqueues:
  Da die *m-threads* Bibliothek nur prioritätengesteuerte Strategien und keine Zeitscheiben-

verfahren implementiert, ist eine getrennte Verwaltung von noch nicht gelaufenen Threads in Startqueues sinnvoll. Insbesondere, da für diese Threads noch kein Stack angelegt ist, verfügen sie über annähernd keine Bindung aufgrund der Cache-Inhalte (Cacheaffinität) bezüglich einer CPU oder eines Nodes. Sie sind die idealen Kandidaten für die Migration im Rahmen eines Lastausgleiches.

- Speicherverwaltung:
  Die bibliotheksinterne Verwaltung von Speicher für Stacks und interne Datenstrukturen zeigt zwei Vorteile. Zum einen ermöglicht sie eine einfache Ausrichtung von Datenstrukturen auf Cachelines und vermeidet den Overhead der Speicherverwaltungsfunktionen der Systembibliothek. Wichtiger ist jedoch die Möglichkeit der Wiederverwendung von noch im Cache geladenen Speicherbereichen auf CPU- und Node-Ebene.
- Dezentrale Runqueues:
  Die Runqueue ist eine der zentralen Datenstrukturen eines Schedulers. In einer Multiprozessorimplementierung kommt es bei Zugriffen daher zwangsläufig zu Locking- oder Cachekonflikten. Um Behinderungen der CPUs untereinander zu vermeiden, arbeitet die *m-threads* Bibliothek mit dezentralen Runqueues. Jeder CPU ist eine eigene Runqueue zugeordnet, die nur im Rahmen eines Lastausgleiches von anderen CPUs referenziert wird. Durch die lokalen Runqueues wird auch schon ohne weitere Betrachtung des Cachezustandes ein sehr viel besseres Cacheverhalten erreicht, da Migrationen nur durch einen Lastausgleich möglich sind und nicht unkontrolliert auftreten.
- Deblockedqueues
  Im Rahmen einer Synchronisation blockierte Threads werden im Fall einer Deblockierung normalerweise wieder in die Runqueue eingehängt. Dies würde jedoch mit einer hohen Wahrscheinlichkeit zu einer Referenzierung der Runqueue durch eine andere CPU führen. Um dies zu vermeiden, werden deblockierte Threads zunächst in einer Deblokkedqueue verwaltet. Bei jedem Threadswitch überprüft eine CPU ihre Deblockedqueue und verlagert eventuell enthaltene Threads in ihre Runqueue.

Für eine detailliertere Diskussion und eine Beschreibung aller in Abbildung 5.3 dargestellten Strukturen wird auf [Stecker95] verwiesen.

Die Auswahl des nächsten auszuführenden Threads erfolgt auf Basis des Affinitätsmaßes, das durch eines der im nächsten Abschnitt beschriebenen Modelle definiert wird. Dem Thread mit der höchsten Cacheaffinität wird die höchste Priorität zugeordnet. Das Scheduling ist nicht unterbrechend, das heißt, daß ein Threadwechsel nur möglich ist, wenn der laufende Thread innerhalb einer Synchronisation blockiert wird oder die CPU freiwillig aufgibt. Nur in dem Fall, daß die eigene Runqueue und alle Startqueues leer sind, wird ein Thread aus der Runqueue einer anderen CPU entnommen. Um einen geeigneten Thread zu finden, wird der positive Effekt des Lastausgleiches mit dem negativen Effekt durch den Verlust an Cacheaffinität aufgrund der Migration verglichen. Es erfolgt keine Suche nach einem globalen Optimum, sondern beginnend bei den Runqueues des eigenen Nodes wird nach dem ersten Thread gesucht, dessen Migration einen insgesamt positiven Effekt erwarten läßt. Dadurch wird eine zu starke Belastung durch den Lastausgleichalgorithmus vermieden.

Messungen auf einer Convex/HP SPP zeigen, daß diese Maßnahmen auch ohne Affinitätsbestimmung, das heißt mit einer einfachen FCFS[11]-Strategie gegenüber zentralen Strukturen zu durchweg positiven Ergebnissen führen. Mit einer synthetischen Anwendung, die nur Synchronisationsoperationen und damit Threadwechsel ausführt, kann gezeigt werden, daß schon auf einem Node mit acht Prozessoren eine ca. 40-fach höhere Zahl von Synchronisationsoperationen pro Sekunde erreicht werden kann. Bei Verwendung von vier Nodes steigt der Faktor auf ca. 120. Für reale Anwendungen kann am Beispiel einer Jacobi-Iteration gezeigt werden, daß Leistungssteigerungen bis zu 40% erreicht werden können. Bei Verwendung von nur einem Node fallen auch hier die Ergebnisse weniger deutlich aus. Detailliertere Meßwerte können den Veröffentlichungen über die *m-threads* Bibliothek entnommen werden [Bellos95, Stecker95, BelSte96 und Bellos98].

5.4.2 Affinitätsmodelle

Die *m-threads* Bibliothek bietet fünf Modelle zur Bestimmung der Cacheaffinität eines Threads an. Eine genauere Beschreibung der Modelle findet sich wiederum in [Stecker95].

- *noaff*:
  Bei dem Affinitätsmodell *noaff* erfolgen keine weiteren Affinitätsbetrachtungen. Die Threads werden in der Reihenfolge ihres Einhängens in die Runqueue ausgeführt (FCFS). Eine Prozessoraffinität besteht jedoch auch in diesem Fall, da eine Bindung über die lokalen Runqueues weiterhin besteht.
- *vtime*:
  Die Cacheaffinität wird, ausgehend von der Annahme, daß der zuletzt ausgeführte Thread den größten Cacheinhalt hat, indirekt bestimmt. Es ist daher keine Hardwareunterstützung in Form von Cachemiss-Zählern notwendig. Die Grundlage bildet eine virtuelle Zeit (*vtime*), die bei jedem Threadwechsel hochgezählt wird.
- *cmisses*:
  Bei diesem Verfahren wird die Anzahl der Cachemisses beim letzten Lauf eines Threads als Maß für den Cachezustand verwendet. Das Verfahren geht davon aus, daß ein Thread mit vielen Cachemisses einen Großteil seiner Arbeitsmenge in den Cache geladen hat und folglich bevorzugt zu behandeln ist. Die Anzahl der Cachemisses kann also direkt als Priorität genutzt werden.
- *cmsum*:
  Statt nur die Cachemisses des letzten Berechnungsabschnittes zu werten (*cmisses*), wird die Summe aller Cachemisses des Threads als Priorität verwendet.
- *reload*:
  In diesem Modell wird ein Erwartungswert für die Anzahl der Cachemisses bei einer erneuten Auswahl des Threads berechnet. In die Berechnung gehen nicht nur die durch den Thread verursachten Cachemisses, sondern auch die Cachemisses anderer Threads ein,

11. **F**irst **C**ome **F**irst **S**erved

die seitdem ausgeführt wurden, da jeder Cachemiss zu einer Verdrängung eines Eintrags des betrachteten Threads führen kann.

Die Messungen, wiederum auf Basis der Jacobi-Iteration und anderer Algorithmen auf Matrizen, zeigen gegenüber der FCFS-Strategie Beschleunigungfaktoren zwischen 80 und 160. Die Ergebnisse zeigen also deutliche Einbrüche bis hin zu sehr deutlichen Verbesserungen, wobei erneut festzustellen ist, daß sich Maßnahmen zur Verbesserung der Cachenutzung stärker auswirken, wenn mehrere Nodes der NUMA-Architektur genutzt werden. Ungeeignet ist das *cmisses*-Modell. Es zeigt gegenüber FCFS durchgehend Einbrüche oder ist nur geringfügig besser. Werden die Cachemisses summiert (*cmsum*), ergibt sich dagegen eine sehr gute Approximation des Cachezustands und damit bis auf Ausnahmen eine deutliche Steigerung gegenüber FCFS. Das komplexere *reload*-Modell zeigt aufgrund des hohen Berechnungsaufwands erst ab mehr als einem Node positive Effekte und kommt auch dann in den meisten Fällen nicht an *cmsum* heran. Interessant ist das gute Abschneiden des einfachen *vtime*-Modells. Auf nur einem Node zeigt es die deutlich besten Ergebnisse und wird erst bei größeren Prozessorzahlen von den Modellen, die Cachemiss-Zähler ausnutzen, überholt[12]. Dies zeigt, daß *vtime* in Relation zum Berechnungsaufwand eine sehr gute Approximation des Cachezustands ermöglicht. Detailliertere Messungen werden wiederum von Bellosa und Steckermeier angegeben [Bellos95, Stecker95, BelSte96 und Bellos98].

5.5 Zusammenfassung

Mit den in den Teilprojekten behandelten Schwerpunkten umfaßt das ELiTE-Projekt einen Großteil der relevanten Bereiche eines modernen Thread-Schedulings auf speichergekoppelten Multiprozessorsystemen. Diese Schwerpunkte sind:

- Extrem leichtgewichtige Prozesse (Threads) auf Benutzerebene
- Anpassung der Softwarearchitektur an die Systemarchitektur
- Integration von Schedulingmechanismen auf Benutzer- und Systemebene
- Effiziente Unterbrechungsmechanismen auf Benutzerebene
- Scheduling unter Berücksichtigung von Cache- und Speichernutzung auf Benutzer- und Systemebene

Damit ist eine Grundlage geschaffen, die es erlaubt, nicht nur neue Mechanismen in einzelnen Bereichen der Problemfelder Scheduling und Speicherverwaltung zu untersuchen, sondern auch deren Zusammenspiel und Abhängigkeiten zu betrachten. Die Integration verschiedener Mechanismen in zwei architekturell unterschiedliche kommerzielle UNIX-Systeme zeigt, daß moderne Schedulingverfahren nicht vor heutigen kommerziellen Betriebssystemen halt machen müssen. Die einzelnen Projekte verdeutlichen, daß ein effizienter Umgang mit der Ressource Cache im Scheduling sowohl im User-Level als auch im Betriebssystemkern nötig und möglich

12. Dies ist insbesondere für die Messungen des *Sleeping-Threads* Mechanismus interessant, die auf einem Singlenode-System ausgeführt wurden (siehe Abschnitt 6.3.3).

ist und daß Scheduling insbesondere auf modernen Rechnerarchitekturen bei weitem nicht mehr nur eine Frage der Verteilung von Prozessorkapazitäten ist.

In den folgenden zwei Kapiteln werden konkrete Mechanismen vorgestellt, mit denen die Probleme des User-Level-Scheduling über eine bessere Integration mit der übrigen Systemsoftware gelöst werden können.

6 Sleeping-Threads

Ein Teilbereich des ELiTE-Projekts befaßt sich mit einer besseren Integration des User-Level-Schedulings in die übrige Systemsoftware ein. Die *Sleeping-Threads,* eine Erweiterung des Threadschedulings im Betriebssystemkern, bilden eine gegenüber herkömmlichen Kernel-Threads modifizierte Abstraktion der vom Betriebssystemkern angebotenen Mechanismen für eine anwendungsinterne Parallelität. Auf dieser Basis kann eine im User-Level implementierte Threadbibliothek ein Threadverhalten realisieren, das dem von Kernel-Threads sehr nahe kommt und die entscheidenden Probleme des User-Level-Schedulings vermeidet.:

- Blockierungen im Kern können erkannt und behandelt werden.
- Die Anzahl der lauffähigen virtuellen Prozessoren kann konstant gehalten werden und entspricht der Anzahl der verfügbaren physikalischen Prozessoren.
- Eine Synchronisation der Threadbibliothek mit einem CPU-Scheduling im Kern (dynamische Partitionierung) ist in einfacher Weise realisierbar.
- Ein Two-Level Scheduling kann weitgehend vermieden werden.

Über den *Sleeping-Threads* Mechanismus ist ein effizientes und gut in das Gesamtsystem integriertes Scheduling möglich. Basierend auf dieser Kombination sind auch neue, die parallele Ausführung nutzende Ansätze zur Lösung anderer systemnaher Probleme möglich. Als Beispiel wird in diesem Kapitel auf ein Problem aus dem Bereich der Speicherverwaltung eingegangen. Ein effizientes und systemintegriertes Threadsystem kann genutzt werden, um bei *out-of-core*-Anwendungen ein Interleaving von Paging und Berechnung und somit ein Verbergen der Zugriffslatenz (Latency-Hiding) zu erreichen. Auf Ebene des User-Level-Schedulings werden damit Verfahren möglich, die im Kernel-Scheduling seit langem üblich sind.

Ein Ziel beim Entwurf der *Sleeping-Threads* bestand darin, über möglichst geringe Erweiterungen im Betriebssystemkern die Voraussetzung für eine Lösung der Integrationsprobleme des User-Level-Schedulings zu schaffen. Der entwickelte Ansatz gleicht in Grundzügen dem Ansatz von Anderson et. al. [ABLL92], unterscheidet sich aber in wesentlichen Punkten bezüglich Effizienz, Eignung für NUMA-Architekturen und Zielsetzung von den *Scheduler Activations*.

Die *Sleeping-Threads* kombinieren die Vorteile verschiedener in Abschnitt 4.5 vorgestellter Ansätze und ermöglichen als Teil einer umfassenden Scheduling-Plattform eine Untersuchung in Zusammenhang mit weiteren Mechanismen.

Nach einer Beschreibung des *Sleeping-Threads* Mechanismus und dessen Implementierung werden sowohl allgemeine Fragestellungen bezüglich einer auf den *Sleeping-Threads* basierenden Intergration des User-Level-Schedulings als auch die konkrete Umsetzung der Anpassung der *m-threads* Bibliothek erörtert. Abschließend werden verschiedene Anwendungen eines integrierten User-Level-Schedulings vorgestellt und anhand der gemessenen Ergebnisse bewertet.

6.1 *Sleeping-Threads* - Eine Erweiterung des Kernel-Schedulers

Eine Erweiterung des Kernel-Schedulers muß in drei Bereichen ansetzen, um die oben genannten Ziele zu erreichen. Zum ersten ist dies die Kommunikation zwischen Kern und Threadbibliothek, zum zweiten die Bereitstellung von Ersatz für blockierte virtuelle Prozessoren und zum dritten die Verlagerung von Schedulingentscheidungen bezüglich Kernel-Threads aus dem Kern in die Threadbibliothek im User-Level.

Objekt der Kommunikation ist der Zustand der Kernel-Threads, der im Kern sowohl passiv, durch Setzen eines neuen Zustands, der von der Threadbibliothek aus gelesen werden kann, als auch aktiv, durch eine explizite Benachrichtigung in den User-Level, propagiert wird. Die Art des Kommunikationsmechanismus ist letztendlich ein Implementierungsdetail. Entscheidend ist, daß die Threadbibliothek den Zustand ihrer Threads nicht nur im User-Level, sondern auch im Kernel-Level verfolgen und damit eine optimale Verwaltung gewährleisten kann. Um dies zu realisieren, muß sie Zugriff auf den im Betriebssystemkern verwalteten Kernel-Thread-Zustand ihrer virtuellen Prozessoren haben.

Über den Zugriff auf Zustandsinformationen der als virtuelle Prozessoren eingesetzten Kernel-Threads kann eine Threadbibliothek die Blockierung eines ihrer virtuellen Prozessoren feststellen. Um in effizienter Weise für Ersatz zu sorgen und damit den Einbruch in der Parallelität einer Anwendung zu vermeiden, reicht dies jedoch nicht aus. Das Erzeugen eines neuen Kernel-Threads durch die Bibliothek wäre viel zu ineffizient und könnte erst erfolgen, wenn eine Blockierung erkannt wird. In diesem Fall muß folglich eine aktive Propagierung des neuen Zustands gleichzeitig mit einer effizienten Ersatzstellung durch den Betriebssystemkern erfolgen.

Wenn ein im Betriebssystemkern blockierter virtueller Prozessor wieder lauffähig (deblockiert) ist, entscheidet in einem nicht modifizierten System der Kernel-Scheduler, wann seine Ausführung fortgesetzt wird. Im Rahmen des hier vorgestellten Schedulingkonzepts kann dies aus zwei Gründen nicht dem Kernel-Scheduler überlassen bleiben. Der virtuelle Prozessor unterliegt zwar als Kernel-Thread dem Kernel-Scheduler, läuft aber im Kontext eines User-Level-Threads der Threadbibliothek. Die Threadbibliothek ist die Instanz, in der eine sinnvolle Entscheidung, wann ein Thread fortgesetzt und welcher Thread der gleichen Anwendung statt dessen verdrängt wird, am ehesten möglich ist. Dem Kernel-Scheduler sollte nur das Scheduling zwischen den Threads verschiedener Anwendungen überlassen bleiben. Der zweite Grund liegt in dem als

Ersatz für den blockierten virtuellen Prozessor bereitgestellten Kernel-Thread. Verbleibt der deblockierte Kernel-Thread unter der Kontrolle des Kernel-Schedulers, muß dieser einen Thread mehr verwalten, als die Threadbibliothek virtuelle Prozessoren nutzt. Dies würde zu verschiedenen Problemen führen, die, innerhalb des Kernel-Schedulers zu lösen, dem Ansatz einer möglichst kleinen Erweiterung des Betriebssystemkerns widerspricht. Die Kontrolle über den deblockierten Kernel-Thread muß folglich aus dem Betriebssystemkern in die Threadbibliothek verlagert werden.

6.1.1 Der Basismechanismus

Im folgenden wird mit dem Basismechanismus der *Sleeping-Threads* eine Erweiterung des Kernel-Schedulers vorgestellt, die den im vorausgehenden Abschnitt genannten Anforderungen entspricht. Die Konzeption des Basismechanismus der *Sleeping-Threads* wurde bereits im Rahmen der *7th IASTED/ISMM International Conference on Parallel and Distributed Computing and Systems (PDCS'95)* veröffentlicht [Koppe95].

Zur Erläuterung des *Sleeping-Threads* Mechanismus wird auf die Darstellung des Effekts des Parallelitätseinbruches bei Blockierungen im Betriebssystemkern (Abbildung 4.2) zurückgegriffen. Zusätzlich werden weitere Threadzustände für User-Level-Threads (**BK** und **DK**) und Kernel-Threads (**SS**, **BS** und **DS**) eingeführt:

- **BK** Im Kern blockierter User-Level-Thread
- **DK** Im Kern deblockierter (wieder lauffähiger) User-Level-Thread
- **SS** Geparkter (schlafender) Kernel-Thread (*Sleeping-Thread*)
- **BS** Blockierter Kernel-Thread (*Sleeping-Thread*); ersetzt den Zustand **B** (blockiert)
- **DS** Deblockierter Kernel-Thread (*Sleeping-Thread*)

Der Zustand **R** für ablauffähige User-Level-Threads wird mit unveränderter Bedeutung beibehalten, ist aber nicht mehr der einzig mögliche Zustand. Andere Threadzustände eines User-Level-Schedulers (z.B. Blockierungen im Kontext der Bibliothek) sind weiterhin nicht von Interesse. Die Zahlen in den Kreisen, die die Kernel-Threads darstellen, entsprechen der Identifikation des virtuellen Prozessors, der durch den Kernel-Thread repräsentiert wird. Sie können auch als Identifikation des physikalischen Prozessors angesehen werden, wenn sichergestellt ist, daß keine zusätzlichen Threadumschaltungen unabhängig vom *Sleeping-Threads* Mechanismus auftreten.

In dem als Beispiel gewählten Szenario verwaltet eine Threadbibliothek neun User-Level-Threads und führt sie nach einer nicht näher spezifizierten Strategie auf drei virtuellen Prozessoren (Kernel-Threads) aus. Entsprechend den in Abschnitt 4.4 entwickelten Anforderungen an ein effizientes User-Level-Scheduling entspricht die Anzahl der virtuellen Prozessoren der Anzahl der verfügbaren physikalischen Prozessoren[1]. In der Ausgangssituation (Abbildung 6.1) sind die ersten drei User-Level-Threads an die virtuellen Prozessoren gebunden. Zusätzlich erzeugt die Threadbibliothek weitere Kernel-Threads, die als Reserve für blockierende virtuelle Prozessoren im Kern geparkt werden[2]. Alle Kernel-Threads, die als virtuelle Prozessoren akti-

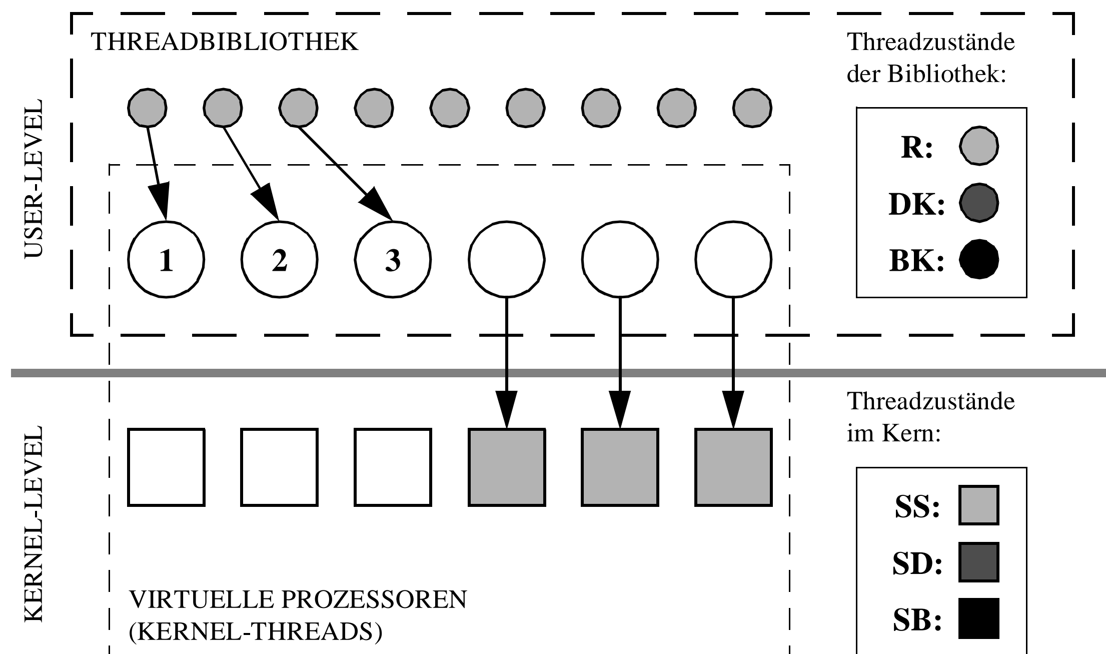

Abb. 6.1: Geparkte *Sleeping-Threads* in Reserve

ven und auch die geparkten, sind mit dem *Sleeping-Threads* Attribut gekennzeichnet, um eine spezielle Behandlung im Scheduling des Kerns zu ermöglichen.

Blockiert der virtuelle Prozessor Nummer 3 beispielsweise durch einen Systemaufruf oder einen Pagefault des in seinem Kontext laufenden User-Level-Threads, wird einer der Reservethreads deblockiert. Der Reservethread kehrt aus dem Aufruf, über den er geparkt wurde, in den User-Level zurück und kann von der Threadbibliothek als Ersatz für den blockierten Kernel-Thread genutzt werden. In diesem Beispiel wählt der Scheduler den sechsten User-Level-Thread zur Ausführung aus. Dieser Mechanismus ist sehr effizient, da lediglich ein Threadwechsel zu dem Reservethread notwendig ist, der im Falle einer Blockierung im Kern in jedem Fall erfolgen müßte. Im Unterschied zu den *Scheduler Activations*, bei denen ein neuer Thread erzeugt, zumindest aber ein Stackframe für einen wiederverwendeten Thread aufgesetzt werden muß, ist hier der Aufwand auf ein Minimum reduziert. Aus Sicht der Anwendung hat sich am Zustand des Systems nichts geändert. Trotz der Blockierung des Kernel-Threads stehen weiterhin drei virtuelle Prozessoren zur Abarbeitung der ablauffähigen User-Level-Threads zur Verfügung. Die virtuellen Prozessoren verhalten sich so, wie dies auch von physikalischen Prozessoren unter der Verwaltung des Betriebssystemkerns erwartet werden würde.

Um die in Abbildung 6.2 dargestellte Situation zu erreichen, muß die Threadbibliothek über den neuen Zustand des mit dem blockierten Kernel-Thread assoziierten User-Level-Threads infor-

1. In der aktuellen Implementierung wird dies dadurch gewährleistet, daß eine *Sleeping-Threads* Anwendung exklusiv in einer statischen Partition ausgeführt wird. Der *Sleeping-Threads* Mechanismus kann jedoch auch genutzt werden, um in einfacher Weise den User-Level-Scheduler mit der aktuellen Prozessoranzahl einer dynamischen Partitionierung zu synchronisieren. Dies wird in Abschnitt 6.2.3 erläutert.
2. Es wird zunächst vereinfachend angenommen, daß immer genügend Reservethreads verfügbar sind. Auf den gegenteiligen Fall und Maßnahmen, die eine genügende Anzahl von Reservethreads gewährleisten, wird in späteren Abschnitten eingegangen.

