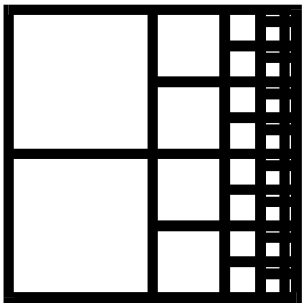
P A R L A B - M i t t e i l u n g e n

Nr. 02 / 95

ParLINUX
Ein paralleles Betriebssystem für Shared-Memory-Architekturen

Thomas Radke



Technische Universität Chemnitz-Zwickau

Fachbereich Informatik

Parallelrechnerlabor PARLAB

LS Rechnerarchitektur

Prof. Dr. W. Rehm

Inhaltsverzeichnis

0. Einleitung 3
1. Systemtechnische Grundlagen 3
    1.1. PC-AT-Kompatibilität 3
    1.2. Konfigurationsermittlung, Rücksetzen und Neustart eines Prozessors 3
    1.3. Prozessoridentifikation 4
    1.4. Interprozessorinterrupts 4
    1.5. Symmetrische Interruptzuweisung und -behandlung 4
2. Das Kernelmodell 5
3. Implementierung 6
    3.1. Der Bootprozeß 6
    3.2. Kernel-Locking 7
4. Praktische Erfahrungen 10
    4.1. Parallelität auf Prozeßebene 10
    4.2. Parallelität auf Threadebene 10
5. Zusammenfassung 12
Literaturverzeichnis 13

0. Einleitung

Dieser Beitrag stellt das Konzept eines LINUX-basierten parallelen Betriebssystems für Shared-Memory-Architekturen vor, lauffähig auf einem symmetrischen Multiprozessorsystem Compaq ProLiant 4000.

Zunächst werden im ersten Kapitel die systemtechnischen Grundlagen zur Programmierung der parallelen Hardware vermittelt. Darauf aufbauend entwickelt Kapitel 2 das Modell für einen LINUX-Kernel, der dieser Hardware entspricht. Auf die Implementierung des Modells wird im nächsten Kapitel näher eingegangen. Insbesondere erfolgt hier die Erläuterung des Bootvorgangs sowie des Kernel-Locking-Mechanismus'. Erste praktische Erfahrungen mit dem LINUX-Kernel stellt Kapitel 4 vor. Eine Zusammenfassung der Ergebnisse und die Ableitung zukünftiger Arbeiten werden im Schlußkapitel gegeben.

1. Systemtechnische Grundlagen

Der ProLiant 4000 ist Compaq's erster Multiprozessor-PC, der mit bis zu 4 Prozessoren ausgestattet werden kann. Informationen zur Inbetriebnahme der parallelen Hardware waren bisher jedoch nur für Dualprozessorsysteme erhältlich (siehe [Compaq94]), zu denen der ProLiant abwärtskompatibel ist. Aus diesem Grund beschränkt sich die gegenwärtige Kernel-Implementierung auf die Nutzung von nur zwei der drei vorhandenen Prozessoren. Das zugrundeliegende Konzept hingegen beinhaltet keine derartige Einschränkung und sollte deshalb ohne Probleme auf die Unterstützung aller drei Prozessoren erweiterbar sein, sobald entsprechende Informationen von Compaq vorliegen.

1.1. PC-AT-Kompatibilität

Um die Kompatibilität zum AT-Standard herkömmlicher Einprozessorsysteme zu gewährleisten, sorgt die Hardware des ProLiant dafür, daß beim Booten des Rechners nur ein Prozessor aktiv ist; dieser wird als bootstrap processor (BSP, im folgenden P1) bezeichnet. Alle anderen sind application processors (APs, im folgenden P2, P3, P4) und bleiben solange inaktiv, bis sie vom BSP explizit durch BIOS-Rufe initialisiert und gestartet werden.

1.2. Konfigurationsermittlung, Rücksetzen und Neustart eines Prozessors

Diese Funktionen werden über 3 verschiedene BIOS-Rufe bereitgestellt.

- Ermittlung der Systemkonfiguration

  Hiermit kann Prozessor P1 abfragen, ob ein zweiter Prozessor P2 im System vorhanden ist.

