Sleeping Threads: Ein Kernmechanismus zur Unterstützung effizienter User-Level-Threads

Christoph Koppe

Februar 1995

TR-I4-15-94

Interner Bericht

Institut für
Mathematische Maschinen
und Datenverarbeitung
der
Friedrich-Alexander-Universität
Erlangen-Nürnberg

Lehrstuhl für Informatik IV
(Betriebssysteme)

Sleeping Threads: Ein Kernmechanismus zur Unterstützung effizienter User-Level-Threads

Christoph Koppe

Universität Erlangen-Nürnberg
Institut für mathematische Maschinen und Datenverarbeitung IV
Lehrstuhl für Betriebssysteme

chkoppe@informatik.uni-erlangen.de

Überblick

Die fortschreitende Entwicklung moderner Parallelrechner in den letzten Jahren ermöglicht - von Seiten der Hardware - die Granularität paralleler Algorithmen weiter zu verfeinern. Neben immer schnelleren Verbindungsnetzwerken mit kurzen Aufsetzzeiten für nachrichtenbasierte Architekturen wurde die Entwicklung von Systemen mit gemeinsamem Speicher vorangetrieben. Gerade für diese Systeme fehlt es jedoch an geeigneter Systemsoftware, um den hoch effizienten Kommunikationsmechanismus der Speicherkopplung für feingranulare parallele Algorithmen nutzen zu können.

Sind nachrichtenbasierte Programmiermodelle, die effizient auf speichergekoppelte Systeme abgebildet werden können, für eine Problemstellung nicht geeignet bzw. nicht effizient genug, muß der Anwender auf die leichtgewichtigen Prozesse (Threads) des Parallelrechnersystems zurückgreifen. Die im Betriebssystem integrierten Threads werden aber immer noch durch einen zu hohen Systemüberbau gebremst und werden daher meistens nur als Grundlage für eine effiziente Threadverwaltung auf der Anwendungsebene genutzt. Diesen User-Level-Threads fehlt jedoch andererseits eine genügende Systemeinbindung, was zu Problemen und Leistungseinbußen führt, insbesondere dann, wenn der meist fachfremde Anwender über keine genügend tiefe Kenntnis des Systems verfügt.

Die ungenügende Systemeinbindung der User-Level-Threads ist jedoch nicht in den Threadbibliotheken selbst begründet, sondern in unzureichenden Schnittstellen der Betriebssysteme. Die meisten Betriebsysteme für speichergekoppelte Parallelrechner ermöglichen zwar mehrere Threads pro Adreßraum, die für ein User-Level-Scheduling innerhalb einer Anwendung genutzt werden können, bieten darüberhinaus aber keinerlei Unterstützung. Wie in den folgenden Abschnitten dargelegt wird, benötigt ein hinreichend ins System eingebundenes User-Level-Scheduling jedoch weitergehende Informationen über z.B. die Zustände der Threads im Betriebssystemkern. Daher ist die erweiterte Kommunikation zwischen Kern und User-Level eine Möglichkeit, eine effiziente parallele Programmierung effektiv innerhalb der Anwendungsebene zu unterstützen.

Die „Sleeping Threads" sind ein Beispiel für den Einsatz einer solchen erweiterten Kommunikation. Die Threadbibliothek kann Kernel-Threads im Betriebssystemkern parken und sie im Zusammenspiel mit dem Kern aktivieren, wenn sich Threads einer Anwendung im Kern blockieren. Über diesen Mechanismus kann gewährleistet werden, daß die User-Level-Threads einer Anwendung immer auf eine konstante Anzahl von virtuellen - unter bestimmten Umständen physikalischen - Prozessoren abgebildet werden. Sämtliche Schedulingentscheidungen innerhalb einer Anwendung werden in die Threadbibliothek verlagert, was ein User-Level-Scheduling ermöglicht, welches die Besonderheiten von parallelen Architekturen, insbesondere hinsichtlich der Caches, berücksichtigen bzw. ausnutzen kann.

1 Einleitung

Mit dem verstärkten Aufkommen von Parallelrechnern im Bereich des Hochleistungsrechnens stellt sich eine neue Aufgabe in der Softwareentwicklung. Für konventionelle Supercomputer - im Sinne von Vektorrechnern - war eine Neuimplementierung von Anwendungen meist nicht notwendig. Alte Fortran Programme konnten im wesentlichen übernommen werden und mußten nur insoweit angepaßt werden, daß sie gut vektorisierbar waren. Für den Einsatz von Parallelrechnern müssen und mußten existierende Programme jedoch grundlegend überarbeitet und in weiten Teilen neu implementiert werden bzw. bei Neuentwicklungen andere Programmiermodelle („message passing" oder „shared memory"-basierte Modelle) eingesetzt werden.[1]

Je nach Struktur der verwendeten Parallelrechner bietet sich der Einsatz verschiedener Programmiermodelle an. Im wesentlichen sind dies „message passing" und „shared memory"-basierte Modelle. In den letzten Jahren haben sich die „message passing"-basierten Modelle weitgehend durchgesetzt. Nicht zuletzt deswegen, weil im Bereich der massiv parallelen Rechner keine „shared memory"-Architekturen zu finden waren. Der Grund hierfür liegt im Flaschenhals des Speicherbusses beim Zugriff vieler Prozessoren auf gemeinsamen Speicher. Dieser Engpaß ist mittlerweile durch das Aufkommen der NUMA-Architekturen[2] entschärft. Parallele Programme, die auf der Verwendung gemeinsamen Speichers basieren, werden daher in Zukunft eine größere Bedeutung erlangen.

Es hat sich gezeigt, daß die Verwendung herkömmlicher Prozesse (eigener Adreßraum, Schutzumgebung, ...), die sich spezielle gemeinsame Speicherbereiche vereinbaren, kein geeignetes Mittel sind, da der Aufwand für Prozeßumschaltungen zu hoch ist. Als Lösung wurden leichtgewichtige Prozesse (Threads) mit gemeinsamem Adreßraum entwickelt. Neben einer schnelleren Umschaltung zwischen den Threads (im Vergleich zu Prozessen) ergibt sich der Vorteil, daß der gesamte Adreßraum zwischen den Threads geteilt wird und damit gemeinsamer Speicher der Normalfall ist (d.h. es müssen keine speziellen Speichersegmente angelegt werden). Dies führt einerseits zu einer einfacheren Programmierung, andererseits ist die Fehleranfälligkeit bei diesem Verfahren größer, da die einzelnen Threads einer Anwendung untereinander über keine Speicherschutzmechanismen verfügen. Mit dem Konzept der Threads war eine deutlich feinere Granularität der Parallelisierung zu erreichen als dies bei „message passing"-basierten Programmiermodellen möglich ist (hohe Aufsetzzeiten für Kommunikation). Ein weiterer Schritt war, die vom Betriebssystem zur Verfügung gestellten leichtgewichtigen Prozesse (Kernel-Threads) als „virtuelle Prozessoren" einzusetzen und eine noch effizientere Threadumschaltung auf der Benutzerebene (User-Level) zu implementieren. User-Level-Threads verursachen andererseits einige Probleme, da ihnen im Vergleich zu betriebssystemunterstützten Threads eine enge Systemeinbindung fehlt [ABLL 92][LMMS 91]. Zu diesen Problemen gehört u.a.:

- Es ist nicht möglich einem bestimmten User-Level-Thread ein Signal zuzustellen.

1. Darüber hinaus ist es notwendig sich über die eingesetzten Algorithmen Gedanken zu machen, da diese sich teilweise wenig für eine effiziente Parallelisierung eignen.
2. Non Uniform Memory Access: Architekturen mit unterschiedlichen Zugriffszeiten auf den Hauptspeicher (abhängig von der Lage des Speichers im Gesamtsystem - kein globaler Speicherbus).

- Das Blockieren eines User-Level-Threads führt zum Blockieren des darunterliegenden Kernel-Threads (virtuellen Prozessors). Innerhalb der betroffenen Anwendung kann Parallelität verloren gehen.
- Zwischen den Schedulingentscheidungen des Kerns und der User-Level-Threadbibliothek[3] kann es zu Interferenzen kommen.

Wir meinen, daß ein Großteil dieser und anderer Probleme durch eine Kommunikation zwischen einer Threadbibliothek einerseits und dem Kern andererseits mit nur geringen Modifikationen und Erweiterungen des Betriebssystems gelöst werden kann. Gemeint ist eine bidirektionale Kommunikation, also nicht nur vom User-Level zum Kern (über Systemaufrufe) sondern auch in umgekehrter Richtung, angestoßen durch Entscheidungen im Kern (d.h. nicht nur über Ergebniswerte von Systemaufrufen).