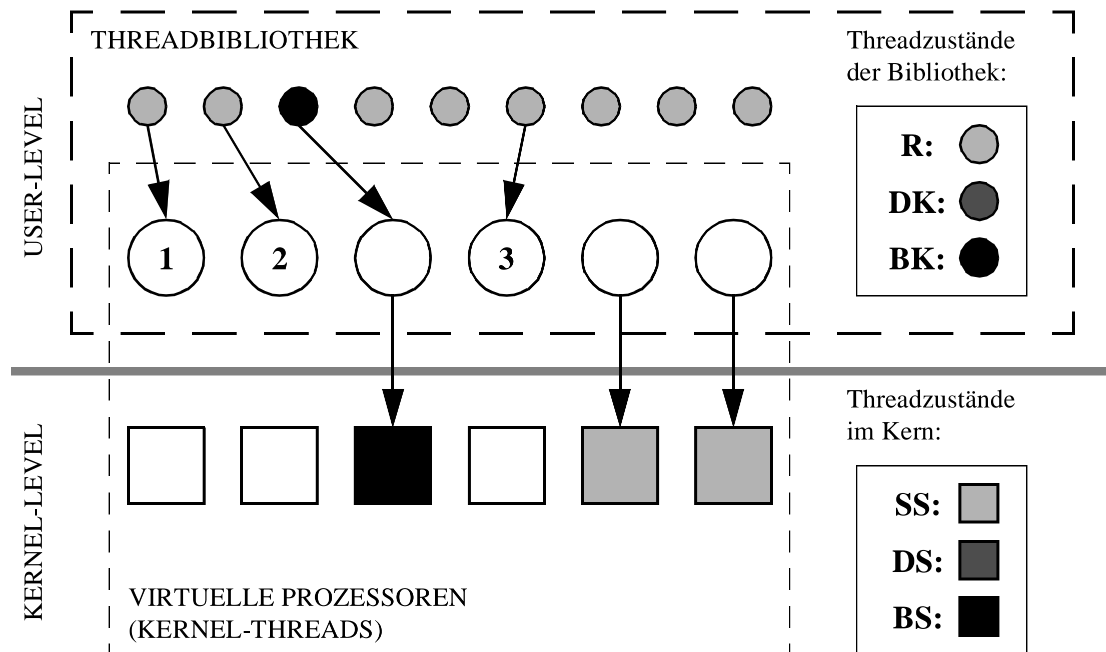

Abb. 6.2: Blockierter Kernel-Thread ersetzt durch geparkten Thread

miert werden. Erst dann kann sie dem User-Level-Thread den Zustand **BK** zuordnen. Außerdem muß die Identifikation des blockierten Kernel-Threads bekannt sein, damit der Reservethread im Kontext des gleichen virtuellen Prozessors weiterlaufen kann. Dies ist notwendig, um ein Affinitätsscheduling in der Threadbibliothek zu ermöglichen. Die direkte Threadumschaltung im Kern zwischen dem blockierten Thread und dem Reservethread stellt sicher, daß der gleiche physikalische Prozessor verwendet wird.

Für eine effiziente Realisierung dieses Informationsflusses zwischen Kern und Threadbibliothek sind zwei Varianten denkbar:

- Die Identifikation des blockierten Threads kann als Ergebniswert des terminierenden Systemaufrufs des Reservethreads übergeben werden. Diese Information reicht aus, um den virtuellen Prozessor und damit auch den User-Level-Thread zu identifizieren.
- Über den Einsatz gemeinsamer Speicherbereiche zwischen Kern und Threadbibliothek kann das gleiche erreicht werden. Diese Lösung ist flexibler und kann zum Austausch weiterer Informationen genutzt werden. Wenn ein schreibender Zugriff auch durch die Threadbibliothek zugelassen wird, müssen jedoch die entsprechenden Routinen im Kern robust genug sein, um nicht durch eventuell falsche oder inkonsistente Daten in diesem Speicherbereich die Integrität des Kerns zu gefährden.

Zur Übergabe der Identifikation des blockierten Threads eignen sich beide Varianten. Auch bezüglich der Effizienz sind keine großen Unterschiede zu erwarten. Eine Übergabe über den Ergebniswert wäre mit erheblich geringerem Aufwand zu realisieren, scheidet jedoch als einziger Kommunikationsmechanismus aus, da die Verwendung eines gemeinsamen Speicherbereichs in anderen Situationen notwendig ist. Daher wurde auch hier die zweite Variante bevorzugt.

Abbildung 6.3 zeigt den Zustand des Systems, nachdem der blockierte Kernel-Thread wieder ablauffähig ist. Der Kern darf den Thread zu diesem Zeitpunkt nicht in einen entsprechenden Zustand setzen und in die Runqueue einhängen, da dann mit vier Kernel-Threads einer zuviel lauffähig wäre. Dies würde unvermeidlich zu einem Two-Level-Scheduling führen. Stattdessen werden Kernel- und assoziierter User-Level-Thread in den Zustand **DS** bzw. **DK** gesetzt und die Entscheidung über eine erneute Ausführung des Threads der Threadbibliothek überlassen.

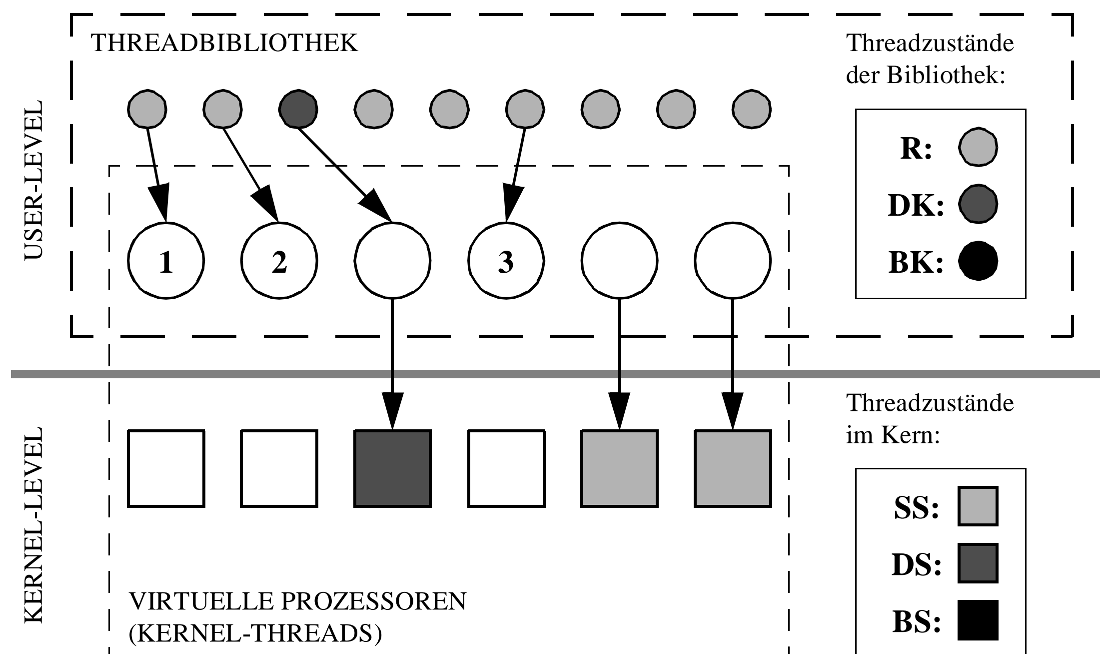

Abb. 6.3: Deblockierter Kernel-Thread

Die Propagierung des neuen Threadzustandes in den User-Level ist in diesem Fall nur über einen gemeinsamen Speicherbereich möglich, da nicht wie im vorausgehenden Fall ein terminierender Systemaufruf genutzt werden kann. Durch den gemeinsamen Speicher kann auch der bei den *Scheduler Activations* auftretende Effekt, daß ein weiterer Thread unterbrochen werden muß, um die Deblockierung des ersten zu propagieren, vermieden werden. Ein weiterer Unterschied zu den *Scheduler Activations* liegt in der Behandlung des deblockierten Threads, der nicht bis zum Verlassen des Kerns fortgesetzt und erst dann wieder als ablauffähig der Kontrolle der Threadbibliothek übergeben wird, sondern sofort der Kontrolle der Threadbibliothek unterliegt. Diese Kontrolle des User-Levels über Threads im Kontext des Kerns führt zu Problemen, die in Abschnitt 6.1.2 diskutiert werden.

Die Threadbibliothek verwaltet in dieser Situation einen User-Level-Thread, der jedoch ablauffähig an einen Kernel-Thread gebunden ist. Soll einer der virtuellen Prozessoren auf diesen User-Level-Thread umschalten, ist daher ein normaler Threadwechsel im User-Level nicht möglich. Der virtuelle Prozessor muß zunächst den Kontext des laufenden User-Level-Threads sichern, um eine spätere Fortsetzung zu ermöglichen, und dann einen Systemaufruf ausführen, der den deblockierten Kernel-Thread und damit ebenfalls den User-Level-Thread fortsetzt. Die Fortsetzung erfolgt durch einen direkten Threadwechsel zwischen beiden Kernel-Threads, wobei der aufrufende im Kern als Reservethread geparkt wird. Abbildung 6.4 zeigt den Zustand

nach einer solchen Umschaltung. Der zweite User-Level-Thread wird nicht mehr ausgeführt, der zweite Kernel-Thread ist geparkt und hat die Kontrolle über den virtuellen Prozessor Nummer 2 an den dritten Kernel-Thread übergeben. Mit diesem Schritt ist die Ausgangssituation wieder erreicht. Drei Kernel-Threads stehen als virtuelle Prozessoren zur Verfügung und drei weitere sind als Reservethreads im Kern geparkt.

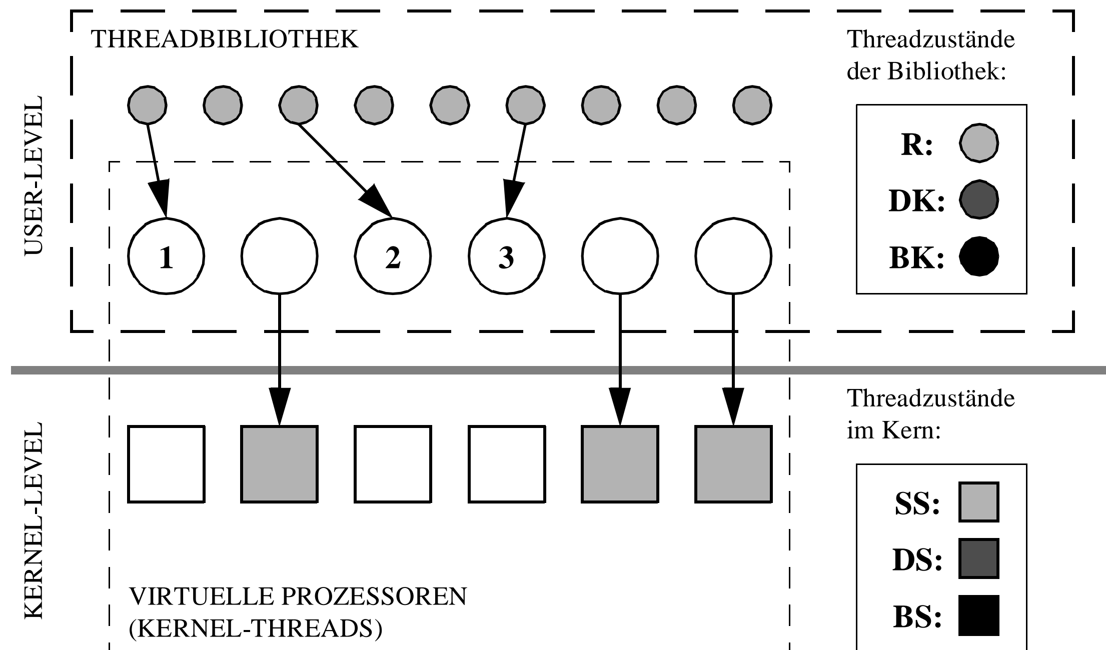

Abb. 6.4: Deblockierter Kernel-Thread fortgesetzt aus dem User-Level

Die letzte Schedulingentscheidung in diesem Beispiel führt jedoch dazu, daß der dritte User-Level-Thread auf einem anderen virtuellen und ebenfalls physikalischen Prozessor fortgesetzt wird und eventuell noch vorhandene Cacheinhalte nicht genutzt werden können. Eine Threadbibliothek mit Affinitätsscheduling hätte in diesem Fall einen anderen User-Level-Thread zur Ausführung auf dem virtuellen Prozessor Nummer 2 ausgewählt. Der dritte User-Level-Thread würde im Rahmen eines Affinitätsschedulings sinnvollerweise durch den virtuellen Prozessor Nummer 3 fortgesetzt. Die Auswahl im Beispiel soll hauptsächlich zeigen, daß die Fortsetzung eines an einen Kernel-Thread gebundenen User-Level-Threads durch jeden virtuellen Prozessor erfolgen kann.

Der *Sleeping-Threads* Mechanismus ermöglicht es folglich einer Threadbibliothek, mit einer - auch im Fall von Blockierungen - konstanten Anzahl von verfügbaren virtuellen Prozessoren zu arbeiten. Sie ist, unter der Voraussetzung, daß eine der Anzahl der virtuellen Prozessoren entsprechende Menge von physikalischen Prozessoren exklusiv verfügbar ist, die einzige Instanz, die das Scheduling der Threads einer Anwendung untereinander regelt. Ein Two-Level-Scheduling mit seinen unter Umständen fatalen Folgen für die Wirksamkeit der optimierten Schedulingstrategien im User-Level kann unter dieser Voraussetzung ausgeschlossen werden.

Ein bedeutender Vorteil des gewählten Ansatzes ist, daß die Strategie des Kernel-Schedulers nicht unmittelbar beeinflußt oder modifiziert wird, sondern orthogonal zu dieser lediglich direk-

te Umschaltungen zu Threads erfolgen, die im Rahmen der Kernel-Strategie nicht auswählbar sind. Dadurch ist der *Sleeping-Threads* Mechanismus mit einer großen Zahl denkbarer Schedulingstrategien des Betriebssystemkerns kombinierbar. Nur um eine Synchronisation der in der Threadbibliothek genutzten virtuellen Prozessoren mit der Anzahl der zu einem Zeitpunkt exklusiv nutzbaren physikalischen Prozessoren zu erreichen, muß eine engere Koordination zwischen der Strategie des Kernel-Schedulers und der über den *Sleeping-Threads* Mechanismus induzierten Schedulingentscheidungen der Threadbibliothek realisiert werden. Die Orthogonalität zur Schedulingstrategie des Kernels ermöglicht darüber hinaus eine im Prinzip einfache Implementierung des *Sleeping-Threads* Mechanismus.

Die These der einfachen Implementierbarkeit geht von einer idealen Situation aus, die in der Realität nicht gegeben ist. Diesen Idealfall stellt ein monolithisches Betriebssystem mit einem einfach und übersichtlich strukturierten Threadscheduler dar. Auch muß ein Betriebssystem für Mechanismen wie die *Sleeping-Threads* geeignet sein und darf nicht über eine Kontrolle bestimmter Ressourcen durch den User-Level verletzbar gemacht werden. Jede Abweichung von diesem Idealfall kann zu Problemen im Sinne einer komplexeren Realisierung oder einer Einschränkung der Funktionalität bzw. der angestrebten Zielsetzung des neuen Mechanismus führen. Im Extremfall sind Verklemmungen des gesamten Systems möglich. Auf einige dieser Probleme wird im folgenden näher eingegangen, bevor die Implementierung in Grundzügen beschrieben wird.

6.1.2 User-Level Kontrolle über Ressourcen des Betriebssystemkerns

Die Blockierung eines Threads im Betriebssystemkern kann weitreichende Folgen für das gesamte System und nicht nur für den Fortschritt der betroffenen Anwendung haben, wenn ein solcher Thread wichtige Systemressourcen, wie zum Beispiel Locks, hält. Ein wesentlicher Punkt beim Design von Betriebssystemen liegt folglich darin, eine Blockierung solcher Threads zu vermeiden oder sicherzustellen, daß wichtige Systemressourcen vor einer Blockierung freigegeben werden bzw. nach einer Blockierung von anderen Threads übernommen werden können. Unumgänglich ist dies bei Blockierungen, die potentiell lang oder sogar unendlich andauern können. In vielen Fällen wird jedoch angenommen, daß eine kurzfristige Blockierung eines solchen Threads und damit der gehaltenen Ressourcen tolerierbar ist. Bei Mechanismen wie den *Sleeping-Threads* ist jedoch keine Aussage über die Dauer einer Blockierung mehr möglich, da die Entscheidung über die Fortsetzung eines deblockierten Kernel-Threads in den User-Level verlagert wird. Eine Anwendung kontrolliert also nicht nur die Fortsetzung eines ihrer eigenen Threads, sondern auch die Freigabe von Systemressourcen, die von dem betroffenen Thread gehalten werden. Mögliche Folgen reichen von einer vorübergehenden Blockierung einzelner Systemdienste bis hin zu Verklemmungen einzelner Anwendungen oder des gesamten Systems.

Zu Problemen kann es nicht nur aufgrund fehlerhafter oder bösartiger Anwendungen kommen, die einen Thread nicht fortsetzen. Ein belegtes Lock kann auch der Grund dafür sein, daß eine Threadbibliothek die notwendigen Aktionen zur Fortsetzung des Threads nicht ausführen kann und mit Auswirkungen auf das Gesamtsystem verklemmt. Von Interesse in diesem Zusammenhang sind nur Systemressourcen, also beispielsweise Locks im Betriebssystemkern, nicht aber

Locks einer Anwendung oder Threadbibliothek. Bezüglich letzterer kann ein fehlerhafter Umgang nur Auswirkungen auf die Anwendung selbst haben. Eine Kontrolle über Systemressourcen, die einer Anwendung normalerweise vorenthalten ist, kann jedoch im allgemeinen nicht zugelassen werden.

Eine konzeptionell saubere Lösung bestünde darin, ein den Anforderungen entsprechendes Betriebssystemdesign zu verwirklichen und als Grundlage für die Realisierung einer User-Level-Kontrolle über das Threadscheduling des Betriebssystemkerns einzusetzen. Ebenfalls denkbar wäre, ein bestehendes Betriebssystem durch tiefe Eingriffe in die Systemstruktur entsprechend anzupassen. Beide Varianten sind im Rahmen von ELiTE nicht realisierbar. Zwar gibt es Ansätze für Betriebssystemkerne, die beispielsweise auf Locks verzichten[3]. Diese kommen aber als Basis für das ELiTE-Projekt, das großen Wert auf eine Integration in kommerzielle Standardsysteme legt, nicht in Betracht. Auch eine Anpassung von UNIX-basierten Betriebssystemen ist aufgrund ihrer Komplexität nicht möglich. Dies würde ein vollständiges Redesign großer Teile des Systems erfordern. Ein gangbarer Weg kann daher nur in einer Einschränkung der zu entwickelnden Schedulingmechanismen liegen.

Eine sehr weitreichende Einschränkung ist in dem Threadkonzept des Betriebssystems XERO zu finden. Auf eine Kontrolle durch den User-Level wird vollständig verzichtet. Ein deblockierter Thread wird ungeachtet eines als Ersatz laufenden Threads fortgesetzt, was zu einem Two-Level-Scheduling führt, bis die Threadbibliothek die Situation erkennt und einen virtuellen Prozessor terminiert. Bei den *Scheduler Activations* wird dieses Problem gelöst, indem ein deblokkierter Thread zunächst ausgeführt wird bis er den Systemkern wieder verläßt. Dies führt zu keinem zusätzlichen Problem hinsichtlich eines Two-Level-Schedulings, da zur Propagierung der Deblockierung in jedem Fall ein anderer Thread der Anwendung unterbrochen werden muß, und damit ein Prozessor verfügbar ist, der vor dem Upcall in die Threadbibliothek für die zeitweilige Fortsetzung genutzt werden kann. Eine zweite Blockierung innerhalb eines Systemaufrufs wird nicht behandelt.

Die Lösung des Problems innerhalb des *Sleeping-Threads* Mechanismus basiert ebenfalls auf einer zeitweiligen Fortsetzung des deblockierten Threads, die jedoch über die Festlegung kritischer Abschnitte sehr viel flexibler erfolgt und nicht grundsätzlich bis zum Verlassen des Betriebssystemkerns andauert. Ein kritischer Abschnitt ist ein Abschnitt im Code des Betriebssystemkerns, in dem im Falle einer Blockierung die Fortsetzung des Threads nicht in den User-Level delegiert werden darf. Im Fall der Deblockierung eines in einem kritischen Abschnitt blockierten Threads wird dieser als lauffähig markiert und damit im Rahmen des Kernel-Schedulings ausgeführt. Erst wenn der Thread den kritischen Abschnitt verläßt, wird er erneut gestoppt, als deblockiert gekennzeichnet und eine weitere Fortsetzung der Threadbibliothek überlassen. Läuft ein Thread innerhalb eines kritischen Abschnitts während einer zeitweiligen Fortsetzung auf eine zweite Blockierung, wird kein weiterer Reservethread aktiviert, da nach der ersten Blockierung bereits einer freigegeben wurde. Nach dem Verlassen des kritischen Abschnitts unterliegt der Thread erneut der Kontrolle der Threadbibliothek, wodurch die Aus-

3. Ein Beispiel für einen Multiprozessorbetriebssystemkern, der auf Locks verzichtet, wurde von Massalin und Pu vorgestellt [MasPu91]. Ob dieser Kern eine problemlosere Integration von Mechanismen wie den *Sleeping-Threads* zulassen würde, wurde im Rahmen der vorliegenden Arbeit nicht untersucht.

nahmesituation eines zusätzlich lauffähigen Threads beendet ist. Im Falle einer weiteren Blokkierung innerhalb des selben Systemaufrufs kann daher erneut ein Reservethread freigegeben werden.

Diese Methode ist sehr viel flexibler als die Lösung der *Scheduler Activations* und bietet vor allem Vorteile im Zusammenhang mit MACH, wie im folgenden Abschnitt noch erläutert werden wird. Sie führt jedoch zu zwei neuen Problemen:

- Ein prinzipielles Problem liegt in dem Kernel-Scheduling, das durch die zusätzlichen ablauffähigen Threads induziert wird.
- Ein eher praktisches Problem stellt die Identifizierung und Festlegung der kritischen Abschnitte dar.

Ein Kernel-Scheduling im Fall einer zeitweiligen Fortsetzung läßt sich aufgrund der Konzeption der *Sleeping-Threads* als ein orthogonal zum Kernel-Scheduler angelegter Mechanismus nicht vermeiden. Da im Fall einer Deblockierung nicht wie bei den *Scheduler Activations* ein Prozessor verfügbar ist, der den Thread fortsetzen könnte, bleibt nur die Möglichkeit, ihn als ablauffähig zu markieren und die Fortsetzung bis zum Ende des kritischen Abschnitts dem Kernel-Scheduler zu überlassen. Auch wenn angenommen wird, daß in den meisten Fällen die zusätzlichen Threads nur sehr kurz laufen und keine großen Speicherbereiche referenzieren, kann das durch sie induzierte Kernel-Scheduling weitreichende Folgen haben. Es werden nicht nur von der Threadbibliothek als virtuelle Prozessoren genutzte Kernel-Threads und damit User-Level-Threads der Anwendung kurz unterbrochen und in ihrem Cacheverhalten beeinflußt, sondern in Abhängigkeit der Strategie des Kernel-Schedulers können weitere Threadumschaltungen induziert werden. Statt eines kurzzeitigen Umschaltens auf einen der fortzusetzenden deblockierten Threads kann ein Prozessorwechsel mehrerer virtueller Prozessoren die Folge sein. Mit einer in vielen Betriebssystemen relativ einfach zu realisierenden Methode[4], die aber noch nicht in den *Sleeping-Threads* Mechanismus integriert ist, können die Folgen jedoch stark eingegrenzt werden. Bindet man einen *Sleeping-Thread*, der im Idealfall ohnehin nur Schedulingentscheidungen der Threadbibliothek unterliegen sollte, fest an seinen Prozessor, wobei er dieses Attribut auch im Fall einer späteren zeitweiligen Fortsetzung behält, kann der Kernel-Scheduler alle ablauffähigen Threads ausführen, ist jedoch stark eingeschränkt. Die Umschaltung zwischen dem aktuell als virtuellen Prozessor genutzten Kernel-Thread und ein oder mehreren Threads, die im Kontext dieses virtuellen Prozessors blockiert wurden und zeitweilig fortgesetzt werden müssen, erfolgt nur auf jeweils dem Prozessor, auf dem die Threads zuvor ausgeführt wurden.