- Rücksetzen von P2

  P1 kann jederzeit P2 zurücksetzen. Dies bewirkt dessen Deaktivierung, d.h. die Unterbrechung seiner bisherigen Befehlsabarbeitung, die Initialisierung aller Register (analog einem Hard-RESET) sowie den Eintritt in eine Warteschleife bis zum Restart durch P1. Innerhalb der Warteschleife greift P2 nicht auf den Prozessorbus zu.

- Neustart von P2

  Diesem BIOS-Ruf wird als Parameter eine Restartadresse übergeben, die von P2 nach dessen Freigabe durch P1 angesprungen wird. Der erste Maschinenbefehl ab dieser Adresse muß ein Flag im Speicher setzen, um P1 einen erfolgreichen Neustart von P2 zu signalisieren.

1.3. Prozessoridentifikation

Jeder im System vorhandene Prozessor ist durch eine Prozessor-ID eindeutig gekennzeichnet und kann sich damit von anderen Prozessoren unterscheiden. Die Abfrage dieser Prozessor-ID erfolgt durch Lesen eines *WHO-AM-I*-Ports. Die Hardware stellt sicher, daß die Identifikation für jeden Prozessor verschieden ist.

1.4. Interprozessorinterrupts

Interprozessorkommunikation wird meist über die Benutzung gemeinsamer Speicherbereiche realisiert. Eine weitere Möglichkeit - insbesondere für asynchrone Kommunikation - bieten Interprozessorinterrupts (IPIs).

Ein Prozessor kann durch Setzen des IPI-Bits im Status-Port eines anderen Prozessors einen Interrupt bei letzterem auslösen. Die daraufhin angesprungene Interrupt-Service-Routine behandelt dann die Kommunikationsanforderung des Quellprozessors. Nach deren Beendigung muß das IPI-Bit im Status-Port wieder rückgesetzt werden, um weitere Interprozessorinterrupts zuzulassen.

1.5. Symmetrische Interruptzuweisung und -behandlung

Compaq's Dualprozessorsysteme verhalten sich asymmetrisch hinsichtlich der Zuordnung externer Interruptanforderungen an Prozessoren. Ausschließlich P1 ist in der Lage, externe Interrupts entgegenzunehmen und zu bearbeiten. Daraus ergibt sich eine entsprechende Disbalance - P1 hat die gesamte I/O-Last zu tragen.

Die neuartige Interrupt-Hardware im ProLiant 4000 besitzt eine symmetrische Betriebsart, bei der externe Interrupts an alle Prozessoren weitergeleitet werden können. Allerdings liegen noch keine Informationen von Compaq vor, wie die Interruptcontroller in diese Betriebsart umgeschalten und reinitialisiert werden müssen.

Anmerkung : Compaq entspricht mit der beschriebenen Funktionalität zum Ansprechen paralleler Systemkomponenten nicht der von Intel als Standard vorgeschlagenen Multiprozessorspezifikation [Intel94]. Es bestehen jedoch weitgehende Analogien zu den vorhandenen Funktionen. Somit ist eine Portierung der Compaq-spezifischen Abschnitte im Kernel auf andere Multiprozessorsysteme, die den Intel-Standard erfüllen, realistisch.

2. Das Kernelmodell

Die in Kapitel 1.1. und 1.4. vorgestellten Hardware-Systemeigenschaften implizieren zunächst ein asymmetrisches Multiprocessing.

Für die ersten Versuche in DOS wurde dies auch implementiert : ein Programm, welches eine Berechnung in zwei voneinander unabhängige Abschnitte partitionierte, veranlaßte P2 über den entsprechenden BIOS-Ruf, die jeweilige Funktion auszuführen und sich anschließend wieder zu deaktivieren (rückzusetzen). P1 übernahm dabei die Rolle des Dispatchers.

Innerhalb von DOS als Single-User-Single-Tasking-Betriebssystem ist dies die einzige Methode, die parallele Hardware überhaupt auszunutzen. Erst in Multitasking-Betriebssystemen wie z.B. UNIX oder Windows NT ergibt sich die Möglichkeit, mehrere Prozessoren zumindest auf Prozeßebene in die Benutzung einzubeziehen.

Prozesse können unabhängig voneinander laufen, wenn sie nicht miteinander kommunizieren bzw. sich synchronisieren. Dies ist in Multi-User-Systemen für Prozesse verschiedener Nutzer (immer ???) der Fall.

