Beleg

zum Thema

Portierung des SCSI-Gerätetreibers von Linux auf L3

eingereicht von:	Frank Mehnert
geboren am:	18. Juli 1971
geboren in:	Altdöbern
Mat.-Nr.:	2133113

betreuender Hochschullehrer:	Prof. Dr. H. Härtig
Institut:	Betriebssysteme, Datenbanken und Rechnernetze
Lehrstuhl:	Betriebssysteme
Ausgabe:	1.5.1996
Abgabe:	1.11.1996

Dresden, 01.11.1996

Inhaltsverzeichnis

Abbildungs- und Tabellenverzeichnis

Abbildungen

Tabellen

I Einleitung

Ein wesentlicher Bestandteil eines Betriebssystems sind Gerätetreiber: Sie stellen dem System Dienste von Hardwarekomponenten zur Verfügung, indem sie Systemaufrufe in hardwarenahe Kommandos umsetzen. Treiber lassen dem Betriebssystem die Programmierschnittstellen verschiedener Komponenten gleich erscheinen. Programmieraufwand wird gesenkt, weil für ein Betriebssystem nur ein Treiber pro Hardwarekomponente notwendig ist, damit alle auf diesem System arbeitenden Programme die Dienste der Komponente ausnutzen können.

Aufgrund dieser wichtigen Aufgaben spielen die verfügbaren Treiber für ein Betriebssystem eine wichtige Rolle beim Kampf um Marktanteile. Ein Beispiel: Lange Zeit bot OS/2 von IBM nur wenig Hardwareunterstützung von Komponenten, die nicht von IBM stammten. Nicht zuletzt war dies ein Grund dafür, daß sich OS/2 erst seit Version 2.0 auf dem Massenmarkt etablieren konnte.

Diese Belegarbeit basiert auf der Diplomarbeit von René Stange [Stange 96]. In dieser Arbeit wird die Vorgehensweise zum Portieren eines Netzwerktreibers von Linux nach L3 beschrieben. Als Beispiel diente dazu der Netzgerätetreiber. Ziel war es, jeden unter Linux lauffähigen Netzgerätetreiber ohne Änderungen am hardwareabhängigen Teil unter L3 nutzen zu können. Diese Aufgabe wurde bewältigt, indem der Treiber in eine Emulation eingebettet wurde, die ihm das Bild eines Linux-Systems suggeriert. Der Treiber merkt somit nicht, daß er nicht unter Linux läuft und muß deshalb nicht verändert werden. Die Emulation wird durch einen eigenen Prozeß unter L3 dargestellt, der die Speicher- und Interruptverwaltung sowie die Schnittstelle zum L3-System übernimmt.

Die angesprochene Arbeit soll hier weitergeführt werden. Für L3 gibt es immer noch nicht ausreichende Treiber. Anlaß für die Entstehung dieser Arbeit war das Fehlen eines universellen SCSI-Treibers. Rudimentäre Unterstützung für Adaptec-1542-kompatible Hostadapter existiert zwar, ist aber direkt im Kern eingebunden und nimmt als Gerätetreiber unter L3 eine Sonderstellung ein, da er bereits zum Systemstart notwendig ist.

Hier soll vielmehr in gleicher Weise wie oben vorgegangen werden. Als Aufgabe steht weniger die Entwicklung eines L3-SCSI-Treibers im Vordergrund als vielmehr das Separieren und Herauslösen der SCSI-Komponente aus Linux sowie deren spätere Verwendung für Echtzeitaufgaben unter dem L4-System.

Als Einarbeitung in die Arbeit mit L3 sollte zuerst versucht werden, die Emulation aus [Stange 96] so zu erweitern, daß auch Netzwerkkarten mit Busmasterfähigkeit unterstützt werden. Zu Beginn dieser Arbeit war gerade eine erweiterte Version des Gerätetreibers für Netzkarten der 3c59x-Generation von 3com erschienen. Diese Alpha-Version sollte ihre Lauffähigkeit unter L3 beweisen. Dabei galt es gleichzeitig, Anpassungen zum neu erschienenden Linux-Kern 2.0 vorzunehmen.