In einem Programmsystem mit der Komplexität eines Betriebssystems ist es nicht einfach, die Stellen zu identifizieren, die durch kritische Abschnitte gesichert werden müssen. Erschwerend kommt hinzu, daß es sich um ein Fremdprodukt handelt, über das interne Kenntnisse nur in sehr allgemeiner Form verfügbar sind. Neben dem Einfügen offensichtlich notwendiger kritischer Abschnitte wurde zur Identifizierung weiterer Abschnitte eine Methode gewählt, die nur mit dem Status der *Sleeping-Threads* als Forschungsprojekt zu rechtfertigen ist, auch wenn anzunehmen bleibt, daß bei kommerziellen Produkten oft nicht anders vorgegangen wird. Im laufen-

4. In vielen Betriebssystemen, auch in SPP-UX, ist der Kernel-Scheduler bereits aus anderen Gründen darauf ausgelegt, einen Thread an einen Prozessor binden zu können.

den System auftretende Verklemmungen und andere Probleme werden untersucht, problematische Abschnitte auf diesem Weg identifiziert und als kritisch geklammert, um eine Kontrolle durch den User-Level in diesen Abschnitten zu verhindern. Es ist mit großer Wahrscheinlichkeit davon auszugehen, daß auf diesem Weg nicht alle Probleme erkannt werden. Die aktuelle Implementierung kann daher keineswegs als stabil bezeichnet werden. Ein anderes Vorgehen ist im Rahmen des ELiTE-Projekts jedoch nicht mit vertretbarem Aufwand zu realisieren. Es ist auch nicht das Ziel des ELiTE-Projekts, in allen Bereichen stabile Produktionssysteme zu entwickeln, sondern neue Wege im Threadscheduling vorzuschlagen und deren potentiellen Nutzen in prototypischen Implementierungen aufzuzeigen.

6.1.3 MACH / SPP-UX

In einem MACH-basierten System ist wie in allen Mikrokernsystemen nicht die gesamte Betriebssystemfunktionalität im Systemkern integriert. Große Teile sind als Serverprozesse im User-Level realisiert, die sich von “normalen” Anwendungen nicht grundsätzlich unterscheiden. Ein Systemaufruf, der in einem monolithischen System durch einen Einsprung in den Systemkern realisiert ist, entspricht in einem Mikrokern, soweit er sich nicht an den Kern selbst, sondern an einen Server richtet, dem Senden einer Nachricht an den Server. Ein Thread des Servers führt die entsprechenden Aktionen aus und sendet eine Nachricht zurück. Der Nachrichtenmechanismus selbst ist als Dienst des Mikrokerns realisiert.

Verfolgt man den Ablauf eines Systemaufrufs in einem derart aufgebauten Betriebssystem, zeigt sich ein Problem in Zusammenhang mit den *Sleeping-Threads* und ähnlichen Mechanismen. Der aufrufende Thread einer *Sleeping-Threads* Anwendung blockiert im Nachrichtenmechanismus des Mikrokerns und wartet auf die Antwort des Servers. Diese Blockierung führt zur Freigabe eines Reservethreads, obwohl ein Serverthread im Auftrag der Anwendung ausgeführt wird. Laufen Anwendungs- und Serverthreads[5] auf der gleichen Menge von Prozessoren, führt dies mit einem Kernel-Scheduling zwischen diesen Threads zu einem Effekt, der eigentlich vermieden werden soll. Die MACH-Version der *Scheduler Activations* zeigt genau dieses Problem (siehe Abschnitt 4.5.1.1).

Um bei den *Sleeping-Threads* diesen Effekt zu vermeiden, wird ein Reservethread erst dann freigegeben, wenn im Fall eines Systemaufrufs der Serverthread blockieren sollte. Um dies zu realisieren, muß nachvollzogen werden, im Auftrag welchen Anwendungsthreads der blockierte Serverthread läuft, da der Mechanismus nur aktiv werden darf, wenn der assoziierte Clientthread ein *Sleeping-Thread* ist und ein Reservethread der entsprechenden Anwendung freigegeben werden muß. Deblockiert der Serverthread, muß auch nicht er, sondern der assoziierte Clientthread entsprechend markiert werden. Wenn die Threadbibliothek die Fortsetzung eines solchen “deblockierten” Clientthreads anstößt, muß der Mikrokern stattdessen den deblockierten Serverthread fortsetzen. Um eine solche Vererbung des *Sleeping-Threads* Attributes an Serverthreads zu ermöglichen, muß eine Verknüpfung zwischen den assoziierten Threads realisiert werden, die eine effiziente Verfolgung von Aufrufbeziehungen zwischen Client- und Server-

5. In Anlehnung an den Sprachgebrauch bezüglich RPC’s werden im weiteren die Bezeichnungen Client- und Serverthread verwendet.

thread erlaubt. In SPP-UX ist dies sehr einfach möglich, da hier der Nachrichtenmechanismus von MACH durch einen effizienteren, RPC genannten Aufrufmechanismus ersetzt wurde (siehe Abschnitt 5.2.2.1), in dem eine Verkettung der Datenstrukturen solcher Threads bereits existiert. Da unter SPP-UX einzelne Nodes der NUMA-Architektur bezüglich des Threadschedulings getrennt behandelt werden (Threads können nicht über Nodegrenzen hinweg migrieren), ist die Vererbung nur bei Aufrufen innerhalb eines Nodes realisiert. Laufen Client- und Serverthread auf verschiedenen Nodes, wird für den Clientthread zum Zeitpunkt des Aufrufs des Servers ein Reservethread freigegeben, da auf dem Clientnode zu diesem Zeitpunkt ein Prozessor frei wird und die Ersatzstellung eines virtuellen Prozessors sinnvoll ist.

Über die Vererbung des *Sleeping-Threads* Attributs wird es möglich, diesen Mechanismus auch in einem MACH-basierten System wie SPP-UX sinnvoll einzusetzen. Das Problem der Kontrolle des User-Levels über die Fortsetzung von im Betriebssystemkern blockierten Threads wird dadurch jedoch verschärft. Dies dürfte einer der Gründe sein, warum die MACH-Version der *Scheduler Activations* Serverthreads nicht berücksichtigt. Das Konzept, deblockierte Threads grundsätzlich bis zum Verlassen des Betriebssystemkerns fortzusetzen und erst dann die Kontrolle an die Threadbibliothek zu delegieren, trägt in diesem Fall nicht mehr. Blockiert ein Serverthread im Mikrokern, während er wichtige Ressourcen des Servers hält, würde eine analoge Situation entstehen, weil eine Fortsetzung des Threads nur bis zum Verlassen des Mikrokerns garantiert werden kann. Dies wäre nicht weniger fatal, da auch der Server über global genutzte Ressourcen verfügt. Ein ähnliches Problem, das sich jedoch nur auf die verursachende Anwendung selbst auswirkt, zeigt der Ansatz der *Scheduler Activations* in Zusammenhang mit dem Emulator. Der Emulator nutzt Locks zur Erhaltung der Konsistenz seiner Datenstrukturen. Tritt eine Blockierung aufgrund eines Pagefaults auf, während ein Thread solche Locks hält, kann die Anwendung verklemmen. Die kritischen Abschnitte der *Sleeping-Threads* sind deutlich flexibler und erlauben eine Lösung dieser Probleme, wenn das Setzen eines kritischen Abschnitts auch in Server und Emulator möglich ist. Auf Realisierung und Effizienz der kritischen Abschnitte in Server und Emulator, die das Setzen von Attributen in den Datenstrukturen des Mikrokerns voraussetzen, wird in dem Abschnitt über die Implementierung näher eingegangen.

6.1.4 Implementierung

Über die Implementierung des *Sleeping-Threads* Mechanismus wird im folgenden ein kurzer Überblick gegeben, der folgende zentrale Elemente umfaßt: Anhand der Datenstrukturen wird der strukturelle Aufbau der Implementation erläutert. Die wenigen Anbindungspunkte, an denen der *Sleeping-Threads* Mechanismus in den Kernel-Scheduler eingehängt wird, verdeutlichen die einfache Integration des Kerns der Erweiterung, während die kritischen Abschnitte ein Beispiel für die realen Probleme darstellen, die zu einer komplexen Realisierung in der Praxis führen. Ein weiterer Abschnitt befaßt sich mit der Vererbung des *Sleeping-Threads* Attributs an Threads des Servers.

Ein erster Prototyp, der noch keine kritischen Abschnitte und nur eine rudimentäre Unterstützung zur Anbindung einer Threadbibliothek realisierte, wurde im Rahmen einer Diplomarbeit entwickelt und beschrieben [Hollma96]. Für eine detailliertere Beschreibung der Implementie-

rung wird auf diese Arbeit verwiesen. Die Kernpunkte werden in den folgenden Abschnitten dargestellt.

Da das aus sechs Hypernodes aufgebaute System nur im sehr eingeschränkten Maß für eine exklusive Nutzung zum Test und Debugging des modifizierten Betriebssystemkerns verfügbar war, konnte eine hinreichende Stabilität der Multinode-Version der *Sleeping-Threads* nicht erreicht werden. Messungen waren daher nur auf dem im Rahmen des ELiTE-Projekts ohne Einschränkungen verfügbaren Single-Node-System möglich.

6.1.4.1 Datenstrukturen

Die zentralen Datenstrukturen der *Sleeping-Threads* stellen die Strukturen zur Verwaltung der geparkten Reservethreads, die Erweiterungen der Threadstruktur, insbesondere der möglichen Threadzustände, und den Aufbau der gemeinsamen Speicherbereiche zur Kommunikation mit dem User-Level dar.

In SPP-UX wird ein Prozeß als eine Menge von jeweils einer MACH-Task für jeden Node, über den sich der Prozeß erstreckt, repräsentiert. Da Threads nicht über Node- bzw. Taskgrenzen hinweg migrieren können, müssen die Reservethreads einer Anwendung für jeden Node einzeln verwaltet werden. Jede Task einer *Sleeping-Threads* Anwendung verfügt demnach über eine eigene verkettete Liste zur Verwaltung der freien Reservethreads.

Die Liste der geparkten Reservethreads ist die einzige relevante Datenstruktur auf Task-Ebene, alle anderen Erweiterungen sind direkt den Threads zugeordnet. Bezüglich der eigentlichen Threadstruktur ist neben einem Indikator zur Erkennung eines *Sleeping-Threads*, einigen Erweiterungen für statistische Zwecke und einem Zeiger auf den gemeinsamen Speicherbereich des Threads insbesondere die Erweiterung der möglichen Threadzustände zu erwähnen. Die Erweiterung des Threadzustands wird nur an den Aufhängepunkten des *Sleeping-Threads* Mechanismus ausgewertet und ist derart angelegt, daß die unmodifizierten Teile des Schedulers den erweiterten Zustand als einen auch im ursprünglichen System sinnvollen Zustand interpretieren. Deshalb reicht es aus, den Kernel-Scheduler nur an wenigen Stellen zu erweitern und nicht vollständig zu überarbeiten.

Eine Darstellung der Realisierung der gemeinsamen Speicherbereiche zwischen Betriebssystemkern und User-Level würde den Rahmen dieses Überblicks sprengen. Es wird ein Mechanismus genutzt, der in SPP-UX bereits integriert ist, um einen effizienten Zugriff der Architectural Interface Library [Helgaa94] auf Systemressourcen wie zum Beispiel Cachemiss-Zähler oder Timer zu ermöglichen. Innerhalb eines Prozesses kann nur auf die Speicherbereiche der Threads des gleichen Prozesses, die auf dem selben Node wie der zugreifende Thread laufen, zugegriffen werden. Diese Einschränkung stellt für die Realisierung einer Threadbibliothek kein ernsthaftes Problem dar, da eine direkte Umschaltung auf einen deblockierten Thread eines anderen Nodes ohnehin nicht möglich ist.

Die in dem gemeinsamen Speicherbereich gehaltenen Daten sind nicht sehr umfangreich. Für einen rudimentären *Sleeping-Threads* Mechanismus reichen die folgenden Einträge aus. In der Implementation werden aus Effizienz- und Statistikgründen bzw. besonderen Anforderungen der *m-threads* Bibliothek einige zusätzliche Einträge verwendet. Auf einige der zusätzlichen

Einträge wird in Zusammenhang mit der Diskussion der Anbindung des User-Level-Schedulings eingegangen.

- `kernel_state`:
  Dient zur Propagierung des Zustands der Kernel-Threads in den User-Level, soweit dies von Interesse für eine Threadbibliothek ist.
- `my_tid`:
  Hält die Thread ID des zugehörigen Kernel-Threads. Dieser Eintrag ist nicht unbedingt nötig, da die Thread ID im User-Level bekannt ist und daher auch in der nicht von außerhalb des Kerns zugreifbaren Threadstruktur enthalten sein kann. Dieser Speicherort wurde gewählt, da die Thread ID eigentlich kein Datum des Mikrokerns, sondern des Servers ist, aber vom Mikrokern benötigt wird, um in den Eintrag `running_for_tid` übertragen zu werden.
- `running_for_tid`:
  Im Fall einer Blockierung wird der Eintrag `my_tid` des blockierten Threads in den Eintrag `running_for_tid` des freigegebenen Reservethreads übertragen, bevor die eigentliche Threadumschaltung erfolgt. Die Threadbibliothek kann mit dieser Information nachvollziehen, für welchen Kernel-Thread und damit für welchen virtuellen Prozessor der Reservethread läuft.
- `current_vcpu`:
  Dieser Eintrag ist ebenfalls nicht unbedingt notwendig, da alle nötigen Informationen aus der Thread ID abgeleitet werden können. Die Threadbibliothek kann einen beliebigen Wert eintragen, der vom Mikrokern im Fall einer Umschaltung auf einen Reservethread lediglich kopiert wird. Sinnvollerweise ist dies die Identifikation des virtuellen Prozessors, der durch den Kernel-Thread repräsentiert wird. Da der Wert vom Mikrokern nicht ausgewertet wird, kann die Threadbibliothek die Semantik festlegen und ihn ihrem Aufbau entsprechend zum Beispiel als Zeiger oder als Index in einer Tabelle verwenden. Der Eintrag ermöglicht eine effiziente und einfache Implementierung.

In Abbildung 6.5 wird die Situation beim Umschalten auf einen Reservethread graphisch dargestellt. Der blockierende Thread **A** und der Reservethread **B** enthalten in ihrer Threadstruktur einen Zeiger (`STH_state`) auf den jeweiligen gemeinsamen Speicherbereich.Vor der eigentlichen Threadumschaltung wird die Zustandsinformation in den gemeinsamen Speicherbereichen aktualisiert (von ablaufend nach blockiert für Thread **A** bzw. von geparkt nach ablaufend für Thread **B**) und Thread ID sowie Identifikation des virtuellen Prozessors in den Speicherbereich der freigegebenen Reservethreads kopiert.

6.1.4.2 Anbindung des *Sleeping-Threads* Mechanismus

Die Stellen, an denen der *Sleeping-Threads* Mechanismus in den Betriebssystemkern eingehängt werden muß, lassen sich aus der Beschreibung des Basismechanismus in Abschnitt 6.1.1 ableiten. Zum Parken von Reservethreads und zum Fortsetzen deblockierter Threads müssen neue Systemaufrufe integriert werden. Das eigentliche Einhängen in den Kernel-Scheduler erfordert Erweiterungen in drei Schedulingabläufen. Im Fall einer Blockierung muß der Kernel-

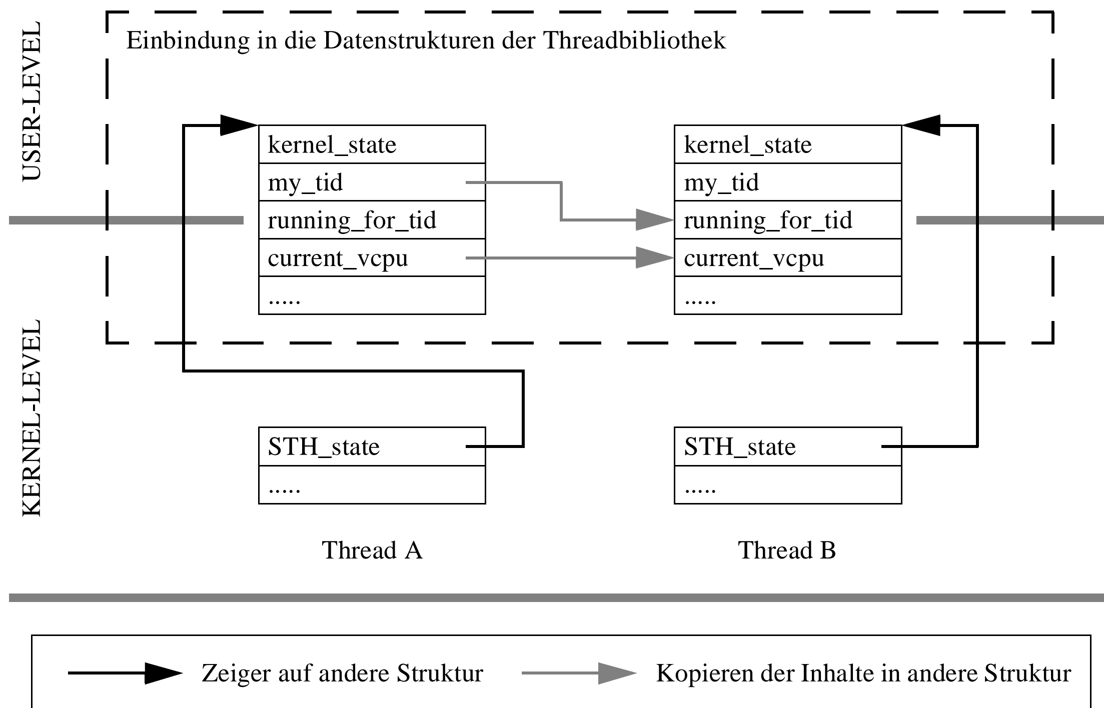

Abb. 6.5: Aufbau der Datenstruktur im gemeinsamen Speicherbereich

Scheduler überprüfen, ob der betroffene Thread ein *Sleeping-Thread* ist und gegebenenfalls nicht nach dem höchst priorisierten Thread in den Runqueues suchen, sondern auf einen freien Reserve-Thread umschalten. Im Fall einer Deblockierung darf der wieder lauffähige Thread nicht in eine Runqueue eingehängt werden, sondern muß nur in seinem gemeinsamen Speicherbereich als deblockiert markiert werden. Blockierungen und Deblockierungen werden in weit verteilten Bereichen des Mikrokerns angestoßen, nutzten jedoch ausnahmslos zentrale Funktionen, so daß sich die notwendigen Modifikationen auf zwei zentrale Stellen konzentrieren. Die dritte Modifikation ist im Dispatcher[6] notwendig, um neu hinzugekommene Fälle zu behandeln. Beispielsweise muß nach einer Umschaltung aufgrund der Fortsetzung eines deblockierten Threads durch die Threadbibliothek der alte Thread in die Liste der freien Reservethreads eingehängt werden.

Neben der Anforderung, Initialisierung und Freigabe von neuen Datenstrukturen, wie beispielsweise der gemeinsamen Speicherbereiche, bei der Generierung bzw. Terminierung von Tasks oder Threads, bilden die Erweiterungen in den genannten drei Codebereichen, zusammen mit den neu zu integrierenden Systemaufrufen, im wesentlichen den Kern des *Sleeping-Threads* Mechanismus. Die Konzentration der notwendigen Erweiterungen auf wenige Stellen ermöglicht eine übersichtliche und einfach zu realisierende Implementierung.

6. Der Dispatcher wird nach einer Threadumschaltung von dem neuen Thread durchlaufen, um den Zustand des alten, nun nicht mehr laufenden Threads anzupassen und den Thread erneut in eine Runqueue oder andere Datenstrukturen einzuhängen.

6.1.4.3 Vererbung an Serverthreads

Aufgrund des RPC Mechanismus, der von SPP-UX statt des Nachrichtenmechanismus von MACH für Serveraufrufe verwendet wird, ist die Vererbung des *Sleeping-Threads* Attributes an Serverthreads sehr einfach zu realisieren. Da Client- und Serverthread - bezüglich eines RPC - miteinander verzeigert sind, reicht es aus, bei Überprüfung des *Sleeping-Threads* Attributes und bei Zugriffen auf den gemeinsamen Speicherbereich grundsätzlich die entsprechenden Datenstrukturen des Clientthreads zu verwenden. Realisiert wird dieser Vorgang über ein Makro, das einen Zeiger auf die Threadstruktur des Clients oder, falls kein RPC und damit keine Client/Server-Verzeigerung vorliegt, des Threads selbst als Ergebnis liefert. Dadurch werden auch Threads des Servers, die selber keine *Sleeping-Threads* sind, entsprechend behandelt, wenn sie stellvertretend für einen *Sleeping-Thread* laufen. Für die Threadbibliothek ist dies vollständig transparent. Sie sieht nur eine entsprechende Statusänderung ihres eigenen Threads, auch wenn diese sich tatsächlich auf den Serverthread bezieht. Entsprechend muß, wenn die Threadbibliothek einen deblockierten Thread fortsetzt, der Mikrokern der Verzeigerung in umgekehrter Richtung folgen und statt auf den Clientthread auf den Server umschalten. Die Unterscheidung zwischen lokalen (innerhalb eines Nodes) und Remote-RPC's (Aufruf über Nodegrenzen hinweg) ist ebenfalls schon über die RPC-Implementierung von SPP-UX gewährleistet. Im lokalen Fall wird aus Effizienzgründen die Threadumschaltung von Client- auf Serverthread im Assembler kodiert. Die zentralen Routinen des Kernel-Schedulers werden nicht aufgerufen und folglich kann für den blockierenden Clientthread kein Reservethread freigegeben werden. Dies erfolgt erst dann, wenn der Serverthread blockieren sollte. Im Remotefall erfolgt die Blockierung durch die zentralen Routinen und damit auch eine entsprechende Behandlung von *Sleeping-Threads*.

6.1.4.4 Kritische Abschnitte

Die Implementierung der kritischen Abschnitte im Mikrokern selbst ist wiederum einfach zu realisieren. Zu Beginn eines kritischen Abschnitts wird ein Flag in den Datenstrukturen des Threads gesetzt und am Ende, soweit zwischenzeitlich keine Blockierung aufgetreten ist, wieder zurückgesetzt. Interessant ist der Fall einer Deblockierung, da der Thread innerhalb eines kritischen Abschnitts nicht unter die Kontrolle des User-Levels gestellt werden darf. Der Thread wird als in Fortsetzung begriffen markiert und erneut auf eine Runqueue gesetzt. Im Rahmen des Kernel-Schedulings wird er fortgesetzt und geht erst beim Verlassen des kritischen Abschnitts in den Zustand deblockiert über. Modifikationen gegenüber einer *Sleeping-Threads* Implementierung ohne kritische Abschnitte sind wiederum nur an wenigen zentralen Stellen erforderlich. Die Blockierung eines Threads während der Fortsetzung darf nicht zur Freigabe weiterer Reservethreads führen und der Dispatcher muß angepaßt werden, um ein erneutes Einhängen des Threads in die Runqueue zu erreichen. Darüber hinaus werden Funktionen zum Betreten und Verlassen eines kritischen Abschnitts benötigt, von denen erstere trivial ist, da lediglich ein Flag gesetzt werden muß. Letztere muß den Thread in den Zustand deblockiert überführen und im Rahmen des Kernel-Schedulings auf einen lauffähigen Thread umschalten.

Durch diese Erweiterung der *Sleeping-Threads* ist es möglich, Kernel-Threads innerhalb kritischer Abschnitte des Mikrokerncodes vor einer Kontrolle durch den User-Level zu schützen. In

einem MACH-basierten System reicht dies jedoch nicht aus. Kritische Abschnitte müssen auch von Emulator und Server aus gesetzt werden können. Im Gegensatz zu kritischen Abschnitten im Mikrokern, die sehr effizient betreten und verlassen werden können, muß hier die Effizienz des Zugriffes auf ein Flag im Mikrokern genauer betrachtet werden. Unkritisch wäre die Verwendung der gemeinsamen Speicherbereiche, die jedoch in diesem Fall nicht realisierbar ist. Die Implementierung der gemeinsamen Speicherbereiche zwischen Kern und User-Level in SPP-UX läßt keine Zugriffe seitens des Emulators zu. Darüber hinaus wäre die Nutzung des gemeinsamen Speichers ohnehin nicht in jedem Fall möglich, da zumindest beim Verlassen des kritischen Abschnitts ein Kerneinsprung unvermeidlich ist, wenn ein in Fortsetzung befindlicher Thread gestoppt werden soll. Im Fall von Serverthreads, die selbst keine *Sleeping-Threads* sind, aber mit solchen assoziiert sein können, ist nicht einmal ein gemeinsamer Speicherbereich mit dem Mikrokern verfügbar. Die Lösung liegt folglich in einer Realisierung durch möglichst effiziente Kerneinsprünge zu Beginn und Ende eines kritischen Abschnitts. Effizienz ist insbesondere deshalb sehr bedeutend, da im Gegensatz zum Emulator innerhalb des Servers nicht festgestellt werden kann, ob der entsprechende Clientthread ein *Sleeping-Thread* ist. Die Kerneinsprünge sind also in jedem Fall, auch für normale Threads, notwendig.