Jeder Prozeß belegt jedoch Ressourcen, die vom Betriebssystem verwaltet und über Systemrufe bereitgestellt werden. Da viele Ressourcen von mehreren (unabhängigen) Prozessen zugleich belegt werden (I/O-Geräte, Speicher), ergeben sich indirekte Abhängigkeiten und damit Einschränkungen bzgl. einer vollständig parallelen Abarbeitung. Aufgabe des Kernels in Multiprozessorsystemen ist es, diese Einschränkungen zu minimieren und Systemdienste konfliktfrei mit maximaler Effizienz zu erbringen.

An diesem Punkt stellt sich die Frage nach der geeignetsten Architektur eines solchen Multiprozessor-Kernels. In [Hwang93] werden drei mögliche Architekturen unterschieden :

1. Master Slave Kernel

   Ein Prozessor (der Master) ist prädestiniert zur Ausführung von Kernelcode; alle anderen arbeiten als Slaves ausschließlich Usercode ab. Sobald ein Prozeß einen Systemruf initiiert, geht seine weitere Ausführung auf den Master über. Ebenso werden alle externen Interrupts vom Master entgegengenommen und behandelt.

2. Floating Executive Kernel

   Prinzipiell sind alle im System vorhandenen Prozessoren befähigt, als Master zu arbeiten. Allerdings geschieht dies nicht vollständig parallel. Zu einem gegebenen Zeitpunkt wird entweder der gesamte Kernel durch einen einzelnen Prozessor belegt (single floating master), oder es existieren verschiedene Kernel-Subsysteme, die gleichzeitig von mehreren Prozessoren betreten werden können (multiple masters in different kernel subsystems). Diese Subsysteme bzw. der gesamte Kernel werden über Locking-Mechanismen geschützt.

3. Multithreaded Kernel

   Hier ist der Kernel nicht mehr monolithisch, sondern selbst in viele voneinander unabhängige Codeabschnitte (Threads) aufgeteilt und damit mehrfach wiedereintrittsfähig. Alle Prozessoren können gleichzeitig Kernelcode abarbeiten. Gemeinsam benutzte Datenstrukturen innerhalb des Kernels werden durch Synchronisationsmechanismen (Mutexes, Semaphoren, Monitore) vor inkonsistenten Zugriffen geschützt.

Wie bereits erwähnt, legt die z.Z. nur asymmetrisch nutzbare Systemhardware die Entwicklung eines Master-Slave-Kernels nahe, da hier der Implementierungsaufwand und die Komplexität am geringsten wären. Doch unter der Voraussetzung, bei Erhalt der entsprechenden Informationen von Compaq zukünftig die Symmetrieeigenschaften des ProLiant aktivieren zu können, wurde der zweite Ansatz für eine Kernelimplementierung gewählt. Zwar besteht hier immer noch der Flaschenhals eines monolithischen, nicht wiedereintrittsfähigen Kernels, doch verspricht dieser Ausgangspunkt wegen seiner symmetrischen Betrachtungsweise günstige Voraussetzungen für eine schrittweise Parallelisierung bis hin zur Entwicklung eines Multithreaded Kernel.

Bild 1 veranschaulicht das Modell eines Kernels mit exklusivem Master.

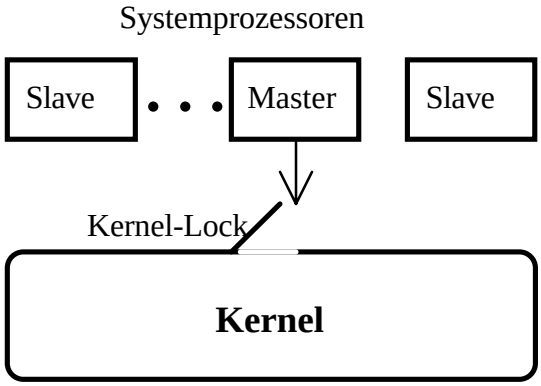


Bild 1 : Floating Executive Kernel mit Single Floating Master

3. Implementierung

Die Realisierung des vorgestellten Multiprozessor-Kernelmodells beruht auf der Implementierung von Threads in LINUX [Radke94]. Die Erweiterungen betreffen dabei den Vorgang des Bootstrapping sowie die Schnittstelle für den Wechsel vom User- in den Kernel-Modus und umgekehrt.