Im Unterschied zum Netzgerätetreiber, der nur rudimentäre Fähigkeiten (Senden, Empfangen, Betriebsmodus) besitzt, stellt der SCSI-Treiber die Steuerung eines vollständigen Buskonzeptes dar. Hier können mit einem Treiber unterschiedlichste Geräte angesprochen werden, jedes mit anderen Übertragungsgeschwindigkeiten und -arten. Das SCSI hat bereits eine gewisse Standardisierung erfahren und gilt im PC-Bereich als Profi-Lösung. Zum Zeitpunkt der Entstehung dieser Arbeit ist der SCSI-2-Standard aktuell, der SCSI-3-Standard soll verabschiedet werden. Diese Aktualität war besonderer Ansporn für das Gelingen der Arbeit.

Bedanken möchte ich mich an dieser Stelle bei den Mitarbeitern des Lehrstuhls Betriebssysteme an der TU-Dresden, besonders bei Rene Stange für die mir entgegengebrachte Zeit bei der Einführung in seine Arbeit, Jean Wolter für seine Auskünfte und Anregungen über L3, Michael

Hohmuth für sein offenes Ohr bei Linux-Angelegenheiten und seine wichtigen Tips bei der Verfassung dieser Arbeit sowie Herrn Prof. Dr. H. Härtig für seine allgegenwärtige Hilfe und das in mich gesetzte Vertrauen.

II Grundlagen und Stand der Technik

II.1 Ein Blick nach vorn: Linux auf L4

Das Betriebssystem L3 hat mittlerweile den Nachfolger L4 bekommen. Es handelt sich hierbei um eine auf Intel-Machinen implementierte Microkernel-Architektur, die von Jochen Liedtke und anderen an der GMD entwickelt wurde [Liedtke 96]. Dieser als „schnellste Microkernel der Welt“ bezeichnete Kern stellt eine hervorragende Grundlage für ein Echtzeitsystem dar, weil er konsequent in Assembler programmiert wurde sowie ausführlichen Gebrauch der Speicherverwaltung moderner Prozessoren (i386 und höher) macht.

Linux ist in einer Arbeit von Michael Hohmuth und anderen an der TU-Dresden auf L4 portiert worden [Hohmuth 96b]. Im Linux-Kern wurden die wichtigsten Komponenten (Speicherverwaltung, Scheduler) durch den Microkern von L4 ausgetauscht. Ziel dieser Operation war die Schaffung eines echtzeitfähigen Betriebsystems mit breiter Softwareunterstützung. Das Team kann beachtliche Erfolge vorweisen: So laufen bereits Standardanwendungen wie X11R6 ziemlich stabil. Zu beachten ist hierbei, daß bis auf den Austausch der wesentlichen Komponenten am Linux-Kern selbst fast nichts verändert wurde.

Linux ist ein Betriebssystem, welches ständig von tausenden Programmierern verändert und verbessert wird. Linus Torvalds, der Schöpfer dieses Systems, wird dabei aus aller Welt unterstützt. Speziell die Entwicklung des Linux-*Kerns* erfolgt sehr dynamisch, ständig werden neue Features integriert und entstehende Fehler baldmöglichst korrigiert. Dennoch handelt es sich durchaus um ein stabil funktionierendes Programm. Zum Zeitpunkt der Bearbeitung war die Version 2.0.23 des Kernels aktuell [Linux 96]. Nachfolgende Ausführungen beziehen sich auf diese Version.

Das Fernziel dieser Belegarbeit soll die Lauffähigkeit des echtzeitfähigen SCSI-Treibers *auf* L4 *neben* Linux sein. Der Treiber ist für die Erledigung von Anfragen eines echtzeitfähigen Dateisystems verantwortlich und kann als Prozeß hoher Priorität über Art und Anzahl der benötigten Systemressourcen (Prozessorzeit, Hauptspeicher, Interrupts) entscheiden, ohne dabei von anderen Prozessen überstimmt zu werden.