Um den Kerneinsprung zu optimieren und den Overhead des normalen Systemaufrufablaufs zu umgehen, werden für den Einsprung von Emulator und Server aus explizit reservierte Traps (Gateways in der Nomenklatur des HP PA-RISC) verwendet. Tabelle 6.1 zeigt den Vergleich zwischen einem normalen Systemaufruf und dem neu implementierten Trap zum Betreten bzw. Verlassen eines kritischen Abschnitts aus dem User-Level. Die Messung vom User-Level aus hat zwei Vorteile. Zum einen ist sie einfacher zu realisieren als innerhalb des Emulators oder Kernels und ist im Fall eines direkten Aufrufs des Mikrokerns äquivalent, da Emulator und Server ebenfalls User-Level Prozesse sind. Zum anderen ist gerade dieser Vergleich, und nicht der Vergleich zwischen zwei Varianten eines Kerneinsprungs vom Server aus, interessant, da kritische Abschnitte im Emulator und Server in den meisten Fällen aufgrund von Systemaufrufen aus einer Anwendung betreten werden. Von Interesse ist folglich der zusätzliche Overhead im Vergleich zu einem normalen Systemaufruf.

Einsprungart	µsec	Taktzyklen
Systemaufruf	82,5	8250
Optimierter Trap	2,1	210

Tab. 6.1: Kritische Abschnitte über Systemaufruf bzw. Trap

Der hochoptimierte Weg über einen reservierten Trap in den Mikrokern ist mit 2,1 µsec bzw. 210 Taktzyklen bei 100 MHz um den Faktor 40 schneller als ein Setzen des Flags für kritische Abschnitte über den normalen Systemaufrufpfad, der unter anderem den Umweg über den Emulator umfaßt. Beide Werte sind daher nicht vergleichbar[7], lassen aber auf einen in diesem Zusammenhang wichtigen Aspekt schließen. Bei den 82,5 µsec handelt es sich aufgrund der Kürze der eigentlichen Aufgabe (Setzen eines einzigen Wertes) im wesentlichen um den Overhead des

7. Vergleichbar wäre der optimierte Aufruf mit dem normalen Mikrokernaufruf des Emulators oder Servers in SPP-UX.

Systemaufrufs. Selbst ein mehrfaches Betreten und Verlassen von kritischen Abschnitten innerhalb eines Systemaufrufs ist daher vertretbar, zumal der gemessene Systemaufruf keine Interaktion mit dem Server umfaßt, sondern lediglich der Mikrokern aufgerufen wird.

Um die Anzahl der notwendigen kritischen Abschnitte in Emulator und Server einzugrenzen, behandelt der *Sleeping-Threads* Mechanismus keine Pagefaults, die im Emulator oder Server auftreten. Lediglich Pagefaults im Anwendungscode führen zur Freigabe eines Reservethreads und damit zur späteren Kontrolle durch den User-Level. Kritische Abschnitte im Emulator und Server sind daher nur notwendig, wenn wichtige Ressourcen gehalten werden und Blockierungen aufgrund expliziter Mikrokernaufrufe auftreten können.

Kritische Abschnitte werden in der aktuellen Implementierung nur an den notwendigsten Stellen eingesetzt. Im Mikrokern sind dies ein Abschnitt im Paging-Code und im Emulator einige Abschnitte bei der Erzeugung und Terminierung von Threads. Im Server werden derzeit keine kritischen Abschnitte genutzt. Damit sind bei weitem nicht alle kritischen Bereiche gesichert, was sich in gelegentlichen Systemverklemmungen u.a. bei massiven parallel ausgeführten E/A-Operationen zeigt. Die Stabilität des Systems reichte jedoch für die durchgeführten Tests und Messungen aus. Auf die Bedeutung dieser Einschränkungen für das EliTE-Projekt wird im weiteren Verlauf eingegangen.

6.2 Anbindung des User-Level-Schedulings

Die *Sleeping-Threads* sind nicht zur direkten Nutzung durch die Anwendungsprogrammierung gedacht, sondern dienen als Basis für ein besser in die Systemsoftware integriertes User-Level-Scheduling. Die ebenfalls im Rahmen des ELiTE-Projekts entwickelte User-Level-Threadbibliothek *m-threads* wurde an die neue Basis angepaßt und bietet, bis auf die Möglichkeit zur Konfiguration einiger Parameter, eine unveränderte Schnittstelle zur Anwendungsprogrammierung.

Die folgenden Abschnitte beschreiben den Kern einer Anbindung an die *Sleeping-Threads* am Beispiel der *m-threads* und gehen in allgemeinerer Form auf zwei weitere Aspekte ein, die Verwaltung der Reservethreads sowie die Synchronisation mit einem CPU-Scheduling im Betriebssystemkern. Anschließend werden die Auswirkungen auf die Effizienz des *m-threads* Schedulers erörtert.

6.2.1 Integration von *Sleeping-Threads* und *m-threads*

Die Erweiterungen der *m-threads* Bibliothek zur Anpassung an den *Sleeping-Threads* Mechanismus konzentrieren sich auf drei Bereiche: Die Einbindung in die Datenstrukturen für virtuelle Prozessoren und Threads, die Behandlung freigegebener Reservethreads in Zusammenhang mit Erweiterungen der eigentlichen Threadumschaltung sowie die Verwaltung deblockierter Threads.

6.2.1.1 Datenstrukturen für virtuelle Prozessoren und User-Level-Threads

Abbildung 6.6 stellt die Datenstrukturen der erweiterten *m-threads* Bibliothek und deren Verbindungen untereinander dar, wobei die neuen Bestandteile fett hervorgehoben sind. Jeder virtuelle Prozessor wird über eine Struktur (vom Typ `vcpu_t`) repräsentiert, die im wesentlichen den Zustand des virtuellen Prozessors (`state`), und einen Verweis auf den aktuell ausgeführten Thread (`cur_tp`) enthält. Jeder Thread (`mthread_t`) verweist auf den virtuellen Prozessor, auf dem er läuft bzw. zuletzt gelaufen ist (`prevcpup`). Weiterhin enthält die Threadstruktur den Zustand des Threads (`state`) und einen Verweis auf den genutzten Stack (`stack`). Eine Repräsentation der als virtuelle Prozessoren genutzten Kernel-Threads ist in der ursprünglichen *m-threads* Implementierung nicht notwendig, da jeder virtuelle Prozessor auf genau einen Kernel-Thread abgebildet ist. Lediglich ein privater 512 Byte umfassender Datenbereich, dessen Format aus Gründen der Kompatibilität zu der SPP-UX eigenen Threadbibliothek CPS[8] festgelegt ist, wird jedem Kernel-Thread zugeordnet[9]. Das erste Speicherwort dieses Bereichs enthält die Kernel-Thread-ID des Threads, das 32. Wort die Identifikation des repräsentierten User-Level-Threads (bzw. des virtuellen Prozessors in der *m-threads* Implementierung). Von Bedeutung in diesem Zusammenhang ist das Kontrollregister `cr27`. Es wird nach dem Start eines Kernel-Threads mit einem Verweis auf den Datenbereich des Threads (bzw. auf das 32. Wort) geladen und erlaubt im weiteren Ablauf eine effiziente Identifizierung des laufenden Threads, da der Inhalt des Registers im Rahmen des Kernel-Schedulings über Threadwechsel hinweg gerettet wird.

Die notwendigen Erweiterungen zur Anbindung der *Sleeping-Threads* ergeben sich aus der veränderten Repräsentation des virtuellen Prozessors. Ein virtueller Prozessor wird nicht mehr durch genau einen Kernel-Thread repräsentiert, da dieser im Fall einer Blockierung durch einen Reservethread ersetzt wird. Kernel-Threads benötigen daher eine eigene Repräsentation. Die entsprechende Struktur (`kthread_t`) wird ab Wort 32 in den - dem Thread zugeordneten - privaten Datenbereich eingebettet, um einen schnellen Zugriff auf die Struktur des aktuell laufenden Threads zu ermöglichen. Aus Kompatibilitätsgründen muß die `kthread_t`-Struktur daher als ersten Eintrag einen Verweis auf den virtuellen Prozessor enthalten. Weitere wichtige Einträge sind Verweise auf den laufenden User-Level-Thread (`mthread`) und den gemeinsamen Speicherbereich mit dem Mikrokern (`shm`). Die Bedeutung des Eintrags `STH_switcher` wird im folgenden Abschnitt erläutert. Entsprechende Rückverweise (`kthread`) müssen in den Repräsentationen von virtuellem Prozessor und User-Level-Thread gesetzt sein. Im Falle des User-Level-Threads führt dies zu Problemen, da die entsprechende Struktur aus Gründen der Effizienz in eine Cacheline passen soll und bereits voll ausgenutzt ist. Entgegen dem Eindruck, den Abbildung 6.6 vermittelt, ist dieser Verweis trotz seiner zusätzlichen Speicherung in der `vcpu_t`-Struktur keineswegs redundant. Er wird zur Identifikation des Kernel-Threads benötigt, an den ein deblockierter User-Level-Thread gebunden ist. Der indirekte Weg über die `vcpu_t`-Struktur ist in diesem Fall nicht möglich, da ein deblockierter und damit aktuell nicht

8. **C**ompiler **P**arallel **S**upport Library
9. Diese kompatiblen Datenstrukturen erlauben die Nutzung von Erweiterungen der Systembibliothek (libc), die den sicheren Aufruf verschiedener Bibliotheksfunktionen parallel durch mehrere Threads sicher stellen.

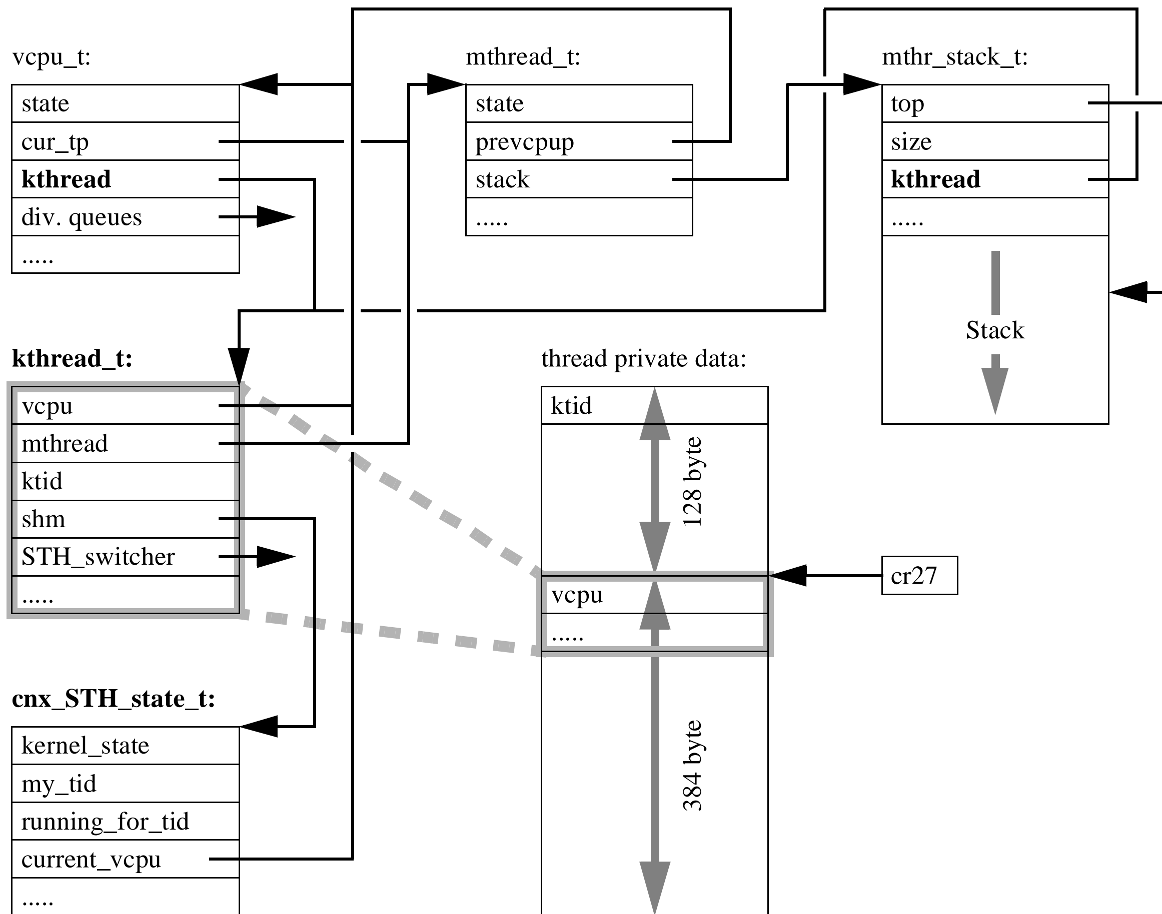

Abb. 6.6: Einbindung der *Sleeping-Threads* in die *m-threads* Datenstrukturen

ausgeführter User-Level-Thread nicht an einen virtuellen Prozessor gebunden ist. Daher wurde der Verweis auf den Kernel-Thread in die Stackstruktur verlagert.

Die beschriebenen Erweiterungen der Datenstrukturen reichen, wie im folgenden ausgeführt wird, zur Anbindung an den *Sleeping-Threads* Mechanismus aus und sind derart angelegt, daß keine grundsätzlichen Änderungen an der Struktur der *m-threads* Bibliothek notwendig werden. Im Fall einer Blockierung kann der freigegebene Reservethread über das Kontrollregister `cr27` auf seine Threadstruktur und damit auf den ihm zugeordneten gemeinsamen Speicher zugreifen. Über den Eintrag `current_vcpu`, der, wie in Abschnitt 6.1.4.1 beschrieben, durch den Mikrokern von dem blockierten Thread übernommen wird, ist die Identifikation des virtuellen Prozessors bekannt. Nach einer Aktualisierung der Verzeigerung zwischen virtuellem Prozessor und Kernel-Thread kann der Reservethread im Kontext des gleichen virtuellen Prozessors das User-Level-Scheduling fortsetzen. Zusätzlich muß der Verweis auf den Kernel-Thread in der Stackstruktur des blockierten User-Level-Threads gesetzt werden. Der umgekehrte Weg, die Fortsetzung eines deblockierten Threads, geht von dem an ihn gebundenen User-Level-Thread aus. Über den Verweis in der Stackstruktur ist der assoziierte Kernel-Thread und sein im gemeinsamen Speicher gespiegelter Zustand zugreifbar. Nach erneuter Aktualisierung zwischen virtuellem Prozessor und Kernel-Thread kann über einen Systemaufruf der deblockierte Kernel-Thread fortgesetzt und der aufrufende Thread als Reservethread geparkt werden.

6.2.1.2 *Sleeping-Threads* und Threadumschaltung im User-Level

Die beschriebene Erweiterung der Datenstrukturen der *m-threads* Bibliothek erlaubt die Speicherung und den effizienten Zugriff auf die Informationen, die für ein Aufsetzen der *m-threads* auf den *Sleeping-Threads* Mechanismus notwendig sind. Eine andere Erweiterung, ein zusätzlicher User-Level-Thread pro Kernel-Thread, der sogenannte Switcherthread, der einem Kernel-Thread fest zugeordnet ist, ermöglicht in einfacher Weise, die Umschaltung eines User-Level-Threads auf einen an einen deblockierten Kernel-Thread gebundenen Thread in die *m-threads* Bibliothek zu integrieren. Ein freigegebener Reservethread läuft im Kontext des Switcherthreads bis zur Umschaltung auf einen neu ausgewählten User-Level-Thread.

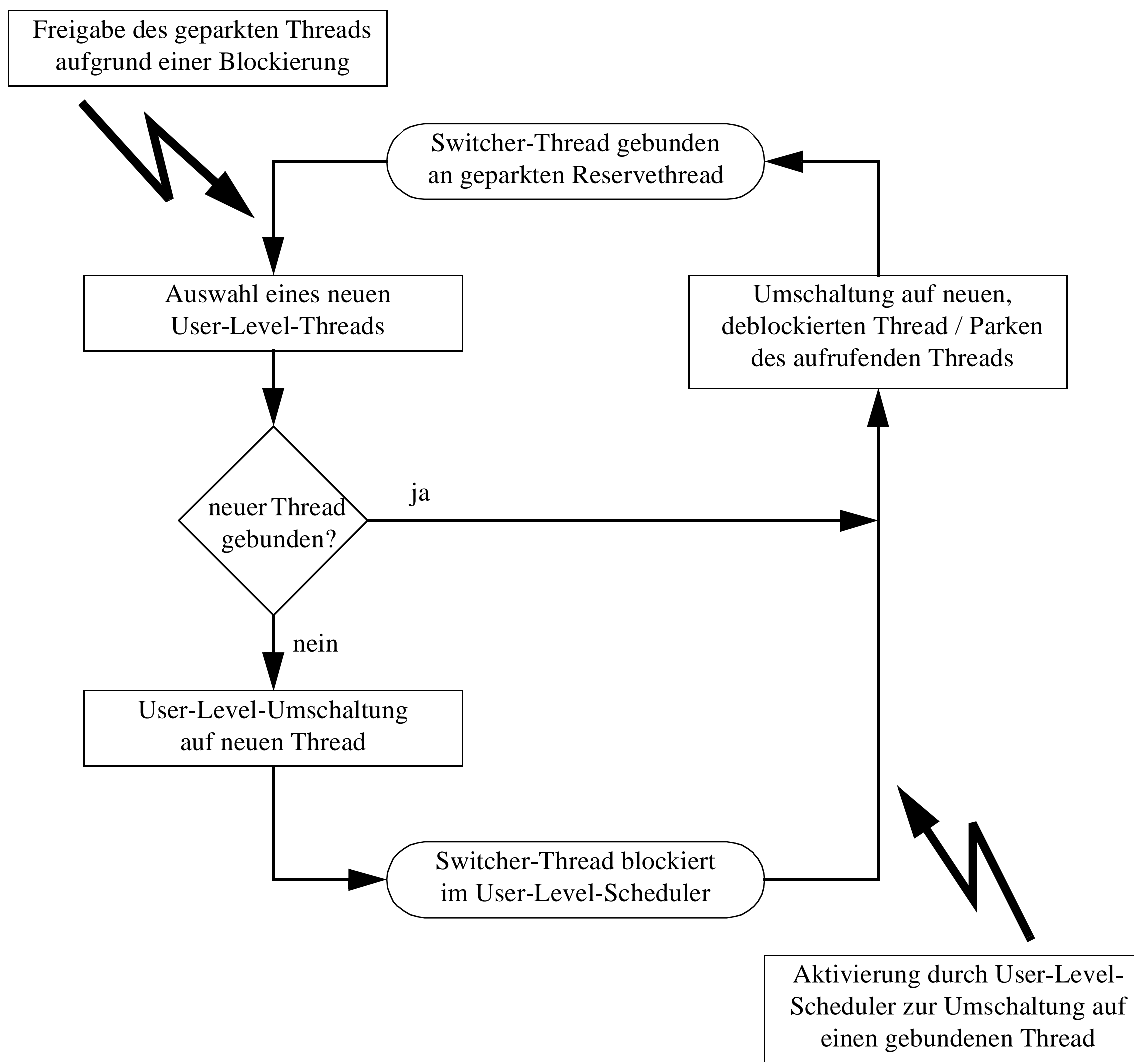

Abb. 6.7: Die Endlosschleife des Switcherthreads

Der Switcherthread durchläuft, bis auf eine Initialisierungsphase, eine Endlosschleife mit zwei Zuständen, in denen er während des überwiegenden Anteils der Laufzeit blockiert ist (siehe Abbildung 6.7). Im ersten Ruhezustand ist er an einen geparkten Reservethread gebunden und wird im Rahmen einer Blockierung durch Freigabe des Reservethreads aktiviert. Im Kontext des virtuellen Prozessors, für den er aktiviert wurde, wählt er einen auszuführenden User-Level-

Thread aus. Ist dieser an einen deblockierten Kernel-Thread gebunden, erfolgt die Threadumschaltung über einen Systemaufruf durch den *Sleeping-Threads* Mechanismus im Kern, wobei der Kernel-Thread des Switcherthreads erneut geparkt wird. Im anderen Fall erfolgt eine Threadumschaltung im User-Level, wodurch der Switcherthread den zweiten möglichen Ruhezustand, eine Blockierung innerhalb des User-Level-Schedulers, einnimmt. In diesem Zustand bleibt der Switcherthread, wenn der zugeordnete Kernel-Thread als virtueller Prozessor aktiv ist. Diesen Zustand verläßt er erst dann, wenn der zugehörige Kernel-Thread im Rahmen des User-Level-Schedulings auf einen gebundenen Thread umschaltet. Die Umschaltung erfolgt indirekt über eine User-Level-Umschaltung auf den Switcherthread. Der Kontext des laufenden Threads wird dadurch auf normalem Weg im Rahmen einer User-Level-Umschaltung gesichert. Erst der Switcherthread schaltet auf den gebundenen Thread um und ist danach, gebunden an einen geparkten Kernel-Thread, wieder im Ausgangszustand.

Die Integration der Umschaltung auf gebundene Threads in die *m-threads* Bibliothek ist dadurch sehr einfach gelöst. Vor der eigentlichen Umschaltung muß überprüft werden, ob der neue Thread ein gebundener Thread ist. Dies ist im Rahmen eines Makros realisiert, das gegebenenfalls den Verweis auf den neuen Thread sichert und durch einen Verweis auf den Switcherthread ersetzt. Anschließend kann die Threadumschaltung unverändert aufgerufen werden, mit dem Unterschied, daß der Switcherthread aktiviert wird und über den gesicherten Verweis auf den gebundenen Thread umschalten kann.

6.2.1.3 Verwaltung gebundener Threads

Der dritte Bereich der Integration von *Sleeping-Threads* und *m-threads* ist die Verwaltung gebundener Threads, also von User-Level-Threads, die mit einem blockierten oder deblockierten Kernel-Thread assoziiert sind. Im Fall einer Blockierung wird über den Reservethread der User-Level-Scheduler aktiv und hängt den gebundenen Thread in eine, jedem virtuellen Prozessor zugeordnete, verkettete Liste für blockierte Threads (Blockedqueue) ein. Die Deblockierung eines Threads wird dagegen nicht aktiv an den User-Level-Scheduler propagiert, sondern kann nur über eine regelmäßige Überprüfung des im gemeinsamen Speicher gespiegelten Zustands erkannt werden. Analog zu der Deblockedqueue, die bei jedem Aufruf des Schedulers in die lokale Runqueue des virtuellen Prozessors übertragen wird, wird auch die Blockedqueue durchlaufen, mit dem Unterschied, daß nicht alle, sondern nur die gebundenen Threads in die Runqueue übertragen werden, deren gespiegelter Kernel-Thread-Zustand von blockiert nach deblockiert übergegangen ist. Im Rahmen des Lastausgleichalgorithmus werden ebenfalls die Blockedqueues anderer virtueller Prozessoren durchsucht. In diesem Fall werden gefundene deblockierte Threads jedoch nicht in die Runqueue, die nach Möglichkeit nur durch den eigenen virtuellen Prozessor referenziert werden sollte, sondern in die Deblockedqueue eingehängt (vergleiche Abschnitt 5.4.1). Gebundene Threads können im Rahmen des Lastausgleiches auch auf andere virtuelle Prozessoren migriert werden. Dies ist jedoch nicht über Nodegrenzen hinweg möglich, da der assoziierte Kernel-Thread an seinen Node gebunden ist.

6.2.1.4 Erweiterungen des *Sleeping-Threads* Mechanismus

Zur Anbindung an die *m-threads* Bibliothek waren Erweiterungen des *Sleeping-Threads* Mechanismus notwendig, von denen die beiden wichtigsten erwähnt werden sollen. Die erste ist von allgemeiner Bedeutung für die Anbindung von User-Level-Threadbibliotheken. Über das Setzen eines Flags (Disable-Flag) im gemeinsamen Speicherbereich ist es in effizienter Weise möglich, die spezielle Behandlung eines *Sleeping-Threads* im Fall einer Blockierung zeitweilig zu unterbinden. Damit ist die Implementierung eines User-Level-Schedulers erheblich vereinfacht, da während des Durchlaufens der Schedulingroutinen, insbesonderer derer, in denen die Behandlung der Reservethreads erfolgt, die Freigabe von Reservethreads ausgeschaltet werden kann. In diesen Fällen ist es günstiger, eine kurzzeitige Blockierung des virtuellen Prozessors hinzunehmen.

Die zweite Erweiterung liegt in den speziellen Anforderungen der *m-threads* begründet. Für die Einbeziehung des Cachezustandes in die Schedulingstrategien ist ein effizienter Zugriff auf die Cachemiss-Zähler notwendig. Unter SPP-UX werden die Zähler der Hardware virtualisiert und für jeden Kernel-Thread als Bestandteil des Threadzustands geführt. SPP-UX ermöglicht einen effizienten Zugriff jedoch nur auf den Zähler des laufenden Threads. Der Zugriff auf Zähler anderer Threads ist nur über einen Systemaufruf möglich, der im Rahmen eines effizienten Schedulings nicht nutzbar ist. Um gebundene Threads nahtlos in das Scheduling der *m-threads* Bibliothek einzubinden, ist jedoch ein Zugriff auf deren Cachemiss-Zähler durch die im Rahmen der Blockierung freigegebenen Reservethreads notwendig. Im Fall einer Blockierung werden daher Ereigniszähler eines Threads, die unter anderem als Cachemiss-Zähler konfiguriert werden können, in den gemeinsamen Speicher kopiert und sind damit effizient zugreifbar.