Dieser Bericht befaßt sich im weiteren mit den „Sleeping Threads”, einem Kernmechanismus zur Lösung der Probleme mit im Kern blockierten User-Level-Threads und Interferenzen zwischen Schedulingentscheidungen von Kern und Threadbibliothek. Die „Sleeping Threads” sind ein Mechanismus, der im wesentlichen nur die Funktionalität des „Parkens” von Reservethreads im Kern zur Verfügung stellt und eine Verlagerung von Schedulingentscheidungen aus dem Kern in den User-Level ermöglicht. Die angesprochenen Probleme müssen über Strategien in einer Threadbibliothek unter Verwendung der neuen Mechanismen gelöst werden.

Der folgende Abschnitt befaßt sich mit grundsätzlichen Überlegungen zur Lösung der genannten Probleme und dem Ansatz, dies durch eine bidirektionale Kommunikation zwischen Kern und User-Level zu realisieren. Die Abschnitte 3 und 4 stellen den Mechanismus der „Sleeping Threads” und die dazugehörigen Strategien vor. Abschnitt 5 geht auf einige Fragestellungen ein, die eng mit der Problematik des hier vorgestellten User-Level-Scheduling zusammenhängen. Abschnitt 6 beschreibt einige Überlegungen zur Implementierung und die letzten beiden Abschnitte befassen sich mit weiteren Arbeiten in diesem Umfeld bzw. geben einen Ausblick auf die weitere Arbeit an diesem Projekt.

2 Kommunikation zwischen Kern und User-Level

Soll eine neue Funktionalität auf der Benutzerebene angeboten werden, für die Modifikationen im Betriebssystemkern selbst notwendig sind, gibt es zwei extreme Wege dies zu implementieren:

- Soviel wie möglich im Kern selbst realisieren und eine umfangreiche Systemschnittstelle anbieten.
- Sowenig wie möglich im Kern integrieren und den Rest in User-Level-Bibliotheken realisieren.

3. Im weiteren ist mit dem Begriff „Threadbibliothek” immer „User-Level-Threadbibliothek” gemeint.

Der erste Ansatz hat den Vorteil, daß bei einer Integration in den Kern vieles einfacher zu realisieren ist, da ein Zugriff auf alle Datenstrukturen des Kerns möglich ist und die Kontrolle über den Systemzustand weitgehend bei einer Instanz, dem Kern, liegt. Grenzen für eine Integration in den Kern sind u.a. durch Effizienzüberlegungen oder eine Überfrachtung des Kerns gegeben. Auch ergibt es keinen Sinn, Funktionalitäten in den Kern zu integrieren, die problemlos innerhalb von Bibliotheken bzw. Laufzeitsystemen[4] realisiert werden können. Der Nachteil dieses Ansatzes ist, daß weitgehende Modifikationen im Kern notwendig sind und keine Möglichkeit von einfachen und schnellen Änderungen oder Erweiterungen gegeben ist. Das System ist wenig flexibel. Der zweite Ansatz kann, je nach Problemstellung, zu einer komplexeren Realisierung führen. Dies ist darin begründet, daß die Statusinformation über das Gesamtsystem zwischen User-Level und Kern verteilt ist. Der Ansatz hat aber unübersehbare Vorteile[5]:

- Eine einfach zu realisierende Trennung zwischen Grundmechanismen und Strategien, die dann sinnvollerweise in einer Bibliothek (oder in einem User-Level-Server, wie z.B. im MACH-System möglich) realisiert werden.
- Ein einfacher Austausch von Strategien ist möglich.
- Nur der im Kern realisierte Teil muß dahingehend überprüft werden, daß er die Integrität des gesamten Betriebssystems nicht beeinträchtigt.
- Je kleiner der Anteil der Erweiterungen im Kern ist, um so problemloser ist eine Integration in andere Systeme.
- Eine weitgehend einheitliche Schnittstelle über Systemgrenzen hinweg ist einfacher zu realisieren.

Der Umfang der Modifikationen oder Erweiterungen im Betriebssystemkern kann unserer Meinung nach weiter verringert werden, wenn bidirektionale Kommunikationsmechanismen zwischen dem Kern einerseits und einer Threadverwaltung auf der Benutzerebene andererseits eingesetzt werden. Eine Kommunikation zwischen Kern und User-Level ermöglicht es Funktionalitäten weitgehend außerhalb des Kerns zu realisieren, für die ansonsten weitgehende Betriebssystemmodifikationen oder Erweiterungen nötig wären. Die Struktur der verwendeten Kommunikationsmechanismen soll allgemein nicht weiter festgeschrieben werden. Sie kann sowohl von der Struktur des Betriebssystems als auch von der zu implementierenden Funktionalität abhängen.

Neben dem im folgenden vorgestellten Mechanismus zur Lösung einiger Schedulingprobleme in Threadbibliotheken gelten entsprechende Überlegungen z.B. auch für die Zustellung von Signalen an User-Level-Threads. Kommunikation kann ebenso zwischen einem Anwendungsprogramm und z.B. einer Threadbibliothek bzw. dem Kern sinnvoll sein. Ein Beispiel hierfür ist die Notwendigkeit in bestimmten Situationen über Schedulingentscheidungen im Voraus informiert zu sein. Dies ermöglicht z.B einen effizienteren Umgang mit dem Betriebsmittel „Cache". Ins-

4. Unter einem Laufzeitsystem wird eine Bibliothek verstanden, die die Kontrolle über ein ablaufendes Programm übernimmt. Threadbibliotheken sind i.a. Laufzeitsysteme.
5. Bei dieser Fragestellung lassen sich die gleichen Argumente anführen, wie in der Diskussion über monolitische bzw. minimale Betriebssystemkerne mit ausgelagerten Servern.

besondere kann dies bei automatisch generierten parallelen Abläufen (parallelisierende Compiler) ausgenutzt werden, um rechtzeitig „Cacheprefetches” anzustoßen. Im Zusammenhang mit Caches ist es besonders wichtig, daß Schedulingentscheidungen nur auf Ebene der Threadbibliothek getroffen werden, da ein Scheduling durch den Kern die Vorteile eines cachesensitiven User-Level-Scheduling zunichte machen würde.

Eine eher allgemeine Betrachtung existierender Threadbibliotheken zeigt, daß sie nicht geeignet sind die Anforderungen von neuen, insbesondere numerischen Algorithmen für Parallelrechner zu erfüllen. Ein solcher Ansatz, der u.a. hohe Anforderungen an die Menge der verfügbaren User-Level-Threads stellt, wird in [Bellos 94] vorgestellt. Auch für solche Fragestellungen erscheint der Ansatz einer verstärkten Kommunikation zwischen Kern und User-Level sinnvoll, da er die Möglichkeit gibt, die eigentlichen Betriebssystemmodifikationen gering zu halten und damit u.a. eine weitere Forschung oft erst ermöglicht.

Langfristiges Ziel dieser Arbeit ist, die Auswirkungen einer Kommunikation zwischen Kern, Threadbibliothek und Applikation - u.a. auf die Effizienz und die Mächtigkeit des Programmiermodells „Threads” - zu untersuchen.

3 Sleeping Threads - der Mechanismus

3.1 Problemstellung

In der folgenden Abbildung wird das Problem der blockierenden User-Level-Threads verdeutlicht. Diese Probleme ergeben sich im verstärkten Maße, wenn Threads blockierende Systemaufrufe verwenden. In vielen Fällen wird dies vermeidbar sein, aber es ist mit Sicherheit ein einfacherer Umgang mit Threads möglich, wenn bei der Programmierung auf solche Dinge nicht geachtet werden muß (ansonsten ist z.B. nicht ohne weiteres eine E/A-Operation innerhalb eines Threads möglich). Ein Blockieren von User-Level-Threads läßt sich jedoch in keinem Falle völlig vermeiden. Unvermeidbar sind zumindest Blockierungen aufgrund von Speicherseitenanforderungen (page faults). In den folgenden Abbildungen werden Threads durch Kreise und Zustände im Kern durch Rechtecke dargestellt. Die Bindung eines User-Level-Threads an einen Kernel-Thread bzw. eines Kernel-Thread an einen Zustand im Kern wird durch Pfeile symbolisiert. In der ersten Abbildung (unverändertes System) sind nur die Zustände **R** (runable bzw. running, falls an einen Kernel-Thread gebunden) und der Kernzustand **B** (blockiert) von Interesse.

Das folgende Scenario soll die Problematik verdeutlichen: Eine Threadbibliothek bildet fünf User-Level-Threads auf drei Kernel-Threads (virtuelle Prozessoren) ab. Führt einer der User-Level-Threads einen blockierenden Systemaufruf aus, dann blockiert der ihn ausführende Kernel-Thread und damit der virtuelle Prozessor. Innerhalb der Anwendung können nur noch zwei Threads parallel ausgeführt werden, auch wenn weitere User-Level-Threads ablauffähig (runable) sind. Der Zustand des User-Level-Thread ändert sich durch die Blockierung nicht, da die Threadbibliothek ein Blockieren im Kern nicht feststellen kann.