3.1. Der Bootprozeß

Wie bereits im ersten Kapitel beschrieben, ist beim Booten des Systems lediglich der BSP aktiv. Nach dem Power-On-Self-Test versucht das BIOS, ein Betriebssystem vom Bootsektor der Festplatte bzw. Floppy Disk zu laden und zu starten. Für LINUX bedeutet dies die Aktivierung des LINUX-Loaders. Er ermittelt zunächst verschiedene Daten zur Systemkonfiguration (Speichergröße, Typ von Festplatte, Grafikkarte und anderen Controllern) und lädt den gepackten Code des LINUX-Kernels in den Speicher nach.

In dieses Setup wurde die Abfrage nach Vorhandensein von APs sowie ggf. deren Aktivierung über die entsprechenden BIOS-Rufe eingefügt. Alle Prozessoren fahren dann an der gleichen Adresse im Setup fort, initialisieren ihre Register für die Umschaltung in den Protected Mode und springen zur Routine, die den komprimierten LINUX-Kernel entpackt. Nur der BSP führt diese Funktion aus; die APs warten solange, bis der BSP die Dekomprimierung beendet hat.

Dieses Warten erfolgt "aktiv" in einer Schleife, indem ständig der Wert eines Speicherflags abgefragt wird, bis der BSP diesen auf 0 setzt.

Anschließend werden die FPUs initialisiert, jeder Prozessor berechnet die Startadresse seines Stacks (anhand seiner Prozessor-ID), lädt sie in den Stackpointer und initialisiert das zugehörige Stackdatensegment. Diese Sequenz läuft vollständig parallel ab und entspricht der des originalen (Einprozessor-) LINUX-Setups.

Bevor die erste C-Funktion *startKernel ()* angesprungen wird, erfolgt die Einrichtung der global gültigen Page-Tabellen - wiederum durch den BSP, auf den die APs warten. In *startKernel ()* initialisiert der BSP die verschiedenen Kernel-Datenstrukturen unter Einbeziehung der im Setup ermittelten Systemkonfiguration (Interruptcontroller, periphere Einheiten u.a.).

Eine Synchronisation aller Prozessoren wird nochmals in der Funktion *mp_init ()* vorgenommen. Hier bestimmen sie ihre Urtask (TSS-Deskriptor in der GDT) und geben die Annahme externer Interruptanforderungen frei. Zu beachten ist, daß jeder Prozessor seine eigene Urtask erhält, wobei diese als Thread vom originalen LINUX-Urprozeß *idle* mit der PID 0 abgeleitet wird.

Nach der Umschaltung vom Kernelmode (Privilegierungsstufe 0) in der Usermode (Privilegierungsstufe 3) und der damit erfolgten Aktivierung der Urtasks wird vom BSP der erste "richtige" Prozeß *init* mit der PID 1 über den Systemruf *sys_fork ()* erzeugt. Anschließend führen alle Prozessoren in ihrer Urtask eine Endlosschleife aus, in der sie mittels *sys_idle ()* in den Kernel verzweigen und den Scheduler aufrufen, um ggf. eine neue Task abzuarbeiten. Mit Eintritt in diese Endlosschleife gibt es keine Unterscheidung mehr zwischen BSP und APs; alle Prozessoren werden von jetzt ab völlig gleich behandelt.

Die Verzweigung in den Kernel kann nur exklusiv von einem Prozessor zu einem gegebenen Zeitpunkt erfolgen. Der dazu implementierte Mechanismus wird im folgenden Abschnitt beschrieben.

3.2. Kernel-Locking

Die exklusive Ausführung von Kernelcode durch einen Prozessor zu einem gegebenen Zeitpunkt wird durch Kernel-Locking erreicht.

Alle möglichen Eintrittspunkte in den Kernel sind mit einer Sperre versehen, die nur jeweils ein Prozessor öffnen kann und erst bei Verlassen des Kernels wieder aufgehoben wird. Versucht ein Prozessor, bei blockierter Sperre in den Kernel zu verzweigen, dann führt er solange ein aktives Warten aus, bis die Sperre freigegeben wurde. Die Sperre selbst stellt ein Speicherflag dar, welches über atomare 80386-Operationen (locked Bit Test & Set bzw. locked Bit Test & Reset) manipuliert wird.