Andere für die Funktionalität nicht unbedingt notwendige Erweiterungen dienen statistischen und messtechnischen Zwecken. Die gewonnenen Daten können zum Teil auch während der Laufzeit zur Optimierung genutzt werden. Eine Auswertung des Zählers für die Anzahl der Fälle, in denen kein geparkter Reservethread verfügbar ist, könnte beispielsweise die Erzeugung weiterer Reservethreads anstoßen.

6.2.2 Verwaltung der Reservethreads

Die *Sleeping-Threads* bilden nur den Mechanismus, der das Ersetzen eines blockierten virtuellen Prozessors einer User-Level-Threadbibliothek ermöglicht. Es liegt in der Verantwortung der Threadbibliothek, genügend Reservethreads im Betriebssystemkern zu parken. Eine Threadbibliothek muß über die Anzahl der verfügbaren Reservethreads Buch führen und gegebenenfalls weitere erzeugen, um Engpässe zu vermeiden, oder überflüssige terminieren, um nicht unnötig über längere Zeit Ressourcen zu belegen. Die Reservethreadverwaltung der *m-threads* Bibliothek ist rudimentär. Es kann lediglich konfiguriert werden, wieviele Reservethreads pro virtuellen Prozessor in der Initialisierungsphase geparkt werden. Eine nachträgliche Erzeugung ist nicht möglich.

Eine Verbesserung dieses rudimentären Ansatzes ist über zwei Wege denkbar. Die aktuelle *Sleeping-Threads* Implementierung behandelt einen blockierenden *Sleeping-Thread* im Rah-

men des normalen Kernel-Schedulings, wenn kein geparkter Reservethread verfügbar ist. Eine Erweiterung könnte für diesen Ausnahmefall die Aktivierung der Threadbibliothek erlauben, um weiter Reservethreads zu erzeugen. Eine bessere Lösung wäre indessen, Strategien in der Threadbibliothek zu implementieren, die die Verfügbarkeit von Reservethreads sicherstellen und damit die Blockierung eines virtuellen Prozessors oder die Aktivierung teurer Ausnahmebehandlungsmechanismen vermeiden. Zudem kann mit einer sehr einfachen Strategie garantiert werden, daß immer genügend Reservethreads verfügbar sind. Entspricht die Anzahl der verfügbaren Reservethreads mindestens der Anzahl der virtuellen Prozessoren, ist selbst im Fall einer gleichzeitigen Blockierung aller virtueller Prozessoren eine genügende Anzahl vorhanden. Sind die Routinen der Threadbibliothek, insbesondere die Behandlung eines freigegebenen Reservethreads, durch Ausschalten des Mechanismus über das Setzen des Disable-Flags im gemeinsamen Speicher gegen das Freigeben weiterer Reservethreads geschützt, kann sichergestellt werden, daß diese Mindestanzahl nicht unterschritten wird. Ein freigegebener Reservethread muß als erstes überprüfen, ob die Mindestanzahl unterschritten ist und gegebenenfalls neue Reservethreads erzeugen. Die kurzfristige Unterschreitung ist tolerierbar, da eine Blockierung des Threads zu diesem Zeitpunkt nicht zu einer Freigabe eines Reservethreads führen würde. Strategien sollten sich jedoch nicht auf diesen Punkt beschränken, da ein mehrfaches aufeinanderfolgendes Auflaufen auf die Mindestgrenze bei jeder Blockierung zur Erzeugung neuer Threads führen würde. Die rechtzeitige Bereitstellung mehrerer Reservethreads ist aus Effizienzgründen sinnvoller. Der Abbau überflüssiger Threads sollte ebenfalls integriert werden, um nach einem einmalig hohen Bedarf - zum Beispiel beim Anlauf einer Anwendung - nicht benötigte Ressourcen wieder freizugeben.

6.2.3 Dynamische Partitionierung - Synchronisation mit CPU-Scheduling

Der *Sleeping-Threads* Mechanismus ist für viele Anwendungen nur dann sinnvoll einsetzbar, wenn eine Anwendung auf einer Menge von exklusiv zugewiesenen Prozessoren ausgeführt wird. Für die aktuelle Implementierung wird dies durch eine exklusive Ausführung der *Sleeping-Threads* Anwendungen in den von SPP-UX unterstützten statischen Partitionen erreicht. Die statische Partitionierung eines Rechensystems kann jedoch gravierende Nachteile haben, wie in Abschnitt 4.4.2 dargestellt wird.

Eine dynamische Partitionierung, basierend auf einem CPU-Scheduling, erfordert jedoch eine zusätzliche Synchronisation zwischen Kernel-Scheduler, *Sleeping-Threads* Mechanismus und Threadbibliothek. Neben der Realisierung der dynamischen Partitionierung selbst, ist der Aufwand für die Integration mit den *Sleeping-Threads*, aufgrund deren orthogonaler Konzeption, jedoch gering. Insbesondere sind keine grundsätzlichen Erweiterungen in der Kommunikation zwischen Threadbibliothek und Betriebssystemkern notwendig. Wie eine Synchronisation zwischen CPU-Scheduler und *Sleeping-Threads* bzw. Threadbibliothek aussehen könnte, zeigt das folgende Fallbeispiel.

In einem MACH-basierten System wie SPP-UX liegt es nahe, eine dynamische Partitionierung auf Basis der Prozessorsets zu realisieren. Jeder parallelen Anwendung wird ein eigenes Prozessorset zugewiesen. Eine zusätzliche Schedulingkomponente im Kern, der CPU-Scheduler,

verteilt die Prozessorressourcen zwischen den Anwendungen bzw. Prozessorsets. Da Thread- und CPU-Scheduling des Betriebssystemkerns voneinander unabhängig sind, ist es beispielsweise möglich, sequentielle Anwendungen gemeinsam in einem Prozessorset auszuführen. Für das CPU-Scheduling sind verschiedenste Strategien denkbar. Interessanter bezüglich einer Integration mit den *Sleeping-Threads* ist jedoch die Methode zur Erhebung des Prozessorbedarfs der einzelnen Prozessorsets. Naheliegend ist eine Orientierung an der durch die ablauffähigen Kernel-Threads erzeugten Last. In Zusammenhang mit den *Sleeping-Threads* führt eine alleinige Umsetzung dieses Ansatzes jedoch zu einem Problem, da es gerade die Zielsetzung der *Sleeping-Threads* ist, die Anzahl der ablauffähigen Kernel-Threads an die der verfügbaren Prozessoren anzupassen. Es entsteht folglich keine erhöhte Last, die eine Zuweisung weiterer Prozessoren induzieren würde. Unerläßlich für die Integration der *Sleeping-Threads* ist daher ein CPU-Scheduling, das Bedarfsanforderungen durch die Anwendungen zuläßt und in den Schedulingalgorithmus einfließen läßt.

Ausgehend von dem Ziel der *Sleeping-Threads*, die Anzahl der physikalischen und virtuellen Prozessoren in Übereinstimmung zu halten, sind offensichtlich zwei Synchronisationspunkte zwischen CPU-Scheduler und *Sleeping-Threads* Mechanismus bzw. Threadbibliothek zu beachten: Das Hinzufügen und Entziehen eines Prozessors. Bei der Zuweisung eines weiteren Prozessors wird ein Reservethread freigegeben und auf dem neuen Prozessor ausgeführt. Die Unterscheidung zwischen einer Freigabe aufgrund einer Blockierung oder eines neu zugewiesenen Prozessors kann innerhalb der Threadbibliothek anhand eines Flags im gemeinsamen Speicher erfolgen. Der Reservethread übernimmt in diesem Fall nicht die weitere Ausführung eines blockierten virtuellen Prozessors, sondern wird als zusätzlicher virtueller Prozessor genutzt. Auch beim Entzug eines Prozessors können weitgehend vorhandene Mechanismen genutzt werden. Ein auf dem zu entziehenden Prozessor laufender Thread kann zu einem sicheren Zeitpunkt, d.h. außerhalb kritischer Abschnitte, unterbrochen und im Rahmen des *Sleeping-Threads* Mechanismus wie ein deblockierter Thread behandelt werden. Dies ermöglicht, ausgehend von der Threadbibliothek, eine Fortsetzung des Threads durch einen anderen virtuellen Prozessor. Ein verbleibendes Problem liegt in der fehlenden Propagierung des neuen Zustands in den User-Level, da anders als im Fall einer Blockierung der Thread nicht durch einen freigegebenen Reservethread der Threadbibliothek als blockiert bekannt gegeben wird. Folglich wird er im Rahmen des User-Level-Schedulings nicht regelmäßig auf den Zustand deblockiert überprüft und damit nicht als fortsetzbar erkannt. Um zu vermeiden, daß die Threadbibliothek zur Erkennung solcher Threads regelmäßig alle virtuellen Prozessoren auf einen Prozessorentzug überprüfen muß, wäre eine Erweiterung der *Sleeping-Threads* sinnvoll. Beispielsweise könnte ein weiterer gemeinsamer Speicherbereich eingeführt werden, der nicht einem einzelnen Thread, sondern der gesamten Task zugeordnet ist. An einer solchen zentralen Datenstruktur wäre eine regelmäßige Überprüfung in effizienter Form möglich. Der Entzug eines Prozessors ist folglich über den *Sleeping-Threads* Mechanismus in Vergleich zu den *Scheduler Activations*, die dies mit einem hohen Aufwand und der Unterbrechung eines weiteren virtuellen Prozessors realisieren, und den *First-Class User-Level-Threads*, die ein weiteres Scheduling des auf dem entzogenen Prozessor laufenden User-Level-Thread ohne ein vorausgehendes Eingreifen des User-Level-Schedulers nicht ermöglichen, sehr einfach gelöst.

Mit der - aus der dynamischen Partitionierung hervorgehenden - wechselnden Zahl von virtuellen Prozessoren kann innerhalb der Threadbibliothek auf verschiedene Weise umgegangen werden. Eine Variante besteht darin, die Anzahl der maximal nutzbaren virtuellen Prozessoren mit der Prozessoranforderung an den CPU-Scheduler gleichzusetzen. Kann die Prozessoranforderung nicht oder zeitweise nicht erfüllt werden, wird eine entsprechende Zahl von virtuellen Prozessoren nicht genutzt. Dies kann jedoch dazu führen, daß nach der Zuweisung eines freien Prozessors ein virtueller Prozessor auf einem anderen Prozessor als bei der letzten Aktivierung ausgeführt wird. In Zusammenhang mit dem Lokalitätsscheduling der *m-threads* Bibliothek erscheint es daher sinnvoller, jedem potentiell nutzbaren physikalischen Prozessor einen eigenen virtuellen Prozessor zuzuweisen, auch wenn eine gleichzeitige Nutzung aller Prozessoren nicht vorgesehen ist. Es würde eine generelle Unterscheidung zwischen aktiven und inaktiven virtuellen Prozessoren erfolgen und nicht - wie bei der ersten Variante - ein nicht nutzbarer virtueller Prozessor als Ausnahme behandelt werden. Ob dieser Ansatz aus Sicht einer guten Cacheausnutzung sinnvoll wäre, ist jedoch zweifelhaft. Da ein CPU-Scheduling auf eine möglichst lange exklusive Bindung eines Prozessors an eine Anwendung ausgelegt ist, um einen optimalen Ablauf zu ermöglichen, kann nach einer erneuten Zuweisung eines Prozessors an eine Anwendung ohnehin kein relevanter Cacheinhalt mehr erwartet werden.

6.2.4 Kosten der *Sleeping-Threads*

Der *Sleeping-Threads* Mechanismus ermöglicht es, im User-Level-Scheduler auf Blockierungen im Betriebssystemkern zu reagieren und im Kontext eines Reservethreads den virtuellen Prozessor fortzusetzen, um einen anderen ablauffähigen User-Level-Thread auszuführen. Der User-Level-Scheduler verhält sich damit analog zum Kernel-Scheduler, der einen blockierten Kernel-Thread durch einen anderen ersetzt, um die freigewordene CPU weiter auszulasten. Die Messung beider Varianten erfolgte ausgehend von einer Anwendung. Die gemessenen Werte (siehe Tabelle 6.2) entsprechen der Zeitdauer vom Aufruf eines blockierenden Systemaufrufs (`cnx_thread_block`) bis ein anderer Thread der Anwendung erneut den Anwendungscode ausführt. Basierend auf Kernel-Threads sind dies 275 µsec. Ein User-Level-Scheduler mit *Sleeping-Threads* Unterstützung ist mit 321 µsec um den Faktor 1,17 langsamer. Das Ergebnis entspricht den Erwartungen, da in beiden Fällen eine Threadumschaltung im Betriebssystemkern und im Fall des User-Level-Schedulings eine Aktivierung des Schedulers sowie eine weitere Threadumschaltung im User-Level (vom Switcherthread der Threadbibliothek auf den Thread der Anwendung) erfolgt. Für einen Vergleich zwischen einem User-Level-Scheduling mit und ohne *Sleeping-Threads* Unterstützung ist dieses Ergebnis jedoch von untergeordneter Bedeutung, da ein blockierter virtueller Prozessor in der unmodifizierten *m-threads* Bibliothek ohnehin nicht weiter genutzt werden kann. Es zeigt jedoch, daß eine systemnahe Anwendung auch im Fall einer hohen Anzahl von Blockierungen auf Basis eines integrierten User-Level-Schedulings mit einem akzeptablen zusätzlichen Overhead für Blockierungen gegenüber einer Realisierung mit Kernel-Threads möglich ist. Verglichen mit den 10 µsec für eine User-Level-

Threadumschaltung der *m-threads* Bibliothek verdeutlicht dieses Ergebnis erneut die Bedeutung des User-Level-Schedulings.

Scheduler	µsec	Faktor
Kernel	275	1,00
User-Level (*Sleeping-Threads*)	321	1,17

Tab. 6.2: Umschaltung auf neuen Thread nach Blockierung im Kern

Tabelle 6.3 zeigt den Overhead der *Sleeping-Threads* Unterstützung für den Fall, daß keine Blockierungen im Betriebssystemkern auftreten. Der eigentliche *Sleeping-Threads* Mechanismus wird also überhaupt nicht aktiv. Gegenüber einer *m-threads* Version ohne *Sleeping-Threads* Unterstützung beträgt der zusätzliche Overhead 1,3% und verdoppelt sich auf 2,6% für nicht unbedingt notwendige Statistik- und Messroutinen. Die angegebenen Werte basieren auf der Messung der Gesamtlaufzeit eines Testprogramms, ohne Initialisierungsphase der *m-threads* Bibliothek, die in der *Sleeping-Threads* Version u.a. die Erzeugung der Reservethreads beinhaltet. Das Testprogramm synchronisiert unter Verwendung von acht virtuellen Prozessoren 16 Paare von User-Level-Threads je 100.000 mal über Semaphore-Operationen. In einer realen Anwendung, die nicht ausschließlich Routinen der Threadbibliothek aufruft, kann der Overhead daher vernachlässigt werden.

Programmlauf ...	Faktor
ohne *Sleeping-Threads* Unterstützung	1,000
mit *Sleeping-Threads* Unterstützung	1,013
mit Sleeping-Threads und Messroutinen	1,026

Tab. 6.3: *Sleeping-Threads* Overhead

6.3 Einsatzgebiete und Anwendungen

6.3.1 Ergonomische Programmierung / Verklemmungsgefahr

Eine der Schwächen des User-Level-Schedulings ist, wie in Abschnitt 4.4.2.3 dargestellt wurde, daß ein User-Level-Scheduler sich im allgemeinen nicht so verhält, wie ein Anwender dies von dem Programmiermodell Threads erwartet. Dies kann zu Parallelitätseinbrüchen bis hin zu Verklemmungen führen oder zwingt zu einer komplexeren und damit fehleranfälligen Programmierung, um problematische Stellen zu umbauen. Der *Sleeping-Threads* Mechanismus erlaubt die Implementierung eines User-Level-Schedulers, dessen virtuelle Prozessoren sich analog zur Prozessorabstraktion des Betriebssystemkerns verhalten. Er ermöglicht eine verklemmungsfreie Implementierung des Programmiermodells Threads als User-Level-Bibliothek, wenn sichergestellt ist, daß Reservethreads immer verfügbar sind. Dies ist, wie in Abschnitt 6.2.2 gezeigt wurde, mit sehr einfachen Strategien möglich. Besondere Vorkehrungen an kritischen Stellen sind damit, wie auch bei Verwendung von Kernel-Threads, nicht mehr notwendig.

Anwendungen, die unter einem Kernel-Scheduling problemlos laufen, unter einem User-Level-Scheduling aber zu langdauernden Blockierungen virtueller Prozessoren oder sogar zu Verklemmungen führen, sind einfach zu konstruieren. Vergleichende Messungen ohne und mit einer *Sleeping-Threads* Unterstützung sind in diesem Fall nicht sinnvoll. Die interessante Fragestellung ist, wieviel zusätzlichen Overhead die bessere Abstraktion des virtuellen Prozessors kostet. Die Ergebnisse aus Abschnitt 6.2.4 zeigen, daß sich Kosten in einem akzeptablen Rahmen halten.

6.3.2 E/A-intensive Anwendungen

Zur Untersuchung des Ein-/Ausgabe-Verhaltens von *Sleeping-Threads* Anwendungen wurden zwei UNIX Systemprogramme mit massivem Dateisystemzugriff gewählt. Als Eingangsbeispiel dient das begrenzt parallelisierbare *grep*, das eine Textdatei nach Vorkommen bestimmter Inhalte, die in Form von regulären Ausdrücken angegeben werden können, durchsucht. Weitergehende Untersuchungen basieren auf dem Programm *du* (***d**isc **u**sage*). *du* durchläuft einen gesamten Dateibaum und addiert die Größe der einzelnen Dateien auf. Als Schedulingstrategie wurde in beiden Fällen *vtime* eingesetzt.

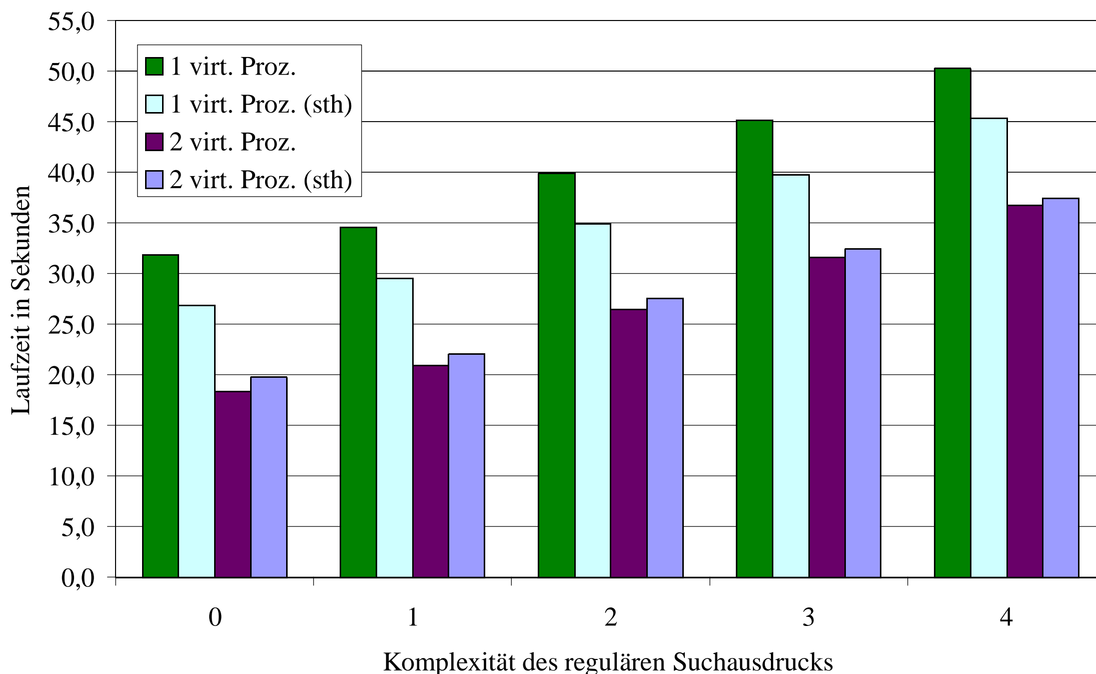

Abb. 6.8: *grep* mit parallelem Einlesen der Daten

Die Parallelisierung von *grep* basiert auf einer Aufgabenteilung zwischen zwei Threads. Ein Thread liest die Datei in einen Ringpuffer ein, der andere durchsucht diesen Puffer. Dadurch können Einlesen und Durchsuchen der Daten parallel ausgeführt werden. Abbildung 6.8 zeigt die Laufzeiten für das Durchsuchen einer 60 MByte großen Datei mit steigender Komplexität des regulären Suchausdrucks unter Verwendung von einem und zwei virtuellen Prozessoren. Die Abkürzung **sth** in dieser und den folgenden Abbildungen bezeichnet die Messreihen, die

mit *Sleeping-Threads* Unterstützung durchgeführt wurden. Bei Verwendung eines virtuellen Prozessors ist nur ein pseudo-paralleler Ablauf möglich, da nicht beide Threads gleichzeitig ausgeführt werden können. Ausgenutzt werden kann jedoch die betriebssysteminterne Parallelität durch unabhängige E/A-Prozessoren. Möglich ist dies nur im Rahmen eines User-Level-Schedulings mit *Sleeping-Threads* Unterstützung, da ansonsten ein blockierender Leseaufruf des einen Threads ebenfalls den virtuellen Prozessor blockiert und damit ein Scheduling des zweiten Threads nicht möglich ist. Der Vergleich beider Versionen zeigt, inwieweit die *Sleeping-Threads* ein Interleaving zwischen E/A und Berechnung erlauben. Die Laufzeitersparnis ist erwartungsgemäß unabhängig von der Komplexität des regulären Ausdrucks.

Werden zwei virtuelle Prozessoren genutzt, geht der Vorteil der *Sleeping-Threads* verloren, da beide Threads, soweit sie ablauffähig sind, gleichzeitig ausgeführt werden können. Der Overhead der *Sleeping-Threads* ist verglichen mit den Ergebnissen aus Abschnitt 6.2.4 (maximal 1,3 bis 2,6%) überraschend hoch, aber nachvollziehbar. Im Gegensatz zum Kernel-Scheduler, der nach abgeschlossener E/A-Operation einen Thread sehr schnell fortsetzen kann, muß in der *Sleeping-Threads* Version der gespiegelte Zustand im gemeinsamen Speicher regelmäßig überprüft werden, um die Deblockierung zu erkennen. Dies führt zu einer zusätzlichen Verzögerung und zu einer Umschaltung auf den Idle-Thread des virtuellen Prozessors, da in diesem Beispiel kein weiterer User-Level-Thread verfügbar ist, der stattdessen ausgeführt werden könnte. Die weitere Ersparnis gegenüber der *Sleeping-Threads* Version auf einem virtuellen Prozessor liegt darin begründet, daß die Leseaufrufe nicht nur einen blockierenden Anteil (Lesen der Daten durch den E/A-Prozessor) haben, sondern zusätzlich die Daten aus dem E/A-Puffer im Betriebssystemkern in den Adreßraum der Anwendung kopiert werden müssen. Dieser Anteil kann erst bei der Verwendung von zwei virtuellen Prozessoren parallel mit dem durchsuchenden Thread ausgeführt werden.

Das Durchlaufen eines Dateibaums und Aufsummieren der Dateigrößen durch das Programm *du* ist für eine Parallelisierung geeigneter als das Durchsuchen einer Datei. Der Dateibaum wird von der Wurzel aus beginnend mit einem Thread durchlaufen. Trifft der Thread auf ein Unterverzeichnis, durchläuft er dieses nicht selbst, sondern aktiviert, soweit nicht ein einstellbares Maximum erreicht ist, einen weiteren Thread, der den entsprechenden Unterbaum auf gleiche Weise bearbeitet, und fährt mit dem nächsten Eintrag fort. Hat ein Thread einen Teilbereich abgearbeitet, wird er suspendiert und kann durch einen anderen Thread erneut für einen Unterbaum aktiviert werden. Das verwendete parallele *du* wird in [Hauth95] näher beschrieben.