STThreads: 14.02.95 13:04

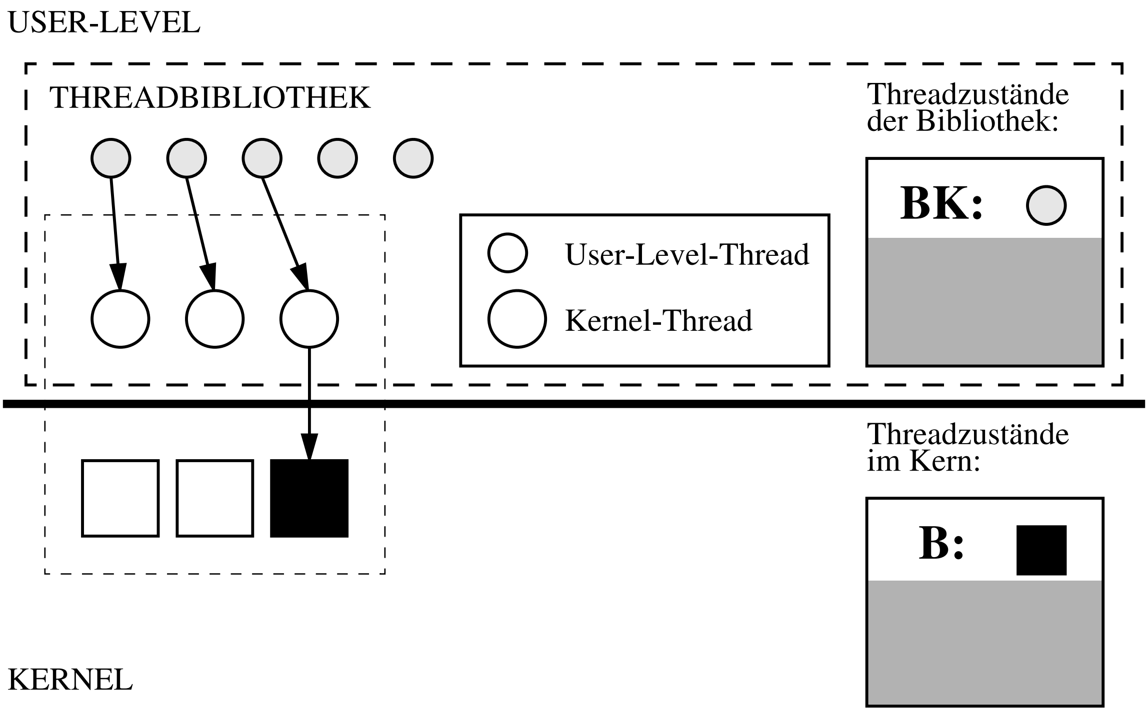

Abb. 3.1: Blockierender User-Level-Thread

Um einen solchen Verlust an Parallelität einer Anwendung zu vermeiden, müßten mehrere virtuelle Prozessoren auf einen physikalischen Prozessor abgebildet werden. Dies hat aber den Nachteil, daß das Scheduling nicht nur noch innerhalb der Threadbibliothek - auf den der Anwendung zur Verfügung stehenden Prozessoren - erfolgt, sondern auch der Kern beteiligt ist, da ein Scheduling der virtuellen Prozessoren selbst notwendig ist. Es bleibt jedoch in jedem Fall ein anderes Problem: Eine Anwendung kann sich verklemmen, wenn genau so viele User-Level-Threads blockieren, wie virtuelle Prozessoren verfügbar sind und zur Auflösung der Situation weiter Threads ausgeführt werden müssen. Mindestens dieses Problem muß gelöst werden, wenn zugelassen werden soll, daß sich User-Level-Threads blockieren. Eine Bindung jedes User-Level-Threads an einen eigenen Kernel-Thread wäre ebenfalls möglich, aber nicht sinnvoll. Die Vorteile des User-Level-Scheduling gingen verloren. Als ein möglicher Lösungsansatz seien die Solaris-Threads von Sun [Sun 93] erwähnt. Stellt der Kern fest, daß alle virtuellen Prozessoren (die „Lightweight Processes" von Sun) einer Anwendung blockiert sind, sendet er der Threadbibliothek ein spezielles Signal zu, um die Erzeugung eines weiteren Kernel-Thread zu veranlassen. Dies ist folglich ein Mechanismus, der erst eingreift, wenn eine Verklemmung vorliegt.

3.2 Grundmechanismus der Sleeping Threads

Der Mechanismus der „Sleeping Threads" leistet einiges mehr. Threads können auf Reserve im Kern geparkt werden, wobei eine sehr schnelle Umschaltung auf einen Reservethread erfolgt, wenn sich ein Thread blockiert. In einer Threadbibliothek können Strategien implementiert werden, die, in Abhängigkeit von den Anforderungen einer Anwendung, diese Reserve-Threads zur Verfügung stellen. Alle Schedulingentscheidungen - auch über deblockierte Threads im Kern - werden in die Threadbibliothek verlagert. Dies bedeutet, daß in Verbindung mit anderen Mechanismen, die einer Anwendung eine Menge von Prozessoren fest zuordnen, keine Beeinflussung einer Anwendung durch Schedulingentscheidungen innerhalb des Kerns mehr möglich ist. Der Mechanismus der „Sleeping Threads" ist optional, d.h. jeder Kernel-Thread der im Kern entsprechend des neuen Mechanismus behandelt werden soll, muß sich dieses Attribut explizit über einen Systemaufruf setzen. Dadurch ist gewährleistet, daß alte Applikationen auf dem modifizierten Kern unverändert weiter verwendet werden können.

In den folgenden Abbildungen zur Verdeutlichung des Mechanismus werden einige für die „Sleeping Threads" neu eingeführte Zustände (der Threadbibliothek und des Kerns) verwendet:

- **BK** User-Level-Thread im Kern blockiert (Threadbibliothek)
- **DK** User-Level-Thread im Kern deblockiert (Threadbibliothek)
- **SB** Blockierter „Sleeping Thread" (Kern); ersetzt den Zustand „blockiert"
- **SD** Deblockierter „Sleeping Thread" (Kern)
- **SS** Schlafender „Sleeping Thread" (Kern)

Desweiteren gibt es den Zustand **R** für ablauffähige User-Level-Threads mit unveränderter Bedeutung (siehe oben) und weitere Zustände der User-Level-Threads (wie z.B. „blockiert im User-Level"), die hier aber nicht von Interesse sind.

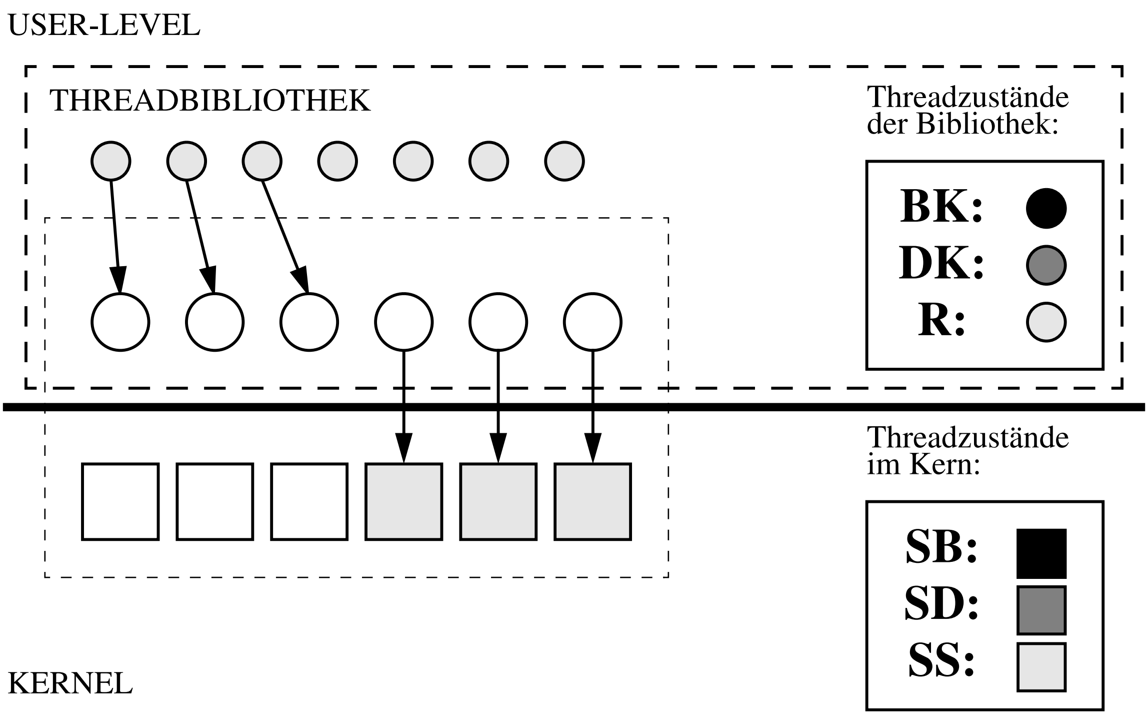

Abb. 3.2: „Sleeping Threads" werden geparkt

STHreads: 14.02.95 13:04

In Abbildung 3.2 hat die Threadbibliothek drei Kernel-Threads als „Sleeping Threads" im Kern geparkt. Realisiert wird dies durch einen weiteren neuen Systemaufruf, der von den entsprechenden Kernel-Thread ausgeführt wird und blockiert, bis der Reservethread benötigt wird.