Der Umstand, daß im System mehrere Tasks aktiv sind, sich davon aber stets nur eine im Kernel befinden kann, erfordert die Gewährleistung einer transparenten Sichtweise des Kernels auf diese Tasks. Der originale LINUX-Kernel kennt nur eine aktuell in Abarbeitung befindliche Task, auf die er mit Zeigern auf deren Threadstruktur und die zugehörige Taskstruktur verweist. Diese Verweise sind in globalen Kernelvariablen enthalten. Für einen Multiprozessor-Kernel mit N Tasks (N = Anzahl der vorhandenen Prozessoren) ergibt sich damit ein N Elemente umfassendes Feld von Verweisen.

Da die Umstellung der Kernelquellen auf eine Feldindizierung enormen Editieraufwand bedeutet hätte, wurde die folgende Lösung bevorzugt : neben dem Feld von Verweisen bleiben die globalen Kernelvariablen als solche erhalten und werden bei Kerneleintritt vom jeweiligen Prozessor aus seinem Feldelement belegt. Beim Verlassen des Kernels erfolgt die Rückspeicherung der (vom Scheduler möglicherweise geänderten) Kernelvariablen in die entsprechenden

Einträge des prozessorzugehörigen Feldelements. In der bereits erwähnten Funktion *mp_init ()* werden bei Systemstart (d.h. noch vor dem ersten Eintritt eines Prozessors in den Kernel) die Feldelemente mit den Verweisen auf die jeweiligen Urtasks der Prozessoren initialisiert.

Der gesamte Kernel kann somit zunächst als ein einziger großer Monitor betrachtet werden, der - wie der originale LINUX-Kernel auch - stets nur eine aktuelle Task "sieht".

Jedoch reicht dieser einfache Mechanismus nicht aus. Aufgrund des möglichen Auftretens asynchroner externer Interruptanforderungen muß der Kernel für einen Prozessor, der sich bereits darin befindet, wiedereintrittsfähig sein. Bild 2 veranschaulicht diese Problematik.