Abbildung 6.9 zeigt die Laufzeit auf vier virtuellen Prozessoren mit einem Maximum von ein bis acht Threads. Die Ergebnisse stellen einen Mittelwert aus jeweils acht Messungen dar. Die Fehlerbalken zeigen die Standardabweichung. Der verwendete Dateibaum umfaßt 1083 Verzeichnisse mit 21258 Dateien. Für die *Sleeping-Threads* Version ist die Anzahl der Reservethreads derart dimensioniert, daß die Verfügbarkeit in allen Fällen garantiert ist. Das Ergebnis entspricht, in der Version ohne *Sleeping-Threads*, den Erwartungen. Ein Speedup ist möglich, solange eine genügende Anzahl von virtuellen Prozessoren eine echte Parallelität aller Threads erlaubt. Übersteigt die Anzahl der Threads die der vier virtuellen Prozessoren, ist eine weitere Verbesserung nicht möglich, da Blockierungen im E/A-System nicht durch die Ausführung lauffähiger Threads überbrückt werden können. Die dennoch auftretende leichte Verbesserung

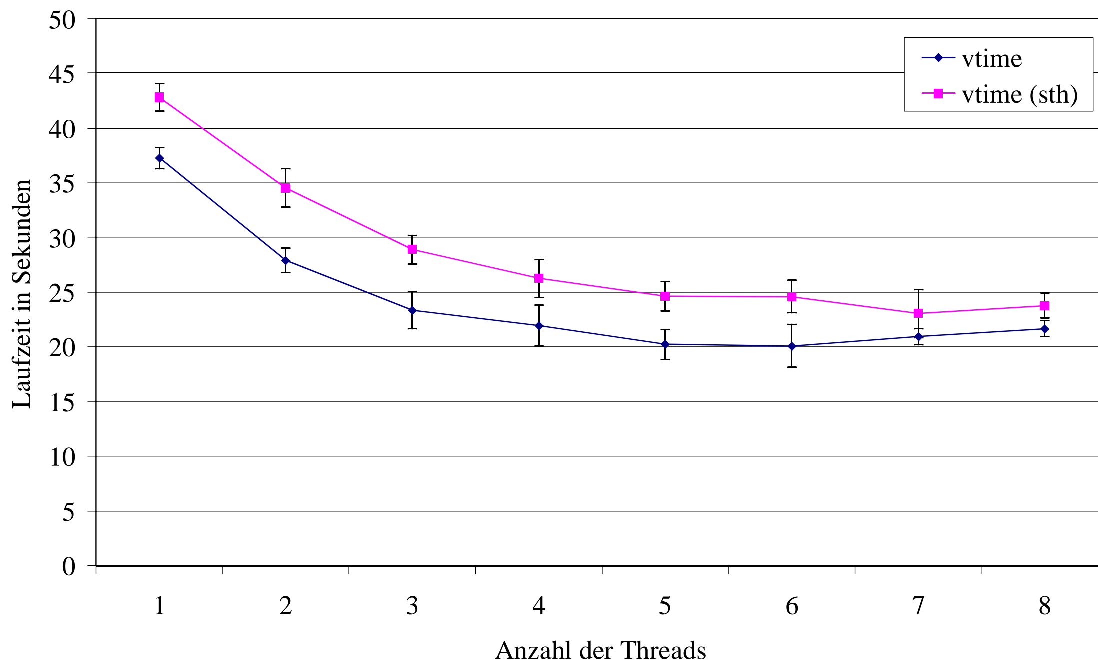

Abb. 6.9: Paralleles *du* auf vier virtuellen Prozessoren

bei mehr als vier Threads könnte durch weniger Locking-Konflikte im Filesystem, aufgrund eines stärker verteilten Zugriffsverhaltens im Dateibaum, erklärt werden. Die allgemein schlechteren Ergebnisse der *Sleeping-Threads* Version können, wie schon im Fall von *grep,* zum Teil mit einer nicht genügenden Anzahl lauffähiger User-Level-Threads erklärt werden. Ein weiterer Anteil kann in von deblockierten Threads gehaltenen Locks des Dateisystems im Server liegen, die durch die Kontrolle der Threadbibliothek über die deblockierten Threads erst verzögert freigegeben werden. Hierfür spricht auch das gelegentliche Auftreten von Verklemmungen bei einer nicht ausreichenden Anzahl von Reservethreads. Das Ausbleiben der erwarteten positiven Auswirkungen der *Sleeping-Threads* bei mehr als vier User-Level-Threads durch eine weitere Nutzung der nicht blockierenden virtuellen Prozessoren im Fall von blockierenden E/A-Aufrufen deutet auf eine bereits erreichte Auslastung des Dateisystems hin. Dies wird auch durch eine Wiederholung des Versuchs mit nur einem virtuellen Prozessor bestätigt (Abbildung 6.10). Auch hier ist die Sättigung für die *Sleeping-Threads* Version bei mehr als vier Threads erreicht, während ohne *Sleeping-Threads* Unterstützung erwartungsgemäß überhaupt kein Speedup erreicht werden kann. Denkbar wäre, daß eine verzögerte Freigabe von Locks aufgrund des *Sleeping-Threads* Mechanismus eine der Ursachen für die schon bei vier Threads erreichte Sättigung ist. Die wird jedoch durch eine Messreihe ohne *Sleeping-Threads* Unterstützung auf Basis von acht virtuellen Prozessoren widerlegt. Auch hier ist die Sättigung bei vier Threads erreicht.

Die gelegentlichen Verklemmungen deuten auf fehlende kritische Abschnitte im Dateisystemcode hin. Auf genauere Untersuchungen wurde in Hinblick auf den prototypischen Charakter der Implementation der *Sleeping-Threads* verzichtet, zumal die vorhandene Stabilität für Meßläufe ausreicht. Die Ergebnisse zeigen, in welche Richtung ein integriertes User-Level-Scheduling für E/A-intensive, systemnahe Anwendungen genutzt werden kann. Zielrichtung ist nicht

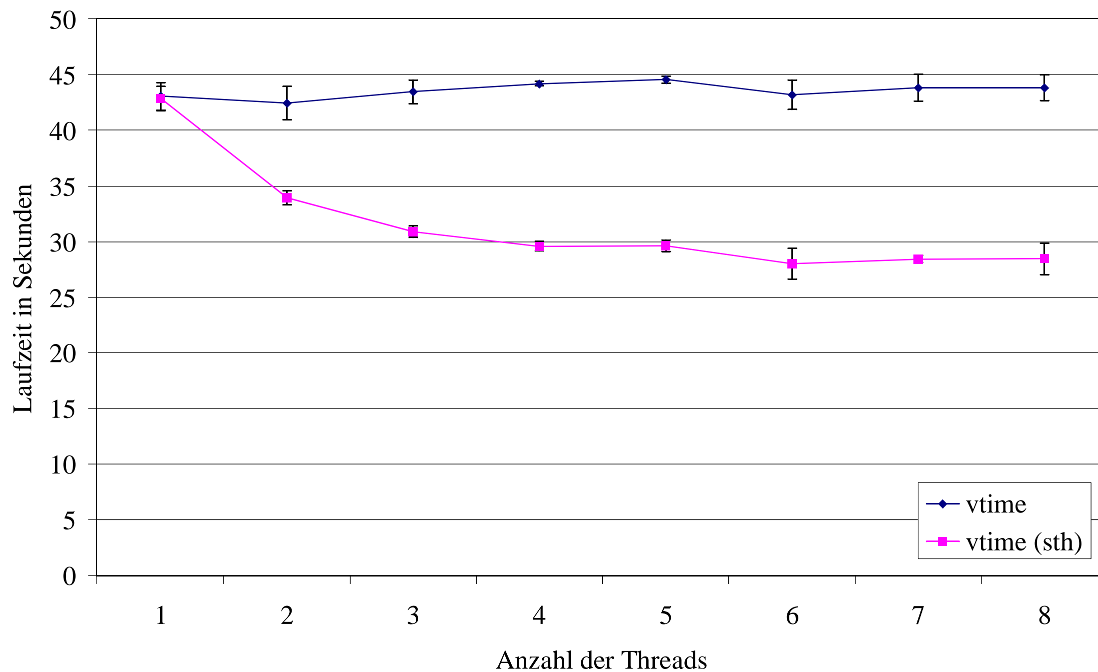

Abb. 6.10: Paralleles *du* auf einem virtuellen Prozessor

unbedingt die Nutzung vieler Prozessoren, sondern ein User-Level-Scheduling auf Basis eines oder weniger Prozessoren. Wie die Ergebnisse für ein paralleles *du* auf nur einem virtuellen Prozessor zeigen, ist es möglich, je nach Art des Integrationsmechanismus mit nur einem oder einem aktiven (im Fall der *Sleeping-Threads*) Kernel-Thread das E/A-System weitgehend auszulasten. Die Anwendung belegt nur einen Prozessor und ermöglicht beispielsweise den ungestörten Ablauf rechenintensiver Anwendungen auf anderen Prozessoren. Auch durch die Verwendung von Kernel-Threads kann dieses Ergebnis nicht ohne Modifikationen im Kernel-Scheduling erreicht werden, da dies zu einer Ausbreitung der Kernel-Threads über mehrere Prozessoren führen würde.

6.3.3 Paging / *out-of-core*-Anwendungen

In einer zweiten Klasse von Anwendungen, bezüglich der ein Aufsetzen auf den *Sleeping-Threads* Mechanismus sinnvoll ist, sind Blockierungen nicht in einer Interaktion mit Systemdiensten begründet, sondern treten asynchron durch Pagefaults auf. Ein auf *Sleeping-Threads* basierendes User-Level-Scheduling kann, analog zu einem Kernel-Scheduler, den an einem Pagefault blockierten Thread durch einen anderen ersetzen. Besonders interessant ist das Verhalten von *out-of-core*-Anwendungen. Pagefaults treten hier in besonders starkem Maß auf, da der Hauptspeicher des Systems schon allein für eine Anwendung nicht ausreicht.

Als Beispiel dient ein einfacher Glättungsalgorithmus auf einer Matrix, bei dem für die Berechnung eines neuen Wertes der Zugriff auf die direkten Nachbarelemente notwendig ist. Die Parallelisierung erfolgt über eine Partitionierung der Matrix, wobei jeder Partition ein Thread fest zugeordnet wird. Aus Gründen der Vereinfachung kann die Anzahl der Partitionen und damit

die Anzahl der Threads nur Quadratzahlen entsprechen. Ein Zugriff auf die gemeinsamen Ränder erfolgt unsynchronisiert mit der Einschränkung, daß eine Partition einer Nachbarpartition höchstens einen Iterationsschritt voraus sein darf. Die Messungen umfassen jeweils fünf Iterationen des Glättungsverfahrens. Zur Initialisierung der Matrix wurde bewußt eine andere Zuteilung zwischen Threads und Speicherbereichen gewählt, um zu zeigen, welchen Vorteil in idealen Situationen ein Scheduling auf Basis der *Sleeping-Threads* haben kann. Der Speicher ist in Schritten jeweils einer Cacheline zyklisch auf die verfügbaren Threads aufgeteilt. Dies führt in vielen Fällen dazu, daß ein Reservethread einen User-Level-Thread zur Ausführung auswählt, der die Speicherseiten referenziert, deren Einlagerung durch einen anderen Thread angestoßen wurden. Bei einer Partitionierung, wie sie für den Glättungsalgorithmus notwendig ist, wäre dies nur in wenigen Fällen möglich.

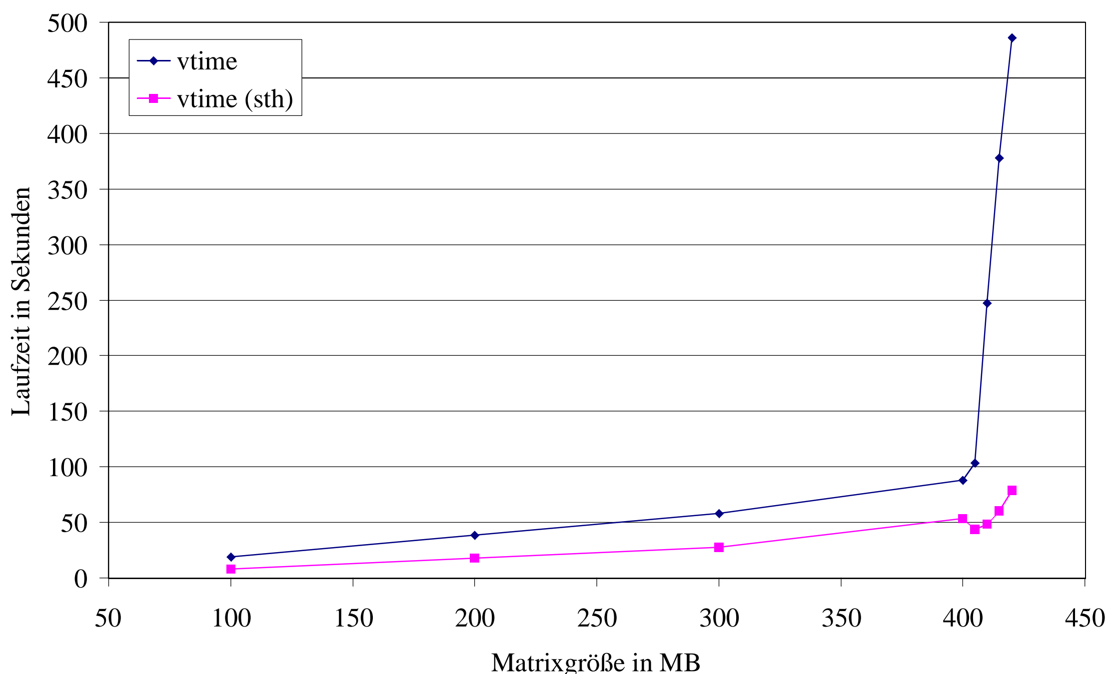

Abb. 6.11: Initialisierung mit 49 Threads bei zunehmender Matrixgröße

Der Einsatz von mehr als einem Node der SPP-Architektur wäre bei einer Anwendung dieser Klasse sinnvoll. Da aus logistischen Gründen jedoch keine einsetzbare Multinode-Version der *Sleeping-Threads* verfügbar ist (siehe Abschnitt 6.1.4), waren nur Messungen mit maximal acht Prozessoren möglich. Daher konzentrieren sich die Messungen auf die *vtime*-Strategie, die sich für Singlenode-Systeme als die günstigste erwiesen hat (siehe Abschnitt 5.4.2). Alle folgenden Messungen wurden mit acht virtuellen Prozessoren durchgeführt. Im Fall der *Sleeping-Threads* Anwendungen wurden für jeden virtuellen Prozessor vier Reservethreads generiert, so daß insgesamt 40 Kernel-Threads genutzt werden konnten. Bei Programmläufen mit 49, 64 oder mehr User-Level-Threads waren daher bei bis zu 10% der Blockierungen keine Reservethreads verfügbar und damit kein weiteres Interleaving von Paging und Berechnung mehr möglich.

Die Abbildungen 6.11 und 6.12 zeigen die Laufzeit für Initialisierung und Glättung unterschiedliche Matrixgrößen unter der *vtime* Strategie. Der drastische Anstieg bei über 400 MB (die wei-

teren Meßpunkte entsprechen 405, 410, 415 und 420 MB) zeigt, daß der Speicher des mit 512 MB ausgestatteten Systems nicht mehr ausreicht. Alle weiteren Messungen basieren auf 420 MB. Der absichtlich günstig gewählte Algorithmus für die Initialisierung zeigt, in welchem Umfang eine Ersparnis durch ein Interleaving von Paging und Berechnung auf Basis der *Sleeping-Threads* in günstigen Fällen möglich ist. Im weiteren wird nur noch die Glättungsphase betrachtet, die nicht speziell für die *Sleeping-Threads* optimiert wurde.

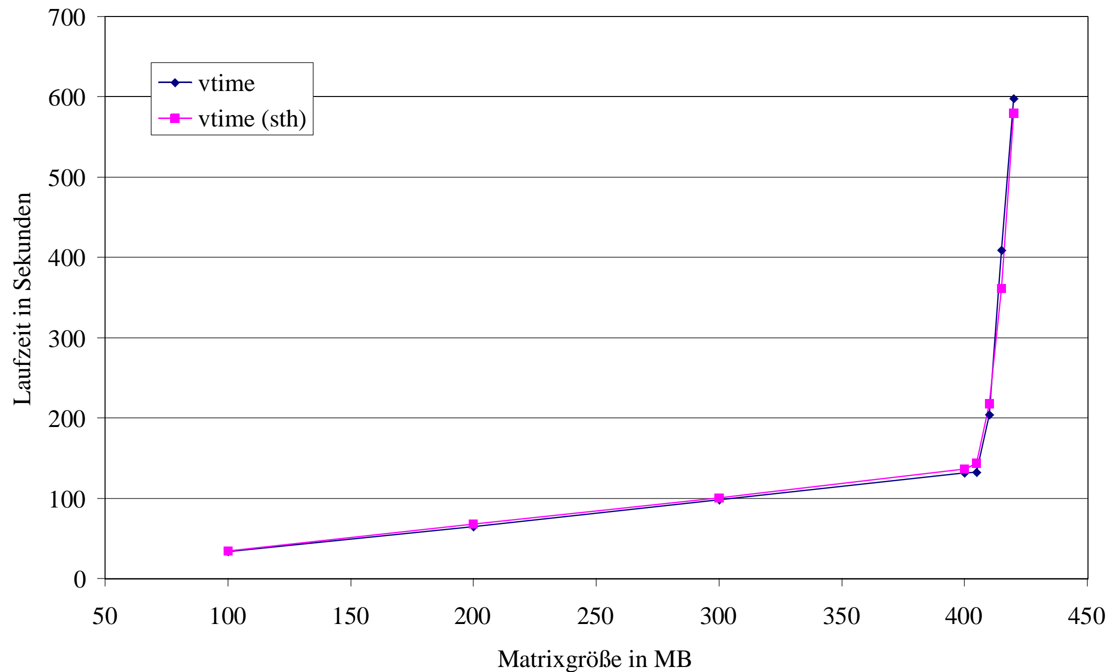

Abb. 6.12: Glättung mit 49 Threads bei zunehmender Matrixgröße

In einem weiteren Experiment werden die Strategien *noaff* und *vtime* mit und ohne *Sleeping-Threads* Unterstützung bei einer Glättung unter Einsatz einer verschieden großen Anzahl von Threads verglichen. Abbildung 6.13 stellt neben der Laufzeit die absolute Zahl an Cachemisses dar. Wie auch in Abbildung 6.14 wird die Laufzeit durch Linien und die Zahl der Cachemisses durch Balken dargestellt. Betrachtet man zunächst nur die Fälle ohne *Sleeping-Threads* Unterstützung, so fällt auf, daß die Anzahl der Cachemisses für beide Strategien gleich und konstant ist. Dies liegt darin begründet, daß der Matrixanteil eines Threads die Cachegröße übersteigt und dadurch der Cacheinhalt durch den Itererationsschritt eines Threads vollständig überschrieben wird, weil aufgrund des nichtunterbrechenden prioritätsgesteuerten Schedulings der *m-threads* ein Thread bis zur nächsten Synchronisation am Ende eines Iterationsschrittes weiterläuft. Jedes Affinitätsscheduling läuft damit ins Leere, soweit der gleiche Thread nicht erneut ausgewählt wird. Auch bei 625 Threads kann keine Verminderung der Cachemisses erreicht werden, obwohl in diesem Fall ein Thread den Cache nur zu ca. 60% auslastet. Das ist dadurch erklärbar, daß in den meisten Fällen mehr als ein anderer Thread zwischen zwei Aktivierungen eines Threads ausgeführt wird. Dadurch wird auch in diesem Fall der Cacheinhalt mit hoher Wahrscheinlichkeit vollständig überschrieben. Eine unterschiedlich gute Nutzung des Cache kann folglich nicht Ursache für die starken Laufzeitdifferenzen zwischen *noaff* und *vtime* sein.

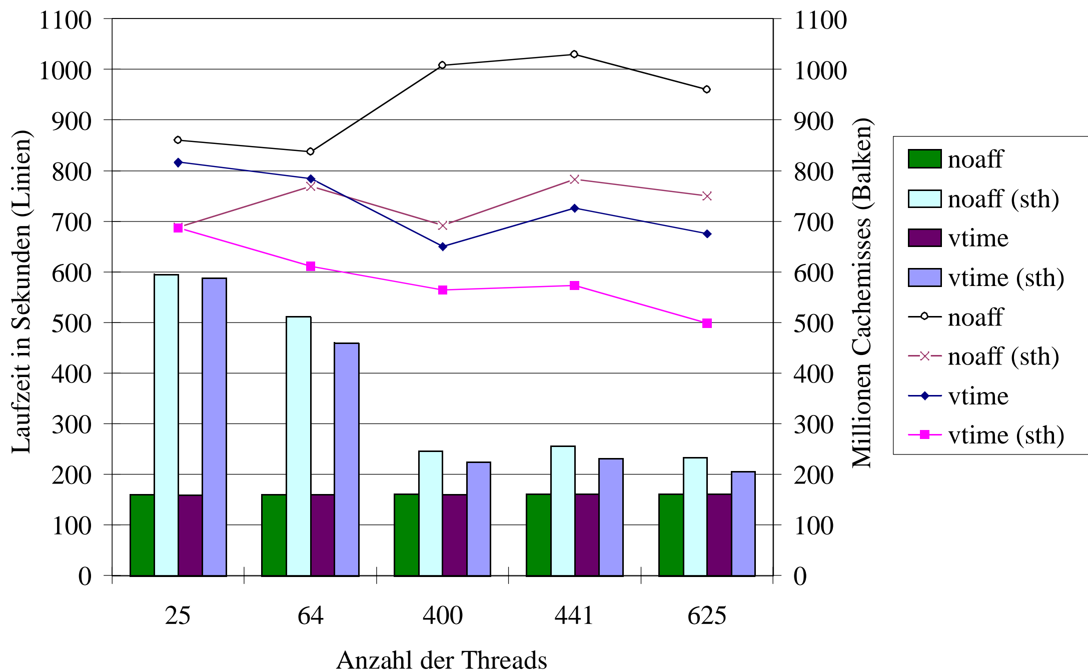

Abb. 6.13: Glättung mit 25 bis 625 Threads (*noaff* und *vtime*)

Die *vtime* Strategie optimiert das Scheduling jedoch auch bezüglich des Hauptspeichers als Cache für die virtuelle Speicherverwaltung. Die Ausführung eines Threads, dessen letzter Lauf nicht zu weit zurück liegt, erhöht die Wahrscheinlichkeit, daß die referenzierten Speicherseiten nicht bereits wieder ausgelagert sind. Die für eine gute Cacheausnutzung entwickelte *vtime* Strategie optimiert unter den geänderten Voraussetzungen einer *out-of-core* Anwendung deren Paging-Verhalten. Ausgefeiltere Strategien wie *reload*, die den Zustand der Caches über Cachemiss-Zähler und nicht nur auf Umweg über die Zeit bestimmen, können daher keine besseren Ergebnisse als *vtime* erzielen. Sie sind, wie durch weitere Messungen bestätigt werden konnte, auch nicht erheblich schlechter, weil sie ebenfalls zuletzt gelaufene Threads bevorzugen.

Der *Sleeping-Threads* Mechanismus führt über die Erkennung von Blockierungen zu einer zusätzlichen Threadumschaltung bei jedem Pagefault und dadurch zu einer deutlichen Erhöhung der Cachemisses. Andererseits kann das Affinitätsscheduling wieder seine Wirkung entfalten, da die Ausführung eines Threads nicht zum Überschreiben des gesamten Cacheinhalts führt. Dies ist an der Annäherung an die Werte für die Messläufe ohne *Sleeping-Threads* Unterstützung bei großen Threadanzahlen und damit kleineren Arbeitsmengen sowie an den generell besseren Werten der *vtime* Strategie gegenüber *noaff* zu erkennen. Der Einfluß dieser teilweise drastischen Unterschiede ist jedoch aufgrund der langen Laufzeit minimal, wie im folgenden noch belegt wird. Das Auseinanderlaufen von *vtime* und *noaff* liegt daher, wie auch schon für den Fall ohne *Sleeping-Threads*, hauptsächlich in der Optimierung des Paging-Verhaltens begründet. Entscheidend für die Bewertung des *Sleeping-Threads* Mechanismus ist jedoch der Vergleich der beiden Messungen für die *vtime* Strategie, die im folgenden für den Bereich von 25 bis 64 Threads näher betrachtet wird. Die abnehmende Tendenz in der Laufzeit, sowohl mit als auch ohne *Sleeping-Threads* Unterstützung, ist auch schon in diesem Bereich zu erkennen. Sie kann

durch bessere Optimierungschancen aufgrund der höheren Threadanzahl und der kleineren Arbeitsmenge der einzelnen Threads erklärt werden.