Blockiert ein User-Level-Thread, für den das „Sleeping Thread" Attribut gesetzt ist, im Kern, geht der entsprechende Kernel-Thread in den Zustand „blockierter Sleeping Thread (**SB**)" über und einer der Reservethreads[6] wird deblockiert und kehrt aus dem Systemaufruf zurück. Der aktivierte Reservethread, kann innerhalb der Threadbibliothek einen ablauffähigen User-Level-Thread auswählen und ausführen. Dieser Mechanismus ist sehr effizient, da lediglich ein normaler Prozeßwechsel im Kern erfolgt (der bei einer Blockierung in jedem Fall notwendig ist) und zwar direkt von dem blockierenden Thread zum einem Reservethread unter Umgehung des üblichen Schedulingalgorithmus des Kerns. Der Zustand des Systems hat sich aus Sicht der Anwendung nicht geändert. Es stehen weiterhin drei virtuelle Prozessoren zur Abarbeitung der ablauffähigen User-Level-Threads zur Verfügung. An dieser Stelle sei noch mal erwähnt, daß sich eine Anwendung mit diesen Eigenschaften natürlich erst dann optimal (unbeeinflußt durch Scheduling des Kerns) verhält, wenn ihr eine den verwendeten virtuellen Prozessoren entsprechende Anzahl von physikalischen Prozessoren fest zugeordnet ist. Auf den Zusammenhang zu solchen Systemeigenschaften wird in Abschnitt 5 eingegangen.

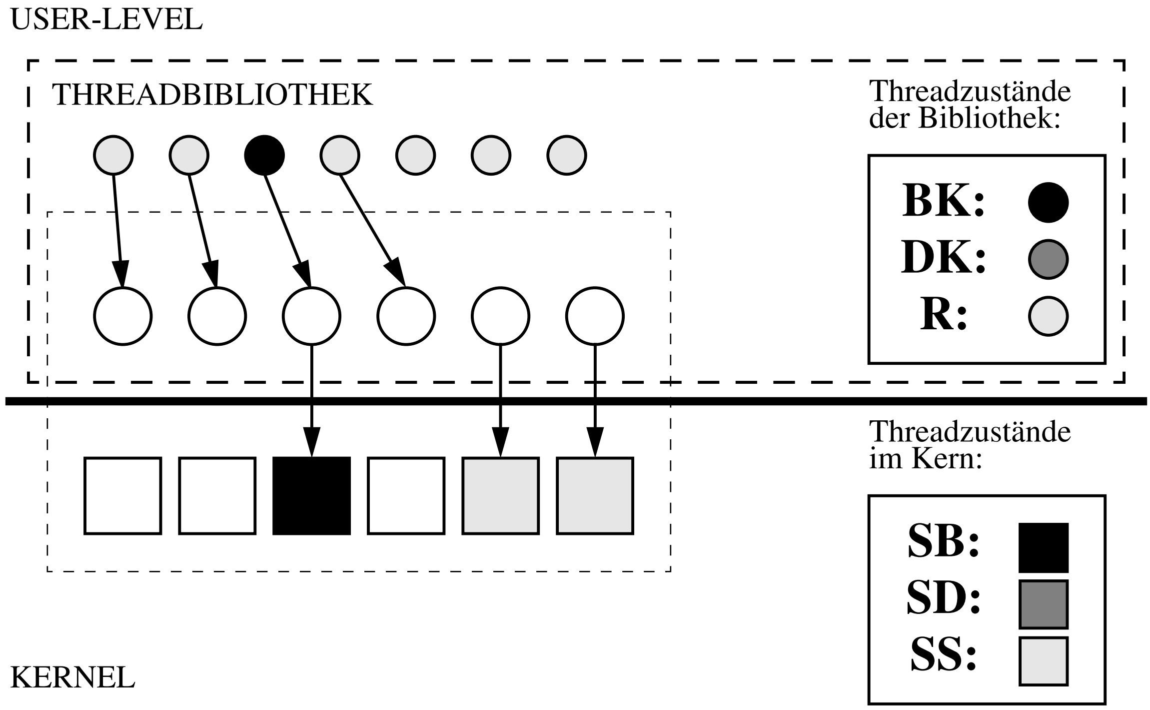

Abb. 3.3: Blockierter Kernel-Thread wird durch Reservethread ersetzt

6. Der Fall, daß kein Reservethread verfügbar ist, wird später behandelt.

Um die in Abbildung 3.3 dargestellte Situation zu erreichen, muß die Threadbibliothek über den neuen Zustand („blockiert im Kern (**BK**)") des blockierten User-Level-Threads informiert werden. Hierfür sind verschiedene Möglichkeiten denkbar:

- Die Identifikation des blockierten Thread kann als Rückgabewert des terminierenden Systemaufrufs des Reservethreads übergeben werden. Hier müßte ein Ausweichmechanismus existieren, wenn kein Reservethread verfügbar ist. Es ist aber in jedem Falle ein Mechanismus zur Behandlung dieser Situation notwendig (siehe auch Abschnitt 3.5).
- Der Einsatz vorhandener Nachrichtenmechanismen zwischen Kern und User-Level. Eine Implementierung auf MACH könnte die MACH-Messages verwenden. Ein expliziter Verwaltungsthread der Threadbibliothek, der nicht das „Sleeping Thread" Attribut gesetzt haben darf, müßte auf Nachrichten dieser Art warten. Dieser Thread ist während des Wartens auf Nachrichten blockiert und belastet das System nicht. Ein entscheidender Nachteil dieser Methode ist aber, daß das Versenden einer Nachricht zu einem Scheduling der Kernel-Threads führt. Der Verwaltungsthread muß zum Nachrichtenempfang einen virtuellen Prozessor der Threadbibliothek verdrängen. Dies ist aber genau das, was durch die „Sleeping Threads" verhindert werden soll. Da Nachrichten dieser Art auch in weiteren Fällen, etwa der Deblockierung eines Threads, notwendig sind, ist diese Variante nicht diskutabel, es sei denn der Verwaltungsthread läuft nicht auf der Menge der physikalischen Prozessoren der Anwendung. Die Problematik, die sich aus dieser Forderung ergibt, wird in den Abschnitten 5 und 6 eingehender diskutiert.
- Eine Verwendung gemeinsamer Speicherbereiche zwischen Kern und Threadbibliothek. Diese Variante ist die mit Abstand effizienteste, da der Kern selbst den Zustand eines User-Level-Threads setzen kann. Im Kern müssen aber Teile der Datenstrukturen der Threadbibliothek bekannt sein, wodurch diese in bestimmten Bereichen (Zustandsinformationen) festgelegt sind. Die entsprechenden Routinen des Kerns müssen robust genug sein, um nicht durch falsche oder inkonsistente Daten in diesem Bereich die Integrität des Kerns zu gefährden.

Die dritte Variante ist aus Effizienzgründen die günstigste und erscheint auch realisierbar, wenn die gemeinsamen Datenstrukturen zwischen Kern und Threadbibliothek klein gehalten werden. Sie ist aber sicher nicht die Variante, die einfach und mit möglichst geringen Erweiterungen des Kerns realisierbar ist. Welche Variante verwendet wird, wird bei der Darstellung der Implementierungsansätze (siehe Abschnitt 6) diskutiert.

Abbildung 3.4 zeigt die Situation, nachdem der blockierte Thread wieder ablauffähig ist. Der Kern darf ihn zu diesem Zeitpunkt nicht zur Ausführung bringen, da dann mit vier Kernel-Threads einer zuviel lauffähig wäre und dies zu einem Scheduling im Kern führen würde. Der Kernel-Thread geht in den Zustand „deblockierter Sleeping Thread (**SD**)" über und wiederum muß die Threadbibliothek über den Zustandswechsel eines User-Level-Thread („deblockiert im Kern (**DK**)") informiert werden. In diesem Falle ist die erste Kommunikationsvariante nicht möglich, da in dieser Situation kein Systemaufruf terminiert. Ist der User-Level-Thread als „deblockiert im Kern (**DK**)" markiert, kann die Threadbibliothek (nicht der Kern), ihn im Rahmen ihres Scheduling wieder zur Ausführung bringen.