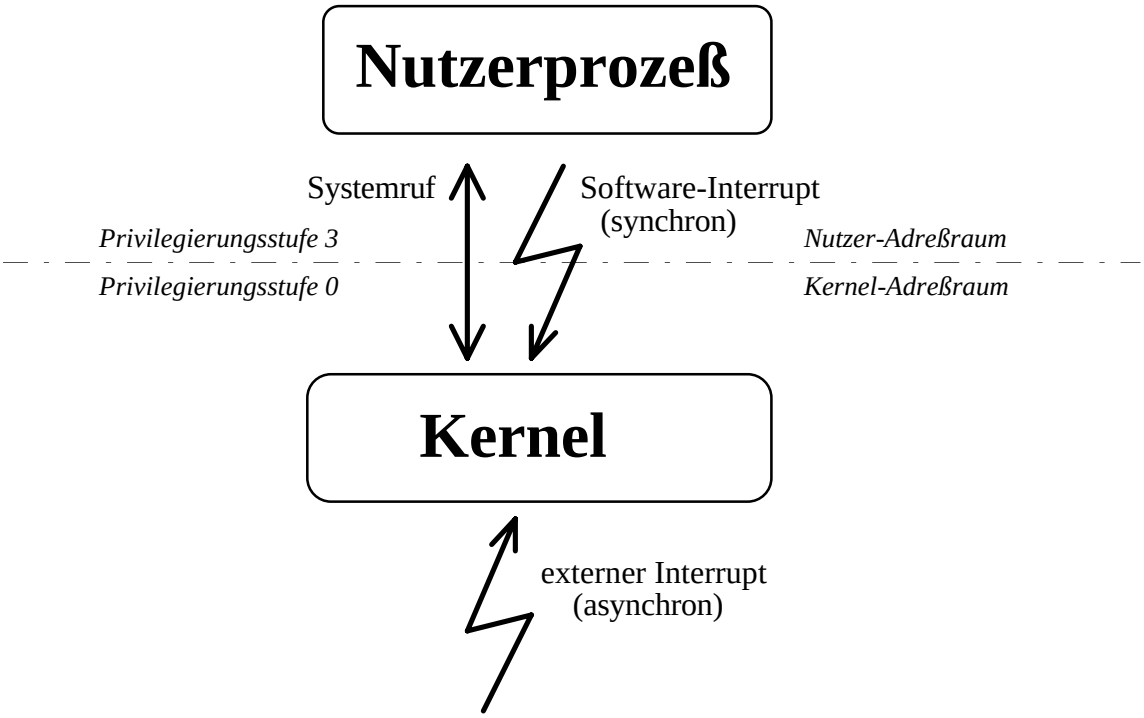


Bild 2 : Eintrittspunkte in den Kernel

Neben dem Speicherflag, das als Sperre dient, werden die Variablen *activeKernelProc* sowie *kernelCounter* notwendig, welche die ID des sich aktuell im Kernel befindlichen Prozessors sowie die Verschachtelungstiefe seiner rekursiven Kerneleintritte protokollieren. *kernelCounter* wird bei jedem Neueintritt in den Kernel erhöht und bei Verlassen dekrementiert. Anhand seines Wertes wird entschieden, ob die Kernelvariablen aus dem Verweisfeld geladen (beim ersten Kerneleintritt) bzw. dorthin gerettet und die Kernelsperre aufgehoben werden muß (beim letzten Kernelaustritt).

Die Aktualisierung all dieser Variablen ist nicht mehr mittels einer einzelnen atomaren Maschinenoperation möglich. Die Assemblercode-Implementierungen dieser Sequenzen sind in den Bildern 3 und 4 in einer C-ähnlichen Syntax vereinfacht dargestellt.

```
TRY_ENTER:      if   (AtomicBitTest&Set (kernelLock)  == UNLOCKED) {
                        activeKernelProc = myProcID ();
                        RestoreProcVariables ();
                } else if   (myProcID () != activeKernelProc) {
                        while (BitTest (kernelLock) == LOCKED)
                                ;                       /* busy wait */
                        goto TRY_ENTER;
                }
                kernelCounter++;
```

Bild 3 : Sequenz für den Eintritt in den Kernel

```
LEAVE_KERNEL :  if (--kernelCounter == 0) {
                        saveProcVariables ();
                        activeKernelProc = -1;      /* invalidate var. */
                        AtomicBitTest&Reset (kernelLock);
                }
```

Bild 4 : Sequenz für den Austritt aus dem Kernel

Zu erwähnen ist die Realisierung der aktiven Warteschleife mit anschließendem Neustart der Kerneleintritts-Sequenz. Die Bittest-Operation greift nicht modifizierend auf die Sperre zu, das Flag wird lediglich beim ersten Mal aus dem Speicher gelesen und befindet sich dann im Prozessorcache. Das Cache-Coherence-Protokoll gewährleistet die Invalidierung dieses Cacheeintrags beim Rücksetzen der Sperre durch den bis dahin aktiven "Kernelprozessor". Dieser besitzt bis dahin den vollen Zugriff auf den Speicherbus.

4. Praktische Erfahrungen

4.1. Parallelität auf Prozeßebene

Der Durchsatz eines Systems (Bearbeitung von Prozessen je Zeiteinheit) sollte sich mit der Zahl der verfügbaren Prozessoren N adäquat erhöhen, im Idealfall mit ihr skalieren. Inwieweit die vorgestellte Implementierung des LINUX-Kernels dieser Erwartung entspricht, wurde am folgenden Beispiel untersucht.

Die Quellen der Simulationsumgebung für Neuronale Netze SESAME umfassen 135 verschiedene C++-Dateien sowie zugehörige Header. Die Kompilierung dieser Quellen kann unabhängig und deshalb parallel erfolgen.

Das unter LINUX verfügbare GNU-make [GNU93] bietet dazu eine Option an, um die Anzahl gleichzeitig aktiver Compilerprozesse festzulegen (**--jobs***n* ..., standardmäßig gilt *n=1*). Diese Option wurde variiert und dabei die Dauer einer Übersetzung sämtlicher SESAME-Quellen bestimmt. Die gewonnenen Ergebnisse (gemessen mit dem Shell-Kommando *time*) enthält Tabelle 1.

Kernel-Version	Nutzerzeit [s]	Systemzeit [s]	CPU-Auslastung [%]
Original-LINUX mit make-Option --jobs1	649	119	99.6
Multiprozessor-LINUX mit make-Option --jobs2	323	90	98.6

Tabelle 1 : Zeiten für die Kompilierung von SESAME

Anmerkung : Wegen der derzeit noch asymmetrischen Betriebsart des zweiten Systemprozessors kann für diesen keine Auslastung bestimmt werden. Die Angaben beziehen sich daher alle auf P1.

Die Messungen wurden bei unbelastetem System durchgeführt.