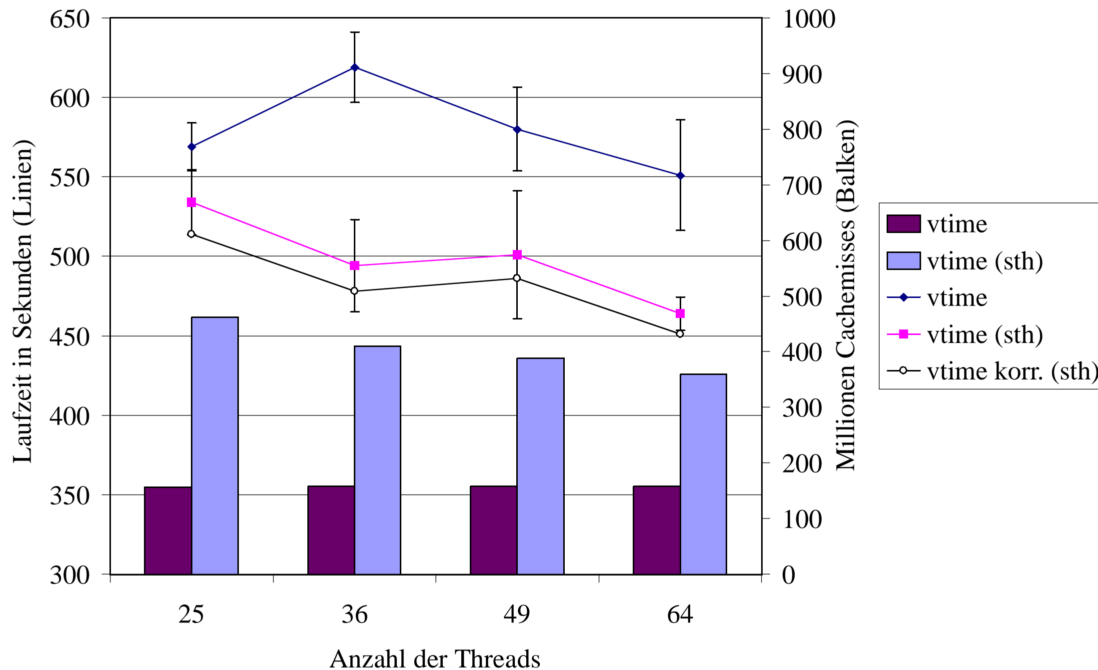

Abb. 6.14: Glättung mit 25 bis 64 Threads (modifiziertes Verfahren)

Abbildung 6.14 vergleicht die Strategie *vtime* mit und ohne *Sleeping-Threads* Unterstützung. Die Ergebnisse sind mit den vorhergehenden Messungen nicht direkt vergleichbar, da zur Verringerung der Laufzeit ein modifiziertes Glättungsverfahren mit weniger Speicherzugriffen eingesetzt wurde. Die Messreihen zeigen jedoch einen analogen Verlauf bezüglich der Cachemisses und der abnehmenden Laufzeit bei steigender Anzahl von Threads. Der Anstieg bei 36 Threads ohne *Sleeping-Threads* Unterstützung ist in mehr oder weniger ausgeprägter Form in allen Messreihen zu erkennen. Eine schlüssige Erklärung für diesen Effekt konnte jedoch nicht gefunden werden. Die dritte Kurve zeigt die Ergebnisse für den *Sleeping-Threads* Fall, korrigiert um die theoretisch mögliche Einsparung, wenn keine Erhöhung der Cachemisses gegenüber einem Lauf ohne *Sleeping-Threads* vorliegen würde. Die weitere Einsparung ist gegenüber dem positiven Effekt der *Sleeping-Threads* vergleichsweise gering und liegt im Bereich der durch die Standardabweichung repräsentierten Laufzeitschwankungen. Die Bedeutung des Cache ist folglich gegenüber den Einflüssen des Paging ebenfalls erwartungsgemäß gering. Die gemessenen Laufzeitunterschiede können trotz des starken Anstiegs der Cachemisses als Maß für die realisierbare Ausnutzung der E/A-Blockierungen des virtuellen Speichersystems durch den *Sleeping-Threads* Mechanismus interpretiert werden.

Die Laufzeitersparnis durch den *Sleeping-Threads* Mechanismus liegt, abgesehen von einer Spitze mit 20% bei 36 Threads, zwischen 6% und 16% bei steigender Threadanzahl. Für das unmodifizierte, speicherintensivere Glättungsverfahren sind Einsparungen von bis zu 33% realisierbar (siehe Abbildung 6.15). Der deutlich erkennbare Einbruch bei 64 Threads auf eine Ersparnis von nur 26% muß aufgrund der Ergebnisse anderer Messreihen als nicht repräsentativ

angesehen werden. Die hohe Standardabweichung ist aufgrund eines starken nichtdeterministischen Verhaltens durch das massive Auftreten von Pagefaults nicht überraschend. Die dennoch überschneidungsfreien Schwankungsbereiche verstärken die Aussagekraft der Messungen. Der stärkere positive Effekt der *Sleeping-Threads* bei einer höheren Threadanzahl wird durch das allgemein bessere Paging-Verhalten ermöglicht. Der Spielraum für ein Einlagern der angeforderten Speicherseiten im Hintergrund bis zum Auslasten des Systems wird dadurch größer. Die Anzahl der Threads kann auch auf direktem Weg zu diesem Effekt beitragen, da die Wahrscheinlichkeit, daß kein ablauffähiger Thread verfügbar ist und der Scheduler leerläuft, mit steigender Zahl sinkt.

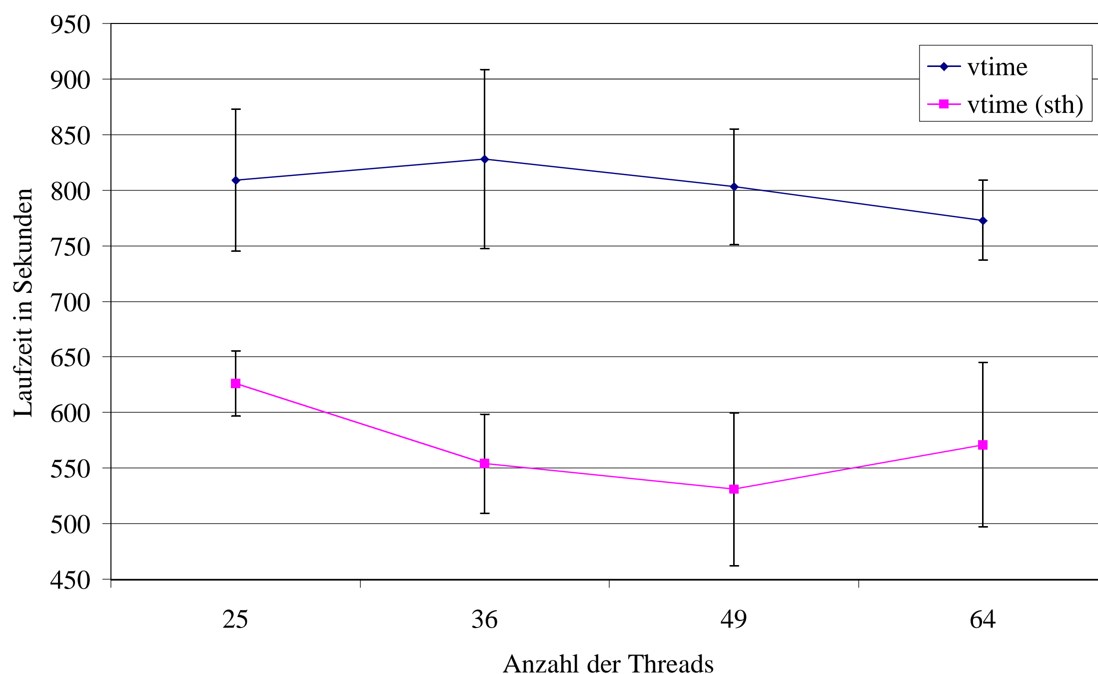

Abb. 6.15: Glättung mit 25 bis 64 Threads

Aus den durchgeführten Versuchen lassen sich zwei interessante Aussagen ableiten. Cache-optimierte Schedulingverfahren, die im allgemeinen im Rahmen eines User-Level-Schedulings realisiert sind, entfalten ihre Wirksamkeit auch in virtuellen Speichersystemen unter starker Paging-Last. Durch eine Systemintegration des User-Level-Schedulings ist, wie im Fall von Kernel-Threads, durch das Einlagern von Speicherseiten im Hintergrund ein Interleaving von Paging und Berechnung effektiv möglich. Durch die Kombination beider Verfahren entsteht eine Alternative zu bisher verbreiteten Ansätzen zur Lösung des *out-of-core* Problems für gut parallelisierbare Anwendungen. Statt effizientere Lösungen an der Speicherverwaltung vorbei zu realisieren oder die Strategien der Speicherverwaltung in Kooperation mit der Anwendung zu verbessern und um ein Pre-Paging zu erweitern, paßt sich das Scheduling an die vorhandenen Pagingstrategien an und versucht, die Abschnitte einer Anwendung auszuführen, die hauptsächlich auf eingelagerte Speicherseiten zugreifen.

Eine weitere Verbesserung ist denkbar, wenn Schedulingstrategien entwickelt werden, die - ähnlich wie zum Beispiel die *reload* Strategie bezüglich Caches - auf Basis einer Analyse des Spei-

chersystems und nicht nur in Abhängigkeit einer virtuellen Zeit optimieren. Die Voraussetzungen dafür sind zum Teil durch den *Sleeping-Threads* Mechanismus geschaffen, über den im Fall einer Blockierung aufgrund eines Pagefaults die Adresse der referenzierten Speicherseite an den User-Level-Scheduler übergeben werden könnte. Sind die Arbeitsmengen der Threads im User-Level-Scheduler bekannt, wäre zum Beispiel eine Strategie denkbar, die die Priorität eines Threads erhöht, wenn eine gerade zur Einlagerung angestoßene Speicherseite in seiner Arbeitsmenge enthalten ist. Informationen über die Arbeitsmenge eines Threads können, wie das *Follow-On* Scheduling innerhalb des ELiTE-Projekts zeigt, in effizienter Weise über eine Analyse der TLB-Misses gewonnen und in den User-Level übertragen werden [Bellos97b und Bellos98].

6.4 Zusammenfassung

Mit den *Sleeping-Threads* wurde eine Erweiterung des Kernel-Schedulers entwickelt, die über eine Integration von User-Level- und Kernel-Threads eine Lösung für die bekannten Probleme des User-Level-Schedulings ermöglicht. Gegenüber älteren Arbeiten in diesem Bereich grenzen sich die *Sleeping-Threads* über einige Details bezüglich Aufbau und Effizienz hauptsächlich jedoch über eine gegenüber dem Kernel-Scheduling orthogonale Realisierung und die Intention des Projekts ab.

Die Betriebssystemerweiterungen des *Sleeping-Threads* Mechanismus setzen an der Blockierung und Deblockierung der Kernel-Threads an und lassen das eigentliche Kernel-Scheduling unverändert. Dadurch sind die *Sleeping-Threads* mit einer Vielzahl von Schedulingstrategien kombinierbar und erlauben zum Beispiel die einfache Integration mit einem CPU-Scheduling zur Realisierung einer dynamischen Partitionierung. Der zweite Vorteil der orthogonalen Konzeption liegt in einer prinzipiell einfachen Implementierung. Die Gründe für eine letztlich sehr komplexe und für einen Produktionsbetrieb ungeeignete Implementierung liegen in der Struktur von UNIX und insbesondere MACH-basierten Implementierungen und betriebssysteminternen Annahmen, die eine Kontrolle des User-Level über Fortsetzung eines blockierten Kernel-Threads nicht ohne massive Probleme erlauben. Systemerweiterungen, die wie bei *Sleeping-Threads* die Kontrolle über systeminterne Abläufe teilweise in den User-Level übertragen, werden daher oft nur in den Entwicklungsabteilungen selbst möglich sein und weitreichende Modifikationen des Betriebssystems erfordern.

Die prototypische Implementierung der *Sleeping-Threads* erlaubt jedoch eine Untersuchung der Möglichkeiten eines systemintegrierten User-Level-Schedulings im Rahmen des breit angelegten Scheduling-Projekts ELiTE. Die Ergebnisse zeigen den potentiellen Nutzen eines nicht unbedingt auf den *Sleeping-Threads* oder dem Betriebssystem UNIX basierten integrierten User-Level-Schedulings. Ein auf den *Sleeping-Threads* basierendes User-Level-Scheduling erlaubt eine Abstraktion des virtuellen Prozessors, die der Prozessorabstraktion des Betriebssystemkerns sehr nahe kommt. Damit sind User-Level-Threadbibliotheken realisierbar, die sich analog zu Kernel-Implementierungen verhalten und den Erwartungen an ein Threadsystem entspre-

chen. Dies hat nicht unerhebliche positive Auswirkungen bezüglich der Gefahr von Verklemmungen und einer ergonomischen Programmierung.

In Kombination mit der *m-threads* Bibliothek konnte gezeigt werden, das ein integriertes User-Level-Scheduling eine gute Basis für E/A-intensive Systemanwendungen ist. Die Nutzung von Wartezeiten auf den Abschluß von E/A-Operationen über den Weg einer Parallelisierung wird möglich, ohne andere Anwendungen durch ein Ausbreiten über mehrere Prozessoren zu behindern. In Zusammenhang mit den cache-optimierten Schedulingstrategien der *m-threads* wurde ein neuer Ansatz für *out-of-core*-Anwendungen entwickelt, der nicht auf Modifikationen des virtuellen Speichersystems basiert, sondern durch ein geeignetes Scheduling den Ablauf der parallelen Anwendungen an das Paging-Verhalten des Systems anpaßt.

Mit den Grundlagen für eine im Rahmen dieser Arbeit nicht implementierte Kombination der *Sleeping-Threads* mit einem unterbrechenden, auf Zeitscheiben basierenden User-Level-Scheduling befaßt sich das folgende Kapitel.

7 Unterbrechungs-mechanismen

Unterbrechendes Scheduling und Zeitscheibenverfahren stellen, neben den auf schlechter Systemintegration beruhenden Problemen, einen zweiten Bereich dar, in dem die Möglichkeiten eines Kernel-Schedulings nicht auf User-Level-Scheduler übertragen werden können. Ursache hierfür sind die ungeeigneten Unterbrechungsmechanismen, die von den Betriebssystemen angeboten werden. Sie sind für ein User-Level-Scheduling nicht effizient genug und orientieren sich oft an veralteten Betriebssystemkonzepten, was in modernen thread-basierten Systemen zu weiteren Problemen führt. Im Rahmen des ELiTE-Projekts wurden die Unterbrechungskonzepte verschiedener Betriebssysteme analysiert und - basierend auf den Ergebnissen - Anforderungen an Unterbrechungsmechanismen zur Implementierung unterbrechender User-Level-Scheduler entwickelt. Für SPP-UX wurde ein Mechanismus implementiert, der diese Anforderungen umsetzt [Reder96].

Im folgenden sind mit dem Begriff *Threads* Kernel-Threads gemeint. Die Unterbrechung von Kernel-Threads bildet über eine damit realisierte Unterbrechung von virtuellen Prozessoren die Grundlage für ein unterbrechendes User-Level-Scheduling.

Nach einer kurzen Betrachtung der Standardmechanismen, wobei der Schwerpunkt auf MACH-basierte UNIX-Systeme gelegt wird, werden die entwickelten Anforderungen erläutert. Anschließend wird der implementierte Mechanismus vorgestellt und auf mögliche Anwendungen eingegangen.

7.1 Standardmechanismen

Die prinzipiellen Probleme, die eine Realisierung von unterbrechenden bzw. zeitscheibenbasierten User-Level-Scheduling verhindern oder zumindest komplizieren, wurden bereits in Abschnitt 4.4.2.4 erörtert. Das Hauptproblem liegt, zumindest bei UNIX-Systemen, in einer Orientierung heutiger Unterbrechungsmechanismen an dem veralteten prozeßorientierten Konzept der Signale, das nur unzureichend an eine Nutzung durch Anwendungen mit mehreren Threads angepaßt wurde. Der angepaßte Signalmechanismus unterscheidet bezüglich der Unterstützung

von Threads zwischen synchronen und asynchronen Signalen. Asynchrone Signale können nicht einem konkreten Thread zugestellt werden, sondern werden an den Prozeß gesendet. Die Auswahl des Threads, der letztendlich zur Signalbearbeitung unterbrochen wird, kann auf unterschiedliche Art festgelegt werden oder wird durch das Betriebssystem bestimmt. Das ALARM-Signal, welches für ein zeitscheibenbasiertes Scheduling genutzt werden könnte, gehört dieser Klasse an. Das timer- oder programmgesteuerte Unterbrechen bestimmter virtueller Prozessoren (Kernel-Threads) ist daher nur mit großem Aufwand und wenig effizient über Signale möglich. Neben einem threadspezifischen Unterbrechungsmechanismus werden zusätzlich threadspezifische Timer benötigt, die ebenfalls von kaum einem Betriebssystem angeboten werden.

Ein zweites bisher nur kurz angesprochenes Problem liegt in der Effizienz des Signalmechanismus, insbesondere unter MACH oder anderen mikrokern-basierten Systemen. Tabelle 7.1 gibt die Zeit zwischen dem Erkennen eines abgelaufenen Timers durch den Betriebssystemkern und der Ausführung des Signalhandlers durch einen Thread im User-Level für unterschiedliche Systeme an [Reder96]. Verglichen mit einer Threadumschaltung im User-Level (10 µsec) ist das Ergebnis des MACH-Systems SPP-UX nicht diskutabel. Der für Real-Time MACH angegebene Wert muß mit den Ergebnissen der monolithischen Systeme verglichen werden, da hier nicht ein - im Zusammenspiel mit dem Betriebssystemserver - implementierter Signalmechanismus gemessen wurde, sondern ein Unterbrechungsmechanismus des Mikrokerns selbst. Auch die Ergebnisse der monolithischen Systeme sind, verglichen mit der Threadumschaltung, zu hoch. Da sie in der Größenordnung einer Kernel-Threadumschaltung liegen, macht ein User-Level-Scheduling auf dieser Basis zumindest bezüglich der Effizienz keinen Sinn. Vergleicht man die SunOS mit dem Nachfolgesystem Solaris, wird deutlich, daß auch monolithische Systeme immer komplexer und daher in vielen Bereichen langsamer werden.

Betriebssystem (Rechner, Taktrate)	µsec	Taktzyklen
SPP-UX (SPP 1000, 100 MHz)	1049	104900
RT-MACH (Gateway 2000, 66 MHz)	162	10692
HP-UX 9.01 (HP715, 33 MHz)	221	7326
SunOS 4.1.3 (Sparcstation 2, 40 MHz)	174	6992
Solaris 2.5 (Sparcstation 10, 36 MHZ)	385	13860

Tab. 7.1: Signalzustellungszeiten nach erkanntem Timerablauf

7.2 Anforderungen an einen Unterbrechungsmechanismus

Die Ineffizienz des Signalmechanismus in MACH-basierten Systemen beruht hauptsächlich auf einem komplexen Zusammenspiel zwischen Mikrokern, Server, Emulator und Anwendungsprozeß. Die Beteiligung von Server und Emulator ist im wesentlichen aus zwei Gründen notwendig: Zum einen ist dies die Bindung des Signalmechanismus an die Abstraktion des Prozesses, die durch Server und Emulator realisiert und im Mikrokern selbst unbekannt ist. Im Fall eines threadspezifischen Unterbrechungsmechanismus entfällt dieser Grund. Der zweite Grund

liegt in der Möglichkeit, über den Signalmechanismus Systemaufrufe zu unterbrechen, was unter Umgehung von Server und Emulator nicht möglich wäre. In Zusammenhang mit einem unterbrechenden User-Level-Scheduling ist dies jedoch nicht notwendig, und im Gegensatz zu einer absichtlichen Unterbrechung eines zu lang blockierenden Systemaufrufs im Rahmen des Schedulings auch nicht sinnvoll. Im Zusammenspiel mit den *Sleeping-Threads* tritt dieses Problem nicht einmal auf, da anstelle des blockierten Threads ein Reservethread aktiv ist.

Neben dieser Einschränkung gegenüber dem Signalmechanismus, die insbesondere auf MACH-basierten Systemen eine erheblich effizientere Realisierung erlaubt, müssen Anforderungen an den Unterbrechungsmechanismus selbst und die benötigten Timer gestellt werden:

- Jeder Thread muß über einen eigenen, unabhängig setzbaren Timer mit einer Auflösung zumindest in der Größenordnung der Zeitscheiben des Kernel-Schedulers verfügen.
- Verschiedene Zeitbasen sollten unterstützt werden (Realzeit, Rechenzeit im Benutzermodus, gesamte Rechenzeit). Für ein User-Level-Scheduling ist zumindest ein Realzeit-Timer notwendig.
- Ein Timer muß zumindest durch den Thread selbst gesetzt und gelöscht werden können.
- Beim Ablauf eines Timers muß gezielt der assoziierte Thread unterbrochen und an einer frei definierbaren Adresse des Benutzeradreßraums fortgesetzt werden.
- Nur im Benutzermodus ablaufende Threads dürfen unterbrochen werden. Eine unterbrechung im Emulator ist ebenfalls nicht erlaubt. Dies würde zu den gleichen Problemen wie durch den *Sleeping-Threads* Mechanismus führen.
- Ist eine Unterbrechung wegen oben genannter Einschränkungen nicht möglich, muß sie vermerkt und nachgeholt werden, sobald der Thread den geschützten Bereich verläßt.

Um die programmgesteuerte Unterbrechung eines Threads durch einen anderen des gleichen Prozesses zu ermöglichen, sind keine weiteren Mechanismen notwendig, wenn Timer auch durch andere Threads des gleichen Prozesses manipulierbar sind. Die Unterbrechung eines anderen Threads kann durch Setzen eines sofort ablaufenden Timers realisiert werden.

7.3 ELiTE Bounce (EBNC)

ELiTE Bounce koppelt threadspezifische Timer mit einem Unterbrechungsmechanismus, der auf einer Interaktion zwischen Benutzerprozeß und Mikrokern unter Umgehung von Server und Emulator[1] basiert. Alle drei oben genannten Zeitbasen werden unterstützt, wobei das Aufsetzen eines Timers sowohl für eine einmalige als auch für zyklische Unterbrechungen möglich ist. Pro Thread ist das Setzen mehrerer unabhängiger Timer möglich, für die unterschiedliche Behandlungsroutinen installiert werden können.

Die Implementation der Timer basiert auf dem alle 10 msec ausgelösten Clock-Interrupt des Mikrokerns[2]. Der Ablauf eines oder mehrerer Timer des aktuell laufenden Threads führt zu einer

1. Der Emulator ist lediglich in die Realisierung der Systemaufrufe an den Mikrokern zum Setzen und Löschen von Timern involviert, jedoch nicht in die Zustellung der Unterbrechungen.

Unterbrechung des Threads, der Ablauf eines Realzeit-Timers eines aktuell nicht laufenden Threads nur zu einer Markierung und gegebenenfalls zu einer Threadumschaltung, um eine Behandlung möglichst schnell durchführen zu können. Für Threads, die zum Zeitpunkt des Timerablaufs im Kontext des Mikrokerns oder Emulators laufen, wird der Aufruf des Unterbrechungshandlers verzögert, bis sie Mikrokern bzw. Emulator verlassen. Der Aufruf des Unterbrechungshandlers erfolgt in zwei Stufen. Zunächst erzwingt der Mikrokern die Fortsetzung des unterbrochenen Threads in einem generischen Handler der EBNC-Bibliothek und übermittelt unter Verwendung des Stacks die Identifikationen aller abgelaufenen Timer. Der Stack wird derart manipuliert, daß der Thread nach Beendigung des generischen Handlers an der unterbrochenen Stelle fortgesetzt wird. Der generische Handler ruft nacheinander die durch die Anwendung installierten Handler für die abgelaufenen Timer auf, wobei für jeden Handler nur ein Aufruf erfolgt und die Identifikationen der abgelaufenen Timer in Form einer verketteten Liste übergeben werden.

Die programmgesteuerte Unterbrechung durch einen anderen Thread der gleichen Anwendung kann über das Setzen eines Realzeit-Timers mit minimaler Laufzeit erreicht werden. Die Unterbrechung erfolgt nach ca. 10 msec.

Die EBNC-Timer sind mit 79 µsec zwischen Erkennung eines Ablaufs und der Ausführung des Unterbrechungshandlers gegenüber den 1049 µsec des auf Signalen basierenden SPP-UX Standardmechanismus erheblich effizienter. Dies ist immer noch um den Faktor acht langsamer als eine Threadumschaltung im User-Level und führt damit im Rahmen eines zeitscheibenbasierten User-Level-Schedulings zu einem erheblichen zusätzlichen Overhead. Gegenüber einem Kernel-Scheduling ist ein unterbrechendes User-Level-Scheduling auf Basis der EBNC-Timer jedoch um das ca. dreifache effizienter.

7.4 Anwendungen

Neben der deutlich besseren Effizienz und einer Unterstützung von Threads ermöglichen die EBNC-Timer gegenüber dem Standardmechanismus eine sehr viel flexiblere Anwendung durch die Unterstützung mehrerer unabhängiger Timer. Eine Anwendung kann auf einfache Weise über unterschiedliche Unterbrechungshandler oder durch die Unterscheidung verschiedener Timer innerhalb eines Handlers zwischen unterschiedlichen Unterbrechungsursachen differenzieren.

7.4.1 User-Level-Scheduling

EBNC-Timer können im Zusammenhang mit einem User-Level-Scheduling für unterschiedliche Zwecke genutzt werden. Sie ermöglichen über eine programmgesteuerte Unterbrechung von virtuellen Prozessoren ein unterbrechendes und damit flexibleres Scheduling. Es können prioritätengesteuerte Strategien implementiert werden, die niedrig priorisierte Threads unterbrechen, wenn für ablauffähige höher priorisierte Threads kein freier virtueller Prozessor ver-

2. Dadurch ist die maximale Auflösung der EBNC-Timer ebenfalls auf 10 msec festgelegt.

fügbar ist. Die zweite Variante besteht in einem zeitscheibengesteuerten Scheduling in einer gegenüber dem Kernel-Scheduling sehr viel flexibleren Form. Beispielsweise sind unterschiedlich lange Zeitscheiben für verschiedene Threads möglich.