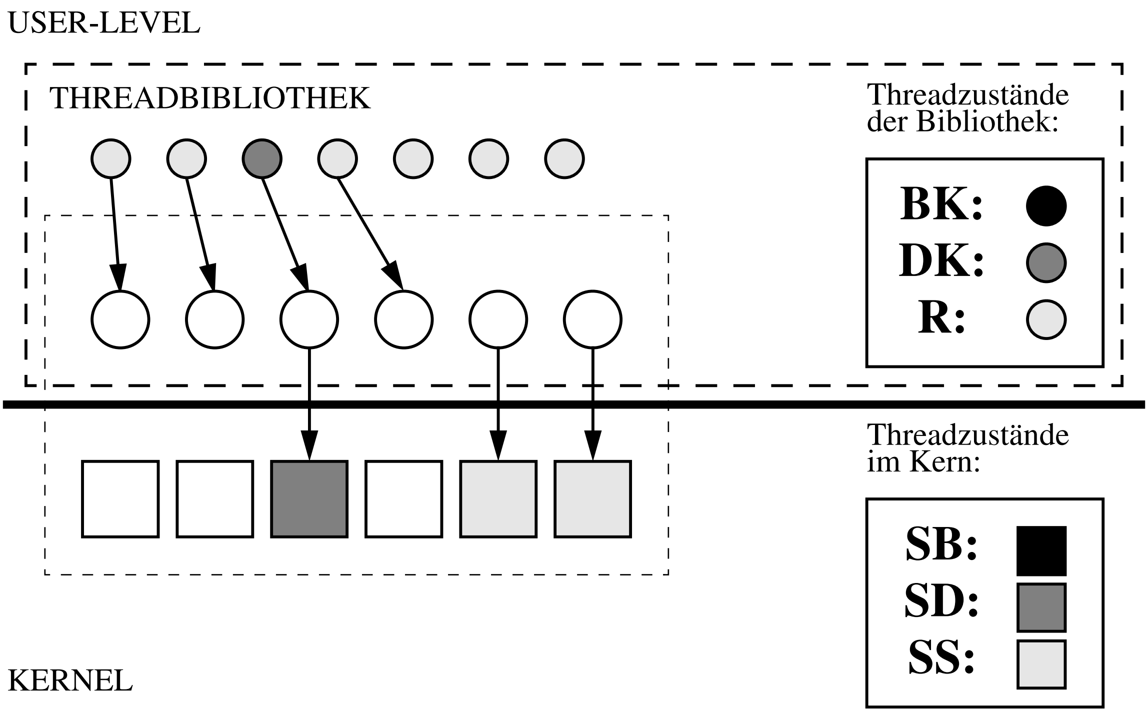

Abb. 3.4: Deblockierter Kernel-Thread

In Abbildung 3.5 wählt der zweite virtuelle Prozessor (von links) den deblockierten Thread zur Ausführung aus[7]. Der virtuelle Prozessor kann keine normale Prozeßumschaltung zwischen dem zweiten und dritten User-Level-Thread vornehmen, da sich der dritte weiterhin im Kernel-Modus befindet. Er führt statt dessen, nach der Kontextsicherung des zweiten User-Level-Threads, einen Systemaufruf durch mit dem er sich im Kern im Zustand „schlafender Sleeping Thread (**SS**)" blockiert und damit die Rolle eines Reservethreads übernimmt. Die Identifikation des an seiner Stelle auszuführenden User-Level- bzw. Kernel-Threads wird als Parameter des Systemaufrufs übergeben. An dieser Stelle wird wieder der normale Schedulingalgorithmus des Kerns umgangen, indem der physikalische Prozessor direkt auf den auszuführenden Kernel-

7. Der bisher laufende zweite User-Level-Thread ist weiterhin ablauffähig. Dies kann auftreten, wenn eine verdrängende User-Level-Schedulingstrategie realisiert ist. In den Abbildungen werden jedoch aus Gründen der Übersichtlichkeit nur die Zustände dargestellt, die für den Mechanismus von Bedeutung sind. Der zweite User-Level-Thread könnte beispielsweise auch im User-Level blockiert sein.

Thread umschaltet. Ab diesem Zeitpunkt wird der Systemaufruf, der zur Blockierung geführt hat, fortgesetzt. Das System befindet sich mit drei virtuellen Prozessoren und drei Reservethreads wieder im ursprünglichen Zustand.

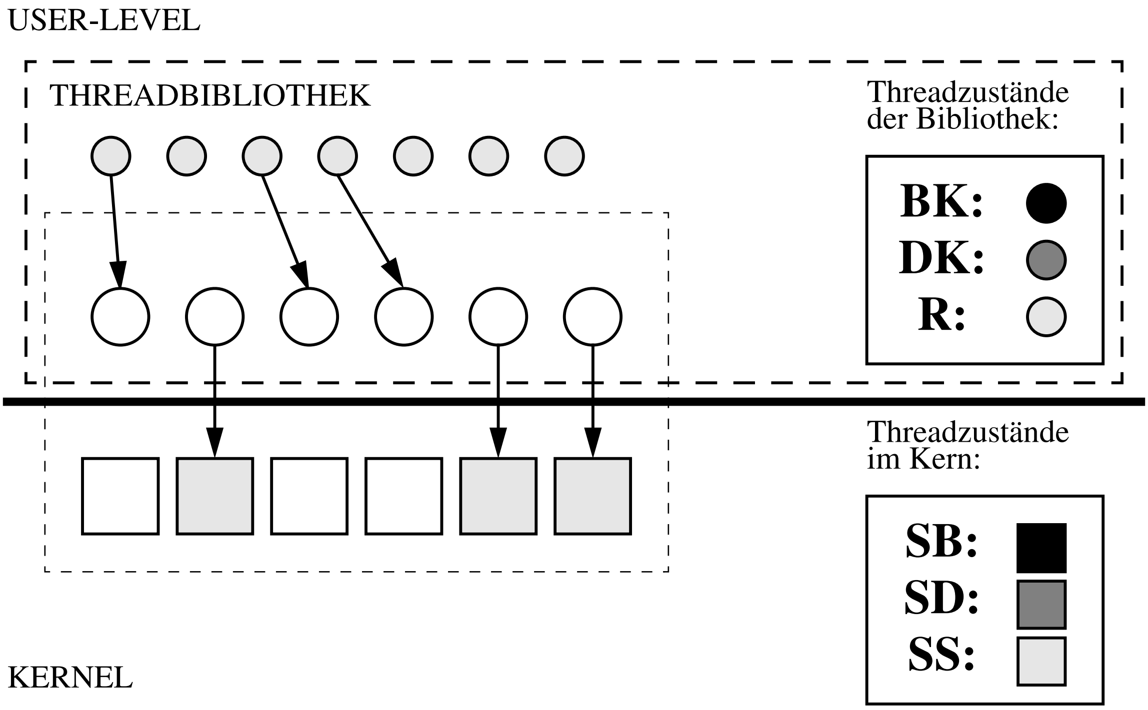

Abb. 3.5: Deblockierter User-Level-Thread wird wieder ausgeführt

3.3 Effizienz

Wird die Effizienz des Mechanismus „Sleeping Threads” betrachtet, lassen sich positive und negative Einflüsse feststellen. Das Aktivieren eines Reservethreads bzw. das Fortsetzen eines deblockierten Threads ist sehr effizient realisiert, da der entsprechende Kernel-Thread lediglich deblockiert und in den User-Level zurückgesetzt werden muß (Rücksprung eines Systemaufrufes). Der an dieser Stelle notwendige Threadwechsel würde in jedem Fall - aufgrund des Blokkierens bzw. Deblockierens im Kernel-Modus - erfolgen und wird durch die Umgehung des üblichen Schedulingalgorithmus weiter beschleunigt.

In diesem Zusammenhang muß untersucht werden, in wie weit es für eine Applikation möglich ist, durch geschicktes Blockieren, Deblockieren und Fortsetzen von Threads einen Prozessor dauerhaft zu binden und damit die Systemintegrität zu unterlaufen. In diesem Fall müßten Mechanismen integriert werden, die ein solches Applikationsverhalten verhindern. Diese Problem kann jedoch nur auftreten, wenn Applikationen nicht über eine fest zugeordnete Menge von Prozessoren verfügen. Der Bericht geht auf die angesprochene Problematik nicht weiter ein.