Die annähernde Halbierung der Gesamtdauer beim Multiprozessor-Kernel läßt auf eine hohe Effizienz schließen. Allerdings waren auch die Rahmenbedingungen in diesem Beispiel sehr günstig : vollständige Parallelisierbarkeit der Aufgabe sowie aufwendige CPU-Aktivitäten innerhalb eines Compilerlaufs.

Engpässe im Kernel treten hier insbesonders bei der Speicherverwaltung (Allokieren bzw. Freigeben von Speicher) sowie in der Dateiverwaltung (Öffnen, Lesen, Schreiben, Schließen von Quell- bzw. Objektdateien) auf. Dies ergaben Profiling-Analysen zur Häufigkeitsverteilung von Systemrufen.

4.2. Parallelität auf Threadebene

Die Implementierung von Threads und entsprechender Synchronisationsmechanismen unter LINUX (siehe [Radke94]) wurden benutzt, um eine weitere Möglichkeit der Auslastung mehrerer Prozessoren zu testen. Der Overhead entsteht hier nicht durch gleichzeitige Systemrufe konkurrierender Prozesse, sondern durch die Synchronisation von Threads eines Prozesses und den damit verbundenen Verzweigungen in den Kernel. Der Aufwand zur Erzeugung und Been-

digung von Threads soll an dieser Stelle nicht in die Messungen einbezogen werden; er wurde bereits in [Radke94] betrachtet.

Als Testprogramm diente die Implementierung des in [Scholz95] beschriebenen Cholesky-Algorithmus. Er eignet sich insbesondere wegen seiner Parallelitätscharakteristik sehr gut zum Testen von Synchronisationsmechanismen. Gemessen wurden die Abarbeitungszeiten zur Berechnung der Matrix in Abhängigkeit von der Anzahl beteiligter Threads. Diagramm 1 zeigt einen Vergleich dieser Zeiten für verschiedene Multiprozessorsysteme, die an der TU Chemnitz-Zwickau zugänglich sind.

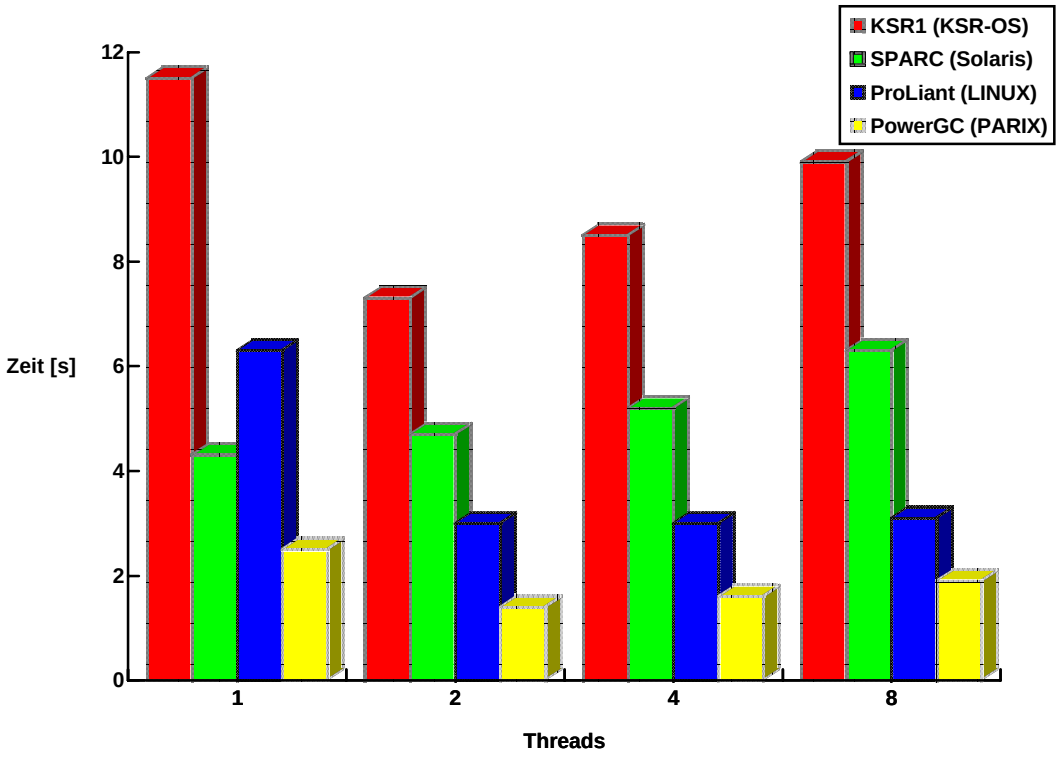


Diagramm 1 : Abarbeitungszeiten des Cholesky-Algorithmus in Abhängigkeit von der Threadanzahl

Anmerkung : Von den insgesamt 8 vorhandenen Prozessoren auf der KSR1 wurden für das Testprogramm nur 2 benutzt.