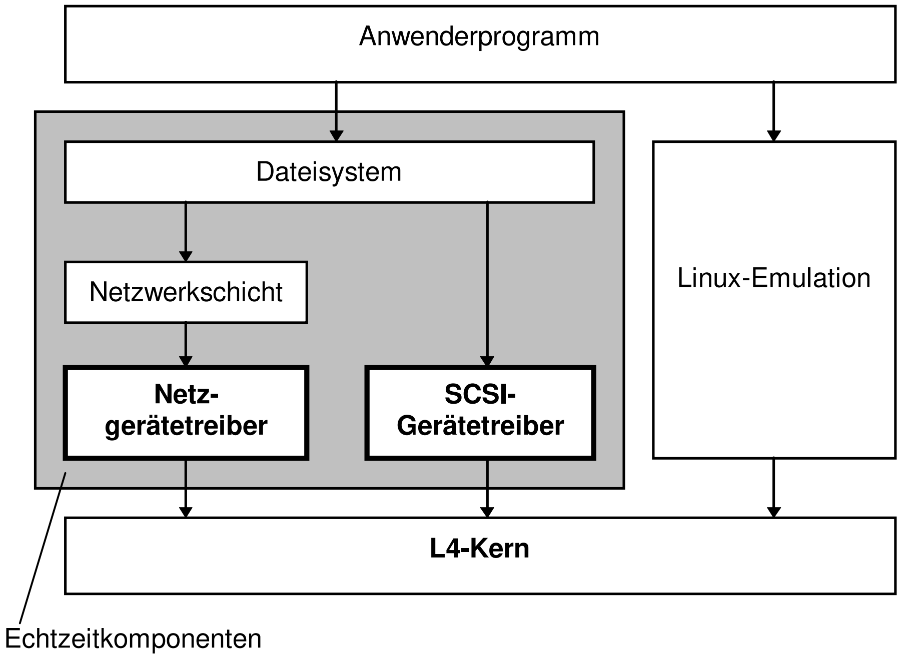

Abbildung II-1: Geplante Einbettung des echtzeitfähigen SCSI-Treibers

Die Linuxemulation stellt die Steuerung und Arbeitsoberfläche dar und hat die Rechte auf die restlichen Ressourcen. Damit soll eine kompromißlose Lösung für ein funktionierendes Echtzeitsystem gefunden werden, wo Anwendungen Systemressourcen *sicher* innerhalb einer festgelegten Zeitspanne zugesprochen bekommen.

II.2 Der SCSI-Standard

SCSI (*Small Computer System Interface*) ist seit Ende der 80er Jahre die professionelle Lösung zum Verbinden verschiedenster Geräte wie Festplatten, CD-ROM-Laufwerke, Bandlaufwerke, Scanner und Drucker an einem parallelen Bus mit 8, 16 oder 32 Datenleitungen [Schmidt 93]. Jedes Gerät hängt an einem nichtverzweigten Bus und kann mit seiner ganz speziellen Datenrate und Übertragungsart angesprochen werden, sodaß ein CD-ROM-Laufwerk niemals eine Festplatte ausbremst, wie das z.B. beim ATA-Interface der Fall ist.

Der aktuelle Standard SCSI-2 erlaubt Datenübertragungsraten von 10Mbyte/s (8 Datenleitungen) bzw. 40Mbyte/s (32 Datenleitungen) im synchronen Modus. Unter SCSI-3 sind Datenraten bis zu 80Mbyte/s (32 Datenleitungen, halbierte Zykluszeit) möglich. Solche Übertragungsraten kann kein aktuelles SCSI-Gerät *dauerhaft* erbringen. Der Bus wird deshalb erst bei parallelem Betrieb von vielen Geräten an einem Hostadapter voll ausgereizt. Mögliche Anwendungen sind RAID-Systeme.