In der Beschreibung des Mechanismus der „Sleeping Threads” wurde angenommen, daß immer ein Reservethread verfügbar ist. In einem realen System ist es die Aufgabe der Strategien diese optimale Situation möglichst gut anzunähern. Das heißt, daß jede Strategie das oben beschriebene Optimum zumindest minimal einschränken wird, da selbst im günstigsten Fall zu Beginn des Programmablaufs eine - dann ausreichende - Zahl von Reservethreads generiert werden muß. Jedes nachträgliche Generieren von Reservethreads verschlechtert die Effizienz des Programmablaufs weiter. Im ungünstigsten Fall muß auf das Fehlen eines Reservethreads reagiert werden.

Einen weiteren negativen Einfluß auf die Effizienz hätte die Verwendung von aufwendigen Kommunikationsmechanismen wie z. B. den Mach-Messages.

3.4 Schedulingverlagerung in den User-Level

Wie im vorhergehenden Abschnitt beschrieben, führt der Kern einen deblockierten Kernel-Thread nie von sich aus fort. Er informiert lediglich die Threadbibliothek über den neuen Zustand des Threads. Dies bedeutet, daß sämtliche Schedulingentscheidungen innerhalb einer Applikation von der Threadbibliothek getroffen werden. Die Threadbibliothek ist im allgemeinen auch diejenige Instanz, innerhalb der es möglich ist, zu bestimmen, wann ein Fortsetzen eines Threads sinnvoll ist (beispielsweise über eine Schnittstelle zur Applikation und/oder entsprechende Schedulingstrategien).

Um den Einfluß des Scheduling innerhalb des Kerns auf eine Applikation völlig auszuschließen, reicht dieser Mechanismus jedoch nicht aus. Es muß zusätzlich gewährleistet werden, daß einer Applikation eine feste Anzahl von Prozessoren, die der Anzahl der aktiven Kernel-Threads entspricht, zugeordnet ist. Erst dann ist ein Scheduling des Kerns zwischen den Kernel-Threads einer oder mehrerer Applikationen ausgeschlossen. Mechanismen, die in diesem Zusammenhang betrachtet werden müssen, sind u.a. Job- bzw. Gangscheduling [Ouster 82][FeRu 90] und der MACH CPU-Server [Black 89][Black 90][Black 90a]. Abschnitt 5 geht auf den Zusammenhang mit solchen Mechanismen näher ein. Eine weitere Einschränkung dieser idealen Situation kann durch die verwendeten Kommunikationsmechanismen zwischen Kern und Threadbibliothek (z.B. bei Einsatz von Mach-Messages) verursacht werden (siehe Abschnitt 3.2).

3.5 Strategieanbindung und Behandlung von Ausnahmesituationen

Um die Frage zu beantworten, ob der in den vorangehenden Abschnitten vorgestellte Mechanismus mächtig genug ist, um Strategien mit dem gewünschten Leistungsumfang anzubinden, müssen die Anforderungen an die Strategien definiert werden. Dabei sind zwei Bereiche zu unterscheiden: Der erste Bereich, der hier nicht näher betrachtet werden soll, umfaßt das eigentliche User-Level-Scheduling. Auch hier muß es eine Schnittstelle zu den „Sleeping Thread"-Strategien geben, da Schedulingentscheidungen und Generieren bzw. Abbauen von Reservethreads von einander abhängen können. Zur Realisierung des eigentlichen User-Level-Scheduling sind, je nach Funktionalität (z.B. Signale an User-Level-Threads), weitere Kernmechanismen notwendig. Der Bereich, der in diesem Zusammenhang von Interesse ist, ist die Funktionalität bezüglich der „Sleeping Thread"-Verwaltung. Eine Strategie muß folgendes leisten:

- Buchführung über die Anzahl der (verfügbaren) Reservethreads.
- Erhöhen bzw. Erniedrigen der Anzahl der (verfügbaren) Reservethreads.
- Buchführung über die Anzahl der im Kern blockierten bzw. deblockierten „Sleeping Threads".
- Fortsetzen von im Kern deblockierten „Sleeping Threads".
- Buchführung über die Anzahl der virtuellen Prozessoren.
- Behandlung der Ausnahmesituation, daß kein Reservethread verfügbar ist.

Diese Funktionalität sollte in Form von strategieunabhängigen Grundfunktionen implementiert werden.

Für die ersten vier Punkte reichen die bisher vorgestellte Mechanismen aus. Da alle Operationen, die Reservethreads betreffen, über die Strategien laufen, ist eine Buchführung über die Anzahl der Reservethreads ohne weiteres möglich. Eine explizite Erhöhung der Anzahl der verfügbaren Reservethreads erfolgt durch Installieren eines neuen Threads als Reservethread. Eine implizite Erhöhung der verfügbaren Reservethreads, ergibt sich bei dem Fortsetzen eines deblockierten „Sleeping Thread", da in diesem Fall der freigebende Kernel-Thread nicht weiter als virtueller Prozessor verwendet werden darf (Anzahl der virtuellen Prozessoren soll konstant sein) und die Funktion des Reservethread übernehmen muß. Eine implizite Verringerung der verfügbaren Reservethreads erfolgt, wenn ein Thread im Kern blockiert und ein Reservethread als Ersatz frei gegeben wird. Eine explizite Verringerung der Reservethreads (wobei die Anzahl der verfügbaren unverändert bleibt) ist möglich, wenn ein im Kern deblockierter „Sleeping Thread" fortgesetzt wird und der freigebende Thread nicht als Reservethread im Kern blockiert wird, sondern den Systemaufruf beendet und dann terminiert. Hier ist eine kleine Erweiterung der vorgestellten Mechanismen notwendig.Eine zweite Erweiterung muß erfolgen um eine Verringerung der Reservethreads auch unabhängig von einem Fortsetzen eines deblockierten „Sleeping Thread" zu ermöglichen.

Das Fortsetzen eines deblockierten „Sleeping Thread" erfolgt direkt durch den beschriebenen Mechanismus aufgrund einer Anforderung durch die Schedulingstrategie. Eine Buchführung über blockierte bzw. deblockierte „Sleeping Threads" ist nicht ohne weiteres möglich, da aus Gründen der Effizienz der Kern - mittels einer Zustandsänderung[8] - nur die Schedulingstrategie über eine Blockierung bzw. Deblockierung informiert. Diese sollte der „Sleeping Thread"-Strategie jedoch Informationen hierüber verfügbar machen, da dies Entscheidungen über Bereitstellung bzw. Rücknahme von Reservethreads beeinflussen kann.

8. Wir betrachten hier nur den Idealfall einer Implementierung über gemeinsame Datenbereiche zwischen Kern und Threadbibliothek.

Als letztes bleiben die Buchführung über die Anzahl der virtuellen Prozessoren und die Ausnahmesituation, daß kein Reservethread verfügbar ist, wenn sich ein „Sleeping Thread” blockiert. Ersteres erscheint in dieser Aufzählung überflüssig, da es ja gerade ein Ziel der „Sleeping Threads” ist, diese konstant zu halten. Eine Buchführung ist aber unerläßlich, da es, wie im folgenden dargestellt, zu einem „Verlust” an virtuellen Prozessoren kommen kann. Zweiteres sollte eine Ausnahmesituation sein, da eine Vermeidung gerade die Aufgabe der „Sleeping Thread”-Strategien ist. Der Thread wird auch in diesem Fall wie ein „Sleeping Thread” behandelt. Es wird aber kein Reservethread als neuer virtueller Prozessor verwendet, sondern die Strategie als Teil der Threadbibliothek muß über den „Verlust” eines virtuellen Prozessors informiert werden. Die Strategie kann entweder die Situation hinnehmen und bei nächster Gelegenheit (z.B. beim Fortsetzen eines deblockierten „Sleeping Thread”) bereinigen oder der Schedulingstrategie einen neuen virtuellen Prozessor zur Verfügung stellen. Unabhängig von dieser Entscheidung können weitere Reservethreads installiert werden. Da nicht garantiert werden kann, daß zu diesem Zeitpunkt ein weiterer virtueller Prozessor verfügbar ist, der unterbrochen werden könnte, um diese Aufgabe durchzuführen, muß innerhalb der Anwendung eine weiterer Thread existieren, der für solche Ausnahmesituationen reserviert ist. Dies könnte, wie schon in Abschnitt 3.2 beschrieben, ein Verwaltungsthread sein, der mittels eines vorhandenen Nachrichtenmechanismus (z.B. den MACH-Messages) auf solche Nachrichten wartet. Eine solche Nachrichtenübermittlung kann aber zu diversen Threadumschaltungen führen, die außerhalb der Kontrolle der Threadbibliothek liegen. Eine alternative Methode ist die Verwendung eines weiteren Systemaufrufs, in dem ein Thread auf diese Ausnahme- und Notfallsituation wartet. Da durch die Blockierung ohne Reservethread ein Prozessor frei wird, könnte dieser unter Umgehung der üblichen Schedulingmechanismen direkt den Notfallthread ausführen. Der Notfallthread behandelt den Engpaß und wartet auf seine nächste Aktivierung. Die Aktivierungen müssen im Kern in einer Warteschlange verwaltet werden, um keine Nachrichten über den Verlust von virtuellen Prozessoren zu verlieren.