Unter PARIX ist keine separate Messung der Systemzeit möglich, die Nutzerzeit beinhaltet daher sowohl die Rechenzeit als auch den Aufwand zur Threadverwaltung und Synchronisation.

Die Tests auf dem SPARC-Compute-Server konnten nicht - wie bei den anderen Rechnern - bei unbelastetem System durchgeführt werden. Das Systemverhalten weicht deshalb evtl. vom erwarteten ab.

Deutlich sichtbar (außer bei Solaris) ist der Sprung in der Abarbeitungszeit bei einem bzw. zwei Threads, d.h. bei Einbeziehung des zweiten Prozessors. Eine weitere Erhöhung der Threadanzahl bewirkt durch den Anstieg der Synchronisationen ein starkes Anwachsen der Gesamtzeiten bei Solaris, so daß hier sogar schlechtere Ergebnisse als mit zwei Threads erzielt werden. Die KSR1, PARIX und LINUX zeigen den erwarteten mäßigen Anstieg der Systemzeiten.

5. Zusammenfassung

Auf der Grundlage von LINUX wurde ein multiprozessorfähiges Betriebssystem entwickelt und auf dem symmetrischen Multiprozessorsystem Compaq ProLiant 4000 implementiert. Das zugrundeliegende Kernelmodell nutzt die Symmetrieeigenschaften der Hardware vorerst nur implizit, da diese noch nicht aktiviert sind. Das Betriebssystem unterstützt Anwenderparallelität auf Prozeß- und Threadebene mit der Einschränkung, daß Systemrufe und externe Interruptanforderungen seriell bedient werden, da stets nur jeweils einem Prozessor erlaubt wird, über einen Locking-Mechanismus den Kernel zu betreten.

Erste Leistungsmessungen bescheinigen dem LINUX-Kernel eine hohe Effizienz im Vergleich mit anderen Multiprozessor-Betriebssystemen.

Besonders betont werden soll an dieser Stelle, daß sich LINUX sehr gut als Ausgangspunkt zur Entwicklung des vorliegenden Kernels eignete. Die gute Strukturierung und die teilweise bereits erfolgte Entkopplung verschiedener Kernel-Subsysteme (z.B. virtuelles Filesystem) bieten günstige Voraussetzungen für eine Parallelisierung des Kernels als nächste Aufgabe. Anzuzweifeln ist jedoch, ob sich der gegenwärtige monolithische LINUX-Kernel in eine Multithread-Architektur ähnlich der von Mach überführen läßt. Hier sind noch viele neue Ansätze notwendig, die zum größten Teil nur im Rahmen des gesamten LINUX-Projekts erbracht werden können.

Literaturverzeichnis

[Compaq94] *QuickFind* End-User 3Q94, Compaq Computer Corporation, 1994.

[GNU93] R. Stallman, R. McGrath, "GNU Make - A Program for Directing Recompilation", Edition 0.40, for make Version 3.63 Beta, January 1993.

[Hwang] K. Hwang, *Advanced Computer Architecture : Parallelism, Scalability, Programmability*, McGraw-Hill, New York, 1993.

[Intel94] *Multiprocessor Specification*, Version 1.1, Order Number 242016-001, Intel Corporation, 1994.

[Radke94] T. Radke, "Implementation von Threads und entsprechender Synchronisationsmechanismen unter LINUX", *PARLAB-Mitteilung 06/94*, Technische Universität Chemnitz-Zwickau, 1994.

[Scholz95] A. Scholz, "Entwicklung von Lern- und Demonstrationsprogrammen sowie Erstellung eines Dokumentationssatzes der Architektur und grundlegender Programmiermodelle des Parallelrechners KSR1", *Diplomarbeit*, Technische Universität Chemnitz-Zwickau, Februar 1995.