Die Unterbrechung eines Threads wird über einen Unterbrechungshandler realisiert, der eine Schedulingroutine zur freiwilligen Prozessoraufgabe aufruft und somit die Möglichkeit für eine erneute Schedulingentscheidung eröffnet. In einem Zeitscheibenverfahren führt dies zu einer Umschaltung auf den nächsten Thread in der Runqueue des virtuellen Prozessors, in einem prioritätengesteuerten Scheduling zu einer Umschaltung auf einen höher priorisierten Thread, der aufgrund der Unterbrechung mit einer hohen Wahrscheinlichkeit zur Ausführung ansteht. Führt die Unterbrechung nicht zu einem Threadwechsel oder wird der unterbrochene Thread später fortgesetzt, setzt er nach Beenden des Unterbrechungshandlers den Programmablauf an der unterbrochenen Stelle fort.

Ein unterbrechendes oder zeitscheibengesteuertes User-Level-Scheduling wurde bisher innerhalb des ELiTE-Projekts nicht realisiert. Die Threadbibliothek *m-threads* ist sowohl aufgrund ihrer cache-optimierenden Strategien als auch in ihrem strukturellen Aufbau für solche Schedulingstrategien nicht geeignet. Die Entwicklung oder Portierung anderer Threadbibliotheken war im Rahmen des Projekts nicht mehr realisierbar.

7.5 Zusammenfassung

Mit den EBNC-Timern wurde ein Unterbrechungsmechanismus entwickelt, der im Gegensatz zu dem Signalmechanismus auch in komplexen Betriebssystemen auf NUMA-Architekturen effizient abläuft. Dies wurde durch eine Orientierung an Threads anstelle von Prozessen, die in einem threadbasierten System ohnehin sinnvoller ist, und durch den Verzicht auf in vielen Anwendungsfällen nicht benötigte Funktionalität erreicht. Der neue Mechanismus ermöglicht durch seine Effizienz und Flexibilität Anwendungen, die bisher in UNIX-basierten Multiprozessorsystemen nicht möglich waren.

EBNC-Timer unterstützen - im Gegensatz zu dem mit dem Signalmechanismus kombinierten Standard-Timer von UNIX-Systemen - threadspezifische Timer sowie die gezielte Unterbrechung bestimmter Threads. Sie sind daher als Basis für ein unterbrechendes oder zeitscheibengesteuertes User-Level-Scheduling geeignet. Solche Schedulingverfahren wurden im Rahmen des ELiTE-Projekts bisher jedoch nicht realisiert.

8 Zusammenfassung

Der Einfluß von Betriebssystemen auf die Fortentwicklung in anderen Softwarebereichen beruht nicht nur auf neuen Konzepten, die neue Möglichkeiten für die Anwendungssoftware eröffnen. Ebenso bedeutend sind die Auswirkungen von Defiziten, die in der langsamen Weiterentwicklung sehr komplexer und inflexibler Betriebssystemarchitekturen begründet sind. Ein Aspekt dieses Problembereichs liegt in der Forderung nach effizienten und flexibel an die Anforderungen einzelner Anwendungen angepaßten Systemdiensten. Heutige kommerzielle Betriebssysteme können diese Anforderungen aufgrund ihrer an einer generellen Einsetzbarkeit orientierten und kaum erweiterbaren Systemdienste nicht erfüllen. In vielen Bereichen führte dies zu Realisierungen außerhalb des Betriebssystems unter Hinnahme der Probleme einer fehlenden Integration oder zu aufwendigen Speziallösungen innerhalb eines Betriebssystems für wenige Anwendungen.

Im Bereich des Threadschedulings führte die Forderung nach Effizienz und flexibler Anpaßbarkeit der Schedulingstrategien zur Verbreitung der User-Level-Threads. Deren ungenügende Systemintegration führt jedoch zu funktionalen Einschränkungen gegenüber einer Realisierung im Betriebssystemkern und zu Problemen, die von einer Relativierung der erwarteten Leistungssteigerungen bis zu massiven Parallelitätseinbrüchen oder Verklemmungen einer Anwendung reichen können.

Mit den *Sleeping-Threads* wurde ein neuer Betriebssystemmechanismus entwickelt, der eine Integration des User-Level-Schedulings ermöglicht. Die *Sleeping-Threads* erlauben die Abstraktion eines virtuellen Prozessors als Basis für ein User-Level-Scheduling, die sich aus Sicht einer Anwendung analog zu der Prozessorabstraktion des Betriebssystems verhält. Dadurch können die bekannten Probleme der User-Level-Threads vermieden werden. Von vergleichbaren Arbeiten grenzt sich der vorgestellte Ansatz vor allem durch einen orthogonal zum Kernel-Scheduler angelegten Aufbau ab, der eine Kombination mit verschiedenen Schedulingmechanismen im Betriebssystemkern erlaubt, und ebenso durch eine konsequente Unterstützung von mikrokernbasierten Betriebssystemarchitekturen. Gerade diese Betriebssystemarchitekturen werden auf

leistungsfähigen NUMA-Architekturen eingesetzt, die eine interessante Zielgruppe für ein effizientes und integriertes User-Level-Scheduling darstellen.

Erstmals wurde ein integriertes User-Level-Scheduling nicht nur unter Verwendung von Standardstrategien mit einem nicht integrierten Pendant verglichen, sondern als Teil eines umfassenden Scheduling-Projekts - dem ELiTE-Projekt (**E**rlangen **Li**ghtweight **T**hread **E**nvironment) an der Universität Erlangen-Nürnberg - aufgefaßt und untersucht. Diese Untersuchungen zeigen unter anderem, daß ein integriertes User-Level-Scheduling zusammen mit hoch optimierten Schedulingstrategien, die ursprünglich über eine bessere Nutzung von cache-basierten Rechnerarchitekturen motiviert waren, als Basis zur effizienten Berechnung von *out-of-core*-Problemen eingesetzt werden kann. Dieser Ansatz nutzt die zentrale Bedeutung des Schedulings auf modernen Multiprozessorsystemen, um ein Problem aus einem anderen Bereich der Betriebssystemforschung zu lösen.

Mit einem zweiten Betriebssystemmechanismus, den EBNC-Timern, wurde die Grundlage für unterbrechende und zeitscheibengesteuerte Schedulingstrategien im User-Level gelegt, die aufgrund ungeeigneter oder insbesondere in komplexen Systemen ineffizienter Unterbrechungsmechanismen in vielen Systemen bisher nicht möglich waren. Im Unterschied zu den Standardunterbrechungsmechanismen vieler Betriebssysteme unterstützen die EBNC-Timer das gezielte Unterbrechen bestimmter Threads und verzichten aus Effizienzgründen auf Funktionalität, die für ein unterbrechendes User-Level-Scheduling nicht notwendig ist. Damit ist eine Klasse von Schedulingstrategien im User-Level realisierbar, die bisher dem Kernel-Scheduling vorbehalten war.

Neben dem integrierten User-Level-Scheduling hat sich eine Reihe von Alternativen zu Kernel- und User-Level-Threads in ihrer heutigen Ausprägung entwickelt. Die beiden interessantesten Ansätze liegen in den Schedulingmechanismen erweiterbarer Betriebssystemarchitekturen und in der direkten Unterstützung von Threads durch Hardwarekomponenten. Erweiterbare Betriebssysteme haben den Status von Forschungsprojekten noch nicht überwunden und *multithreaded* Prozessoren stehen erst am Anfang einer kommerziellen Weiterentwicklung. Ihr Einfluß auf kommende Entwicklungen im Bereich des Schedulings ist daher heute noch schwer einschätzbar. Die Ergebnisse der vorliegenden Arbeit sind in zweierlei Hinsicht von Bedeutung für die Zukunft des Threadschedulings: Sie zeigen, zum einen welche Möglichkeiten in einem durch Software realisierten Scheduling, das sich von den heute üblichen Konzepten löst, und damit in den Schedulingkonzepten neuer Betriebssystemarchitekturen liegen können. Zum anderen untermauern sie die Notwendigkeit einer - wie nachgewiesen wurde - möglichen Integration moderner Schedulingverfahren in heutige kommerzielle Betriebssysteme, da ansonsten die Leistungsfähigkeit moderner Multiprozessorsysteme über eine langwierige Etablierungsphase neuer Verfahren in Soft- oder Hardware hinweg nicht in einem adäquaten Umfang ausgenutzt werden kann.

Literatur

ABBG+86 M. J. Accetta, R. V. Baron, W. Bolosky, D. B. Golub, R. F. Rashid, A. Tevenian M.W. Young: “Mach: A New Kernel Foundation for UNIX Development”; Proceedings of the 1986 Summer USENIX Conference, Seiten 93-112; Juni 1986

ABLL92 T. E. Anderson, B. N. Bershad, E. D. Lazowska, H. M. Levy: “Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism”; ACM Transactions on Computer Systems, Volume 10, Number 1, Seiten 53-79; Februar 1992

ACCK+90 R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield, B. Smith: “The Tera Computer System”; Tera Computer Company, Seattle; Juni 1990

AKKM+95 G. Alverson, S. Kahan, R. Korry, C. McCann, B. Smith: “Scheduling on the Tera MTA”; Proceedings of the IPPS’95 Workschop on Job Scheduling Strategies for Parallel Processing, Seiten 19-44; April 1995

Bellos95 F. Bellosa: “Memory-Conscious Scheduling and Processor Allocation on NUMA Architectures”; Interner Bericht TR-I4-94-06, Institut für mathematische Maschinen und Datenverarbeitung IV, Universität Erlangen-Nürnberg; Mai 1995

Bellos97a F. Bellosa: “Memory Access - The Third Dimension of Scheduling”; Interner Bericht TR-I4-97-01, Institut für mathematische Maschinen und Datenverarbeitung IV, Universität Erlangen-Nürnberg; Januar 1997

Bellos97b F. Bellosa: “Follow-On Scheduling: Using TLB Information to Reduce Cache Misses”; 16th ACM Symposium on Operating System Principles, Work-in-Progress and Poster Session; 1997

Bellos98 F. Bellosa: “The Three Dimensions of Scheduling”; Eingereichte Dissertationsschrift, Institut für mathematische Maschinen und Datenverarbeitung IV, Universität Erlangen-Nürnberg; Juni 1998

BelSte96 F. Bellosa, M. Steckermeier: “The Performance Implications of Locality Information Usage in Shared Memory Multiprozessors”; Journal of Parallel and Distributed Computing, Special Issue on Multithreading for Multiprocessors, Volume 37, Number 1, Seiten 113-121; August 1996

BiGoWa94a G. Bier, K. Goss, T. Watson: “RPC (INKS) Design Document”; Convex Corporation, Richardson; August 1994

BiGoWa94b G. Bier, K. Goss, T. Watson: "Multinode RPC (INKS) Design Document"; Convex Corporation, Richardson; August 1994

Black89 D. J. Black: "The Mach cpu_server: An Implementation of Processor Allocation"; Carnegie Mellon University; August 1989

Black90a D. J. Black: "Scheduling Support for Concurrency and Parallelism in the Mach Operating System"; Technical Report CMU-CS-90-125, Carnegie Mellon University; April 1990

Black90b D. J. Black: "Scheduling and Resource Management Techniques for Multiprocessors"; Dissertationsschrift, Technical Report CMU-CS-90-152, Carnegie Mellon University; Juli 1990

BLRC94a B. N. Bershad, D. Lee, T. H. Romer, J. B. Chen: "Avoiding Conflict Misses Dynamically in Large Direct-Mapped Caches"; Proceedings of the 6th Symposium on Architectural Support for Programming Languages and Operating Systems (ASPLOS VI), Seiten 158-170; Oktober 1994

BLRC94b B. N. Bershad, D. Lee, T. H. Romer, J. B. Chen: "Dynamic Page Mapping Policies for Cache Conflict Resolution on Standard Hardware"; Proceedings of the 1st Symposiun on Operating Systems Design and Implementation (OSDI'94), Seiten 255-266; 1994

BlWaTi98 J. M. Blum, T. M. Warschko, W. F. Tichy: " PULC: ParaStation User-Level Communication. Design and Overviev"; PC-NOW Workshop of the 12th International Parallel Processing Symposium and the 9th Symposium on Parallel and Distributed Processing, Orlando, Florida; April 1998

BMVL93 P. Barton-Davis, D. McMamee, R. Vaswani, E. D. Lazowska: "Adding Schedular Activations to MACH 3.0"; Proceedings of the USENIX Mach III Symposium, Seiten 119-136; April 1993

Brecht94 T. B. Brecht: "Multiprogrammed Parallel Application Scheduling in NUMA Multiprocessors"; Dissertationsschrift, Technical Report CSRI-303, Computer Systems Research Institute, University of Toronto; Juni 1994

BSPS+95 B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, G. Chambers, S. Eggers: "Extensibility, Safty and Performance in the SPIN Operating System"; Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP'95), Seiten 267-284; 1995

CaFeLi94 P. Cao, E. W. Felten, K. Li: "Application-Controlled File Caching Policies"; Proceedings of the Usenix Summer 94 Technical Conference, Seiten 171-182; 1994

CDGJ+92 L. Cardellei, J. Donahue, L. Glassman, M. Jordan, B. Kalsow, G. Nelson: "Modula-3 Language Definition"; ACM SIGPLAN Notices, Volume 27, Number 8, Seiten 15-42; August 1992

Convex93 “Convex Exemplar Architecture”, First Edition; Convex Press, Richardson; November 1993

Convex94 “Convex Exemplar Programming Guide”, First Edition; Convex Press, Richardson; Oktober 1994

Dion96 D. Dion: “A User-Level Unix Server for the SPIN Operating System”; Technical Report TR-UW-CSE-96-11-01, Department of Computer Science and Engineering, University of Washington, Seattle; Oktober 1996

EnKaTo95 D. R. Engler, M. F. Kaashoek, J. O’Toole Jr.: “Exokernel: An Operating System Architecture for Application-Level Resource Management”; Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP’95), Seiten 251-266; 1995

FeiRud90 D. Feitelson, L. Rudolph: “Distributed Hierarchical Control of Parallel Processing”; IEEE Computer, Volume 23, Number 5, Seiten 65-77; 1990

FisDal95 S. Fiske, W. J. Dally: “Thread Prioritization: A Thread Scheduling Mechanism for Multiple-Context Parallel Processors”; Proceedings of the First International Symposium on HPCA; Januar 1995

GDFR90 D. Golub, R. Dean, A. Forin, R. Rashid: “Unix as an Application Program”; Proceedings of the 1990 Summer USENIX Conference, Seiten 87-95, Juni 1990

GooCox93 B. Goodheart, J. Cox: “The Magic Garden Explained: The Internals of UNIX System V Release 4”; Prentice Hall; 1993

Hauth95 A. Hauth: “Vergleichende Untersuchung von Threadbibliotheken auf ihre Eignung für numerische und systemnahe Anwendungen“; Studienarbeit 4/95, Institut für mathematische Maschinen und Datenverarbeitung IV, Universität Erlangen-Nürnberg; April 1995

Helgaa94 B. Helgaas: “Convex Camelot Mach Microkernel Design Specification, Architectural Interface Library”; Convex Corporation, Richardson; März 1994

HenPat96 J. L. Hennessy, D. A. Patterson: “Computer Architecture: A Quantitative Approach”, 2nd Edition; Morgan Kaufmann Publishers, San Francisco; 1996

HHLS+97 H. Härtig, M. Hohmuth, J. Liedtke, S. Schönberg, J. Wolter: “The Performance of μ-Kernel-Based Systems”; Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP’97); Operating Systems Review, Volume 31, Number 5, Seiten 52-65; ACM Press, New York; 1997;

Hofman91 F. Hofmann: “Betriebssysteme: Grundkonzepte und Modellvorstellungen”, 2. überarbeitete Auflage; B. G. Teubner, Stuttgart; 1991

Hollma96 J. Hollmann: “Adding Sleeping Threads to a MACH based Operating System (Erweiterung eines MACH basierten Betriebssystems um das Konzept der Sleeping Threads)”; Diplomarbeit 4/96, Institut für mathematische Maschinen und Datenverarbeitung IV, Universität Erlangen-Nürnberg; März 1996

IGHS94 R. A. Iannucci, G. R. Gao, R.H. Halstead, B. Smith: "Multithreaded Computer Architecture: A Summary of the State of The Art"; Kluwer, Norwell; 1994

IKNM91 S. Inohara, K. Kato, A. Narita, T. Masuda: "A Thread Facility Based on User/ Kernel Cooperation in the XERO Operating System"; Proceedings of the 15th IEEE International Computer Software and Applications Conference, Seiten 398-405; September 1991

InKaMa93 S. Inohara, K. Kato, T. Masud:. "Unstable Threads - Kernel Interface for Minimizing the Overhead of Thread Switching"; Technical Report, Department of Information Science, University of Tokyo; 1993

KEGB+97 M. F. Kaashoek, D. R. Engler, G. R. Ganger, H. M. Briceño, R. Hunt, D. Mazières, T. Pinckney, R. Grimm, J. Jannotti, K. Mackenzie: "Application Performance and Flexibility on Exokernel Systems"; Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP'97); Operating Systems Review, Volume 31, Number 5, Seiten 52-65; ACM Press, New York; 1997;

Keppel93 D. Keppel: "Tools and Techniques for Building Fast Portable Threads Packages"; Technical Report TR-UW-CSE-93-05-06, University of Washington; 1993

KLVA93 K. Krueger, D. Loftesness, A. Vahdat, T. Anderson: "Tools for the Development of Application-Specific Virtual Memory Management"; Proceedings of the Conference on Object-Oriented Programming Sytems, Languages and Applications (OOPSLA'93), Seiten 48-64; Oktober 1993

Koppe94 C. Koppe: "Sleeping Threads: Ein Kernmechanismus zur Unterstützung effizienter User-Level-Threads"; Interner Bericht TR-I4-94-15, Institut für mathematische Maschinen und Datenverarbeitung IV, Universität Erlangen-Nürnberg; 1994

Koppe95 C. Koppe: "Sleeping Threads: A Kernel Mechanism for Support of efficient User Level Threads"; Proceedings of the 7th IASTED/ISMM International Conference on Parallel and Distributed Computing and Systems (PDCS'95), Seiten 11-15; IASTED-ACTA PRESS, Anaheim; 1995; ebenso veröffentlicht als: Technical Report TR-I4-95-07, Institut für mathematische Maschinen und Datenverarbeitung IV, Universität Erlangen-Nürnberg; 1995

KSR91 "KSR1 Principles of Operation"; Kendall Square Research, Waltham; 1991

LewDan96 B. Lewis, D. J. Berg: "Threads Primer: A Guide to Multithreaded Programming"; Prentice Hall, Upper Saddle River; 1996

Liedtk96 J. Liedtke: " L4 Reference Manual - 486, Pentium, Pentium Pro - Version 2.0"; GMD Arbeitspapier 1021; September 1996

MasPu91 H. Massalin, C. Pu: “A Lock-Free Multiprocessor OS Kernel”; Technical Report CUCS-005-91, Department of Computer Science, Columbia University; Juni 1991

MGHK+94 J. G. Mitchell, J. J. Gibbons, G. Hamilton, P. B. Kessler, Y. A. Khalidi, P. Kougiouris, P. W. Madany, M. N. Nelson, M. L. Powell, S. R. Radia: “An Overview of the Spring System”; Proceedings of the Compcon Spring 1994, San Francisco; 1994

MKAK94 K. Mackenzie, J. Kubiatowicz, A. Agarwal, K. Kaashoek: “FUGU: Implementing Translation and Protection in a Multiuser, Multimodel Multiprocessor”; Technical Memorandum MTI/LCS/TM503, Laboratory of Computer Science, Massachusetts Institute of Technologie; Oktober 1994

MoDeKr96 T. C. Mowry, A. K. Demke, O. Krieger: “Automatic Compiler-Inserted I/O Prefetching for Out-of-Core Applications”; Proceedings of the 2nd Symposion on Operating Systems Design and Implementation (OSDI’96), Seiten 3-17; 1996

MSLM91 B. D. Marsh, M. L. Scott, T. J. LeBlanc, E. P. Markatos: “First-class user-level threads”; Operating Systems Review, Volume 25, Number 5, Seiten 110-121; 1991

Ouster82 J. K. Ousterhaug: “Scheduling Techniques for Concurrent Systems”; Proceedings of the Third International Conference on Distributed Computing Systems, Seiten 22-30; 1982

PerSit96 S. E. Perl, R. L. Sites: “Studies of Windows NT Performance using Dynamic Execution Traces”; Proceedings of the 2nd Symposion on Operating Systems Design and Implementation (OSDI’96), Seiten 169-199; 1996

RBFG+ R. Rashid, R. Baron, A. Forin, D. Golub, M. Jones, D. Julin, D. Orr, R. Sanzi: “Mach: A Foundation for Open Systems - A Position Paper”; School of Computer Science, Carnegie Mellon University

Reder95 U. Reder: “Implementierung eines effizienten Prozeßumschalters auf Benutzerebene”; Studienarbeit 2/95, Institut für mathematische Maschinen und Datenverarbeitung IV, Universität Erlangen-Nürnberg; Februar 1995

Reder96 U. Reder: “Betrachtung und Implementation effizienter benutzergesteuerter Unterbrechungsmechanismen in Mach”; Diplomarbeit 17/96, Institut für mathematische Maschinen und Datenverarbeitung IV, Universität Erlangen-Nürnberg; August 1996

RieKle96 T. Riechmann, J. Kleinöder: “User-Level Scheduling with Kernel Threads”; Technical Report TR-I4-96-05, Institut für mathematische Maschinen und Datenverarbeitung IV, Universität Erlangen-Nürnberg; Juni 1996

RitTho74 D. M. Ritchie, K. Thompson: “The UNIX Time-Sharing System”; Communications of the ACM, Volume 17, Number 7, Seiten 365-375; Juli 1974

SilGal94 A. Silberschatz, P. B. Galvin: “Operating System Concepts”, 4th edition; Addison-Wesley, Reading; 1994

Spin98 “Adapting SPIN for SMP”; http://www.cs.washington.edu/homes/ulbright/DualSpin.html, Department of Computer Science and Engineering, University of Washington, Seattle; Juli 1998

Stecker95 M. Steckermeier: “Einbeziehung von Lokalitätsinformation in Mechanismen zur Prozeßverwaltung auf Benutzerebene“; Diplomarbeit 25/95, Institut für mathematische Maschinen und Datenverarbeitung IV, Universität Erlangen-Nürnberg; Dezember 1995

Stoneb81 M. Stonebraker: “Operating System Support for Database Management”; Communications of the ACM, Volume 24, Number 7, Seiten 412-418; Juli 1981

Sun96 “Ultra Enterprise X000 Server Family: Architecture and Implementation”; Technical White Paper; Sun Microsystems, Mountain View; 1996

Tokoro90 M. Tokoro: “Computational Field Model: Toward a new Computing Model/ Methodology for Open Distributed Environment”; Proceedings of the 2nd Workshop on Future trends in Distributed computing Systems; Sepember 1990; ebenso veröffentlicht als: Sony CSL Technical Report SCSL-TR-90-006; 1990

Tucker93 A. Tucker: “Efficient Scheduling on Multiprogrammed Schared-Memory Multiprocessors”; Dissertationsschrift, Department of Computer Science, Stanford University; Dezember 1993

TuEgLe95 D. M. Tullsen, S. J. Eggers, H. M. Levy: “Simultaneous Multithreading: Maximizing On-Chip Parallelism”; Proceedings of the 22nd Annual International Symposium on Computer Architecture; June 1995

WCCJ+74 W. Wulf, E. Cohen, W. Corwin, A. Jones, R. Levin, C. Pierson, F. Pollack: “HYDRA: The kernel of a multiprocessing operating system”; Communications of the ACM, Volume 17, Number 6, Seiten 337-345; July 1974

WheBer92 B. Wheeler, B. N. Bershad: “Consistency management for virtually indexed caches”; Proceedings of the 5th Symposium on Architectural Support for Programming Languages and Operating Systems (ASPLOS V); October 1992

Lebenslauf

Name:	Christoph Koppe
geboren am:	31. Jan. 1964
Geburtsort:	Bonn

Ausbildung:

1970 – 1971	Katholische Grundschule Plittersdorf, Bonn
1971 – 1972	Schweizer Schule, Singapur
1972 – 1975	Katholische Grundschule Plittersdorf, Bonn
1975 – 1984	Heinrich-Hertz-Gymnasium, Bonn
1984 – 1985	Studium der Informatik an der Friedrich-Alexander-Universität Erlangen-Nürnberg
1985 – 1986	Kriegsersatzdienst, Bonn
1986 – 1993	Studium der Informatik an der Friedrich-Alexander-Universität Erlangen-Nürnberg
1993 – 1998	Wissenschaftlicher Assistent an der Friedrich-Alexander-Universität Erlangen-Nürnberg, Institut für Mathematische Maschinen und Datenverarbeitung IV, Lehrstuhl für Betriebssysteme (Prof. Dr. Fridolin Hofmann) und am Regionalen Rechenzentrum Erlangen