Eine Schnittstelle für die Strategien soll hier nicht definiert werden. Wichtig werden dabei jedoch hauptsächlich zwei Punkte sein: Wie kann eine einfache Austauschbarkeit der Strategien (eventuell auch dynamisch zur Laufzeit) gewährleistet werden und zu welchen Zeitpunkten bzw. bei welchen Ereignissen wird die „Sleeping Thread”-Strategie aktiviert.

4 Sleeping Threads - die Strategien

Wie schon im vorhergehenden Abschnitt angedeutet setzt sich eine Threadbibliothek, die auf dem „Sleeping Thread”-Mechanismus aufsetzt, aus Grundfunktionen und zwei austauschbaren Strategien zusammen. Eine der Strategien implementiert das User-Level-Scheduling der Threads. Hierfür sind verschiedene Varianten (z.B. unterbrechend/nicht unterbrechend, prioritätsgesteuert/Zeitscheibenverfahren ...) denkbar. Diese Strategie, die Schedulingstrategie, wird hier nicht weiter betrachtet.

Die zweite Strategie ist die „Sleeping Thread"-Strategie.Über sie wird gesteuert, wann Reservethreads generiert bzw. abgebaut werden. Um ein optimales Verhalten einer Applikation zu erreichen, muß diese Strategie sowohl an die Applikation als auch an die Schedulingstrategie angepaßt sein. Ein Beispiel für eine sehr einfache Strategie ist eine, die bei Applikationsstart eine feste Anzahl von Reservethreads installiert und dann nur noch aufgerufen wird, wenn Ausnahmesituationen (keine Reservethreads verfügbar) behandelt werden müssen. Dies kann sinnvoll sein, wenn die Anzahl der maximal benötigten Reservethreads abschätzbar ist, weil beispielsweise die Anzahl der User-Level-Threads begrenzt und niedrig genug ist. Ein anderer sinnvoller Ansatz ist es, die Anzahl der verfügbaren Reservethreads konstant bzw. innerhalb eines bestimmten Bereichs zu halten.

Im Zusammenhang mit beiden Arten von Strategien („Sleeping Thread"- und Schedulingstrategie) stellt sich folgende Frage: Ist es sinnvoll den „Sleeping Thread"-Mechanismus zu erweitern, um steuern zu können, welche Art von Blockierungen über den Mechanismus behandelt werden sollen und welche nicht. Eventuell kann ein besseres Applikationsverhalten erreicht werden, wenn z.B. Blockierungen aufgrund von Seitenanforderungen in Folge eines Seitenfehlers(Pagefault) nicht behandelt werden, sondern den üblichen Betriebssystemmechanismen überlassen werden, wobei die kurzzeitige Blockierung eines virtuellen Prozessors hingenommen wird. Entsprechende Parameter könnten durch die Applikation gesetzt werden. Zu untersuchen ist, ob der „Sleeping Thread"-Mechanismus, ohne zu großen Aufwand, so in ein System integrierbar ist, daß solche Unterscheidungen möglich werden.

5 Weitere Mechanismen

Eines der hier beschriebenen Ziele der „Sleeping Thread" ist, das Scheduling innerhalb einer Anwendung aus dem Betriebssystemkern in den User-Level zu verlagern. Unter einem multitasking-fähigen Betriebssystem ist dies jedoch i.a. nicht möglich, da die virtuellen Prozessoren einer „Sleeping Thread"-Anwendung zu Gunsten anderer Applikationen vom Kern suspendiert werden können. Der „Sleeping Thread"-Mechanismus ermöglicht es nicht dies, z.B. durch Reservieren einer Menge von Prozessoren, zu verhindern. Es gibt jedoch eine Reihe anderer Ansätze, die zum Ziel haben einzelne Threads oder Prozesse einer Anwendung auf einem Multiprozessor gleichzeitig zur Ausführung zu bringen. Eine Variante ist das in [Ouster 82] und [FeRu 90] beschriebene Job- bzw. Gangscheduling. Hier wird eine Schedulingstrategie eingesetzt, die versucht bevorzugt solche Threads oder Prozesse parallel auszuführen, die zu einer Anwendung gehören. Eine Beeinflussung der Anwendung durch Schedulingentscheidungen des Kerns wird mit diesem Verfahren nicht verhindert, aber Wartezeiten auf andere Threads oder Prozesse der gleichen Anwendung sind im Mittel kürzer, da diese mit größerer Wahrscheinlichkeit zur gleichen Zeit ausgeführt werden. Eine weitere Lösung, die für MACH entwickelt wurde, ist der MACH CPU-Server, der u.a. in [Black 89] beschrieben wird. Der CPU-Server geht einen anderen Weg. Eine Anwendung kann eine Menge von Prozessoren reservieren. Diese Prozessormenge steht der Anwendung exklusiv zu Verfügung, bis diese sie wieder frei gibt, oder der Zeitrahmen der Reservierung abgelaufen ist. Die bei diesem Verfahren notwendigen Regeln zur fairen Prozessorvergabe werden durch den CPU-Server überwacht. Innerhalb bestimmter Zeitrahmen ist also ein ausschließliches, unbeeinflußtes User-Level-Scheduling einer Anwendung möglich.

Solche Verfahren können mit dem „Sleeping Thread"-Mechanismus kombiniert werden, um User-Level und Kern-Scheduling weiter zu entflechten und dadurch die Nachteile zweier konkurrierender Schedulingstrategien (siehe Abschnitte 2 und 3) zu vermeiden. Welche Mechanismen für eine Kombination mit den „Sleeping Threads" sinnvoll sind hängt im hohen Maß von dem Basisbetriebssystem der „Sleeping Threads"-Portierung ab. Wird beispielsweise ein Gangscheduling vom Betriebssystem unterstützt, kann auf die Implementierung anderer Mechanismen zunächst verzichtet werden. Eine in den „Sleeping Threads" integrierte Prozessorzuteilungstrategie ist aber als ein weiterer Schritt anzustreben. Insbesondere erscheint ein Mechanismus sinnvoll, über den sich Kern und Threadbibliothek über die Anzahl der zugeordneten Prozessoren verständigen können. Dabei muß der Kern die Möglichkeit haben, einer Anwendung Prozessoren zu entziehen, wenn diese für andere Anwendungen benötigt werden. Dies wäre ein Kompromiß zwischen flexiblen Prozessorscheduling und exklusiver Reservierung.

6 Implementierungsansätze

Eine erste Implementierung ist unter dem Betriebssystem MACH geplant. Als Systeme stehen die Convex Exemplar Architektur und eine Portierung von Mach 3.0 für ein MC 88000 basiertes Multiprozessorsystem von Motorola zur Auswahl. Die Convex Exemplar ist eine NUMA Architektur mit bis zu 128 HP RISC 7100 Prozessoren. Das Betriebssystem basiert auf OSF/1 AD [Convex 94]. Das alternative System ist ein am IMMD IV entwickelter Port von MACH 3.0 auf ein symmetrisches 4 Prozessor VME System von Motorola mit uniformen Speicherzugriff (MVME 188) [Meier 94]. In diesem System fehlen jedoch einige interessante Aspekte, die NUMA Architekturen betreffen.

Einige der Implementierungsüberlegungen treffen jedoch nicht nur auf MACH-Systeme, sondern zumindest allgemein auf UNIX-Systeme zu. Um eine Kompatibilität des Systems zu alten Anwendungen zu erhalten, muß eine Datenstruktur gefunden werden, in der kodiert werden kann, ob Thread den neuen Mechanismus oder das normale Systemscheduling nutzen soll. Geeignet wäre z.B. ein ungenutztes Bit im Thread Status. Entsprechende Datenstrukturen im Kern können um Anteile für die Verwaltung „Sleeping Threads" erweitert werden. Eine Kompatibilität bleibt erhalten, wenn die Systemschnittstellen, die auf diese Datenstrukturen zugreifen, angepaßt werden. Systemanwendungen, die Kenntnis über die Größe dieser Datenstrukturen haben (z.B. ps) müßten neu übersetzt werden. Da die Behandlung der Reservethreads und blokkierten bzw. deblockierten „Sleeping Threads" im Kontext einer Anwendung erfolgt, sollten prozeß-spezifische Warteschlangen zur Verwaltung verwendet werden. Die notwendigen Modifikationen im Kern konzentrieren sich auf die Schedulingmechanismen, insbesondere auf die Funktionen, die Thread blockieren bzw. deblockieren.

Zu klären ist, ob die in Abschnitt 3.2 aufgeworfene Frage, wie die Threadbibliothek über Zustandsänderungen im Kern informiert wird. Da die Variante der Systemaufrufrückgabewerte nicht immer einsetzbar und die Aktivierung eines Verwaltungsthreads zu teuer ist, bleibt für eine effiziente Implementierung nur die Verwendung gemeinsamer Speicherbereiche zwischen Threadbibliothek und Kern. Bei einer Implementierung unter MACH ist dies einfach zu realisieren und kann deshalb auch als Lösung für einen ersten Prototypen gewählt werden. Ebenfalls einfach unter MACH zu realisieren ist das Umgehen der Schedulingmechanismen zur schnellen Umschaltung auf Reservethreads. Mach unterstützt mit dem Handoff-Scheduling [Black 90a] Mechanismen, die dies ermöglichen.

7 Verwandte Arbeiten

Der Gegensatz zwischen einer effizienten Threadumschaltung einerseits und einer ungenügenden Systemintegration andererseits findet sich in fast allen heute verfügbaren Threadbibliotheken. Von den verschiedenen Lösungsansätzen sollen hier die „Scheduler Activations" [ABLL 92] und die „First-Class User-Level Threads" [LMMS 91] erwähnt werden.

Die „Scheduler Activations" sind dem Ansatz der „Sleeping Threads" am ähnlichsten. Für einen blockierten Thread stellt der Kern der Threadbibliothek eine neue „Scheduler Activation" (Kernel-Thread) zur Verfügung. Der Hauptunteschied ist, daß dies nicht ein im voraus geparkter Thread ist sondern eine im Kern neu instanziierte „Scheduler Activation", die einen so genannten Upcall in den Adreßraum der Threadbibliothek (Anwendung) ausführt. Für den Upcall wird ein freier prozessor benötigt. Im Fall einer Blockierung wird der freiwerdende Prozessor verwendet. Im Fall einer Deblockierung dagegen, muß der Kern einen User-Level-Thread unterbrechen, um den Upcall auszuführen, um die Threadbibliothek über die Deblockierung zu informieren. Eine andere Möglichkeit besteht nicht, da der Upcall der einzige Kommunikationskanal vom Kern zur Threadbibliothek ist. Dies bedeutet, das der Kern in das User-Level Scheduling eingreift, indem er beispielsweise einen hochprioren Thread unterbricht, um die Deblockierung eines Threads mit niedrigerer Priorität bekannt zu machen. Ein wichtiger Unterschied ist, daß eine Verwaltung von Reservethreads nicht notwendig ist. „Scheduler Activations" werden bei Bedarf generiert. Inwieweit dies gegenüber der Aktivierung geparkter Threads zu Effizienzunterschieden führt, muß noch näher betrachtet werden. In den „Scheduler Activations" ist eine Prozessorzuteilungsstrategie bereits integriert. Eine Portierung auf MACH ist in [BMVL 93] beschrieben.

Die „First-Class User-Level Threads" verfolgen einen anderen Ansatz. Kern und Threadbibliothek arbeiten auf gemeinsamen Datenstrukturen, die zum Teil im User-Level schreibbar bzw. nur lesbar sind. Über diese Datenstrukturen wird eine Schnittstelle zwischen Kern und Threadbibliothek realisiert. Teil dieser Schnittstelle sind Softwareinterrupts vom Kern an die Threadbibliothek. Die in diesem Zusammenhang interessanten Softwareinterrupts sind „thread blocked in kernel" und „thread unblocked in kernel", über die der Kern der Threadbibliothek Blockierungen bzw. Deblockierungen bekannt gibt. Die Threadbibliothek kann, da sie Zugriff auf Teile der Datenstrukturen hat auf einen lauffähigen Thread umschalten. Besonderheiten von

NUMA-Architekturen werden dadurch berücksichtigt, daß der Kern einen virtuellen Prozessor immer auf dem gleichen physikalischen Prozessor ausführt und Softwareinterrupts immer dem virtuellen Prozessor zustellt, der den betroffenen Thread ausführt bzw. ausgeführt hat. Damit ist die Threadbibliothek die einzige Instanz, die einen Thread auf einen anderen Prozessor verlagen kann.

8 Zusammenfassung und Ausblick

Mit den „Sleeping Threads" wurde hier ein Mechanismus vorgestellt, der es ermöglicht einige der Probleme von User-Level-Threadbibliotheken zu lösen. Über das Parken von Reservethreads im Bertiebssystemkern kann die Threadbibliothek auf im Kern blockierte Threads reagieren und die Anzahl der virtuellen Prozessoren auch in solchen Situationen konstant halten. Sämtliche Schedulingentscheidungen innerhalb einer Anwendung - auch aufgrund von Blockierungen im Kern - werden über diesen Mechanismus in den User-Level verlagert. Dadurch ist es möglich die architekturellen Besonderheiten modernen Parallelrechner (z.B. die Speicher- und Cachearchitektur) in den User-Level-Schedulingstrategien zu berücksichtigen, ohne daß Threadumschaltungen auf Systemebene die Vorteile dieser Strategien relativieren. Basierend auf den „Sleeping Threads" können effiziente User-Level-Threadverwaltungen aufgebaut werden, die besser in des Betriebssystem eingebunden sind als bisherige Lösungen. Die gute Systemeinbindung wird durch eine erweiterte Kommunikation zwischen Threadbibliothek und Kern realisiert. Wir meinen, daß dieser Ansatz ebenfalls geeignet ist weitere Funktionalitäten von Kernel-Threads auch für User-Level-Threads zu realisieren. Dies gilt u.a. für Signale an einzelne Threads. Ein weiterer Vorteil der erweiterten Kommunikation ist, daß große Teile der Lösungen im User-Level realisiert werden können und nur geringe Modifikationen und Erweiterungen im Betriebssystemkern selbst notwendig sind.

Ein Ziel der weiteren Arbeiten in diesem Projekt ist, die Auswirkungen einer Kommunikation zwischen User-Level und Kern im Bereich der parallelen Programmierung auf Architekturen mit gemeinsamen Speicher (insbesondere NUMA-Architekturen) zu untersuchen. Wie wirkt sich der vorgestellte Mechanismus, eine weitergehende Kommunikation bzw. eine weitere Verlagerung von Teilen des Multiprozessorsupports vom Kern in den User-Level auf Effizienz und Mächtigkeit des Programmiermodells „Threads" aus?

Die nächsten Schritte sind eine Implementierung der „Sleeping Threads" und die Anpassung von Threadbibliotheken an den neuen Mechanismus. Basiserend auf dieser Implementierung soll das Verhalten von Anwendungen unter Verwendung der „Sleeping Threads" untersucht werden. Dabei wird ein Schwerpunkt auf dem Vergleich verschiedener Schedulingstrategien hinsichtlich der Cacheausnutzung auf NUMA-Architekturen liegen.

9 Literaturverzeichnis

ABLL 92 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, Vol. 10, No.1, Seiten 53-79, 1992

Bellos 94 F. Bellosa, Parallele leichtgewichtige Prozesse zur Implementierung adaptiver numerischer Verfahren, Universität Erlangen-Nürnberg, IMMD IV, Interner Bericht TR-I4-2-94, 1994

Black 89 D. J. Black, The Mach cpu_server: An Implementation of Processor Allocation, unveröffentlichtes Manuskript, Carnegie Mellon University, 1989

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

Black 90a D. J. Black, Scheduling and Resource Management Techniques for Multiprocessors, Carnegie Mellon University, Technical Report CMU-CS-90-152, 1990

BMVL 93 P. Barton-Davis, D. McMamee, R. Vaswani, E. D. Lazowska, Adding Scheduler Activations to Mach 3.0, USENIX Mach III Symposium Proceedings, Seiten 119-136, 1993

Convex 94 Convex Exemplar Architecture, Second Edition, Convex Press, Richardson, 1994

FeRu 90 D. Feitelson, L. Rudolph, Distributed Hierarchical Control of Parallel Processing, IEEE Computer, Vol. 23, No. 5, Seiten 65-77, 1990

LMMS 91 T. J. LeBlanc, E. P. Markatos, B. D. Marsh, M. L. Scott, First-Class User-Level Threads, Operating Systems Review, Vol. 25, No. 5, Seiten 110-121, 1991

Meier 94 E. Meier, Portierung des Machmicrokernels auf MVME 188, Universität Erlangen-Nürnberg, IMMD IV, Studienarbeit, 1984

Ouster 82 J. K. Ousterhaug, Scheduling Techniques for Concurrent Systems, Proceedings of the Third International Conference on Distributed Computing Systems, Seiten 22-30, 1982

Sun 93 SunOS 5.3 Guide to Multithread Programming, Sun Microsystems, Mountain View, 1993