An einer Datenübertragung auf dem SCSI-Bus sind immer mindestens ein *Initiator* (Auftraggeber) und ein *Target* (Diensterbringer) beteiligt. Über einen festen Satz von Befehlen kann ein Initiator mit jedem Gerät kommunizieren, ein geräteabhängiger Teil kann vom Hersteller nach Bedarf implementiert werden und geht auf die Besonderheiten jeder Geräteart ein. Praktisch kann jedes Gerät am Bus zum Initiator werden (der Hostadapter zählt als ein Gerät). Damit sind auch Übertragungen zwischen zwei SCSI-Geräten denkbar.

Alle Abläufe auf dem SCSI-Bus setzen sich aus acht Busphasen zusammen [Schmidt 93], siehe **Tabelle II.1**. Der dahinter zu erkennende strenge Ablauf bedeutet zwar einerseits einen großen Verwaltungsaufwand (was sich nicht zuletzt im Preis gängiger SCSI-Systeme niederschlägt) und einen relativ großen Anteil der Befehle am Gesamtdatenvolumen, läßt aber dadurch den gleichzeitigen Betrieb von verschiedensten Geräten an einem Bus zu.

Nr.	SCSI-Busphase		Erläuterung
1	Default	*Bus-Free*	Normaler Zustand - kein Gerät belegt den Bus
2	Protokoll	*Arbitration*	Ein oder mehrere Initiatoren melden Bedarf für den Bus an. Rangfolge nach ID (ID7 most, ID0 least)
3		*Selection*	Ausgewählter Initiator spricht sein Target an
4		*Reselection*	Ein Target, das den Bus während eines laufenden Commandos freigegeben hatte, nimmt wieder Verbindung zum Initiator auf, nachdem neu arbitriert wurde
5	I/O	*Command*	Übertragung eines Kommandobytes
6		*Data*	Übertragung eines Datenbytes
7		*Message*	Target hinterlegt/holt Nachricht beim/vom Initiator
8		*Status*	Target beendet SCSI-Kommando und teilt Initiator mit, ob das Kommando erfolgreich war.

Tabelle II.1: Zustände des SCSI-Busses

Besondere SCSI-Eigenschaften

Nachfolgend sollen wichtige Features von SCSI-Systemen erklärt werden. Das Verständnis wird bei der Besprechung der verschiedenen Lösungsvarianten vorausgesetzt.

Aufwendiges Busprotokoll - Lange Ausführungszeiten

Aufgrund des relativ komplexen Busprotokolls des SCSI-Standards, kann es ziemlich lange dauern, bis ein SCSI-Befehl vollendet ist. Die Prioritätsreihenfolge der einzelnen SCSI-Geräte am Bus kann durch die ID-Nummer gesteuert werden: Je höher die Nummer, desto höher die Priorität des Gerätes. Da bei jeder Datenübertragung mehrere Busphasen durchlaufen werden, kann selbst die Übertragung eines einzigen Datenbytes mehrere Millisekunden in Anspruch nehmen. Hinzu kommt der Verwaltungsaufwand für die DMA-Steuerung. Hier ist SCSI dem einfacheren ATA-Interface klar unterlegen.

Deshalb ist es unter SCSI ein angesagtes Ziel, die **Anzahl der Kommandos** zu **minimieren** und die zu **übertragenen Daten pro Kommando** zu **maximieren**.

Disconnect/Reconnect

Speziell neuere SCSI-Komponenten und Hostadapter unterstützen den Mechanismus *disconnect/reconnect*: Eine Target, z.B. Festplatte, erhält einen I/O-Befehl vom Initiator. Die Platte kann dann sofort den Bus freigeben, während der Lesekopf zur richtigen Position auf dem Speichermedium fährt und die ersten Sektoren in den Cache gelesen werden. Während dieser Zeit können andere Übertragungen auf dem Bus stattfinden. Nachdem die Festplatte bereit ist, die Daten zu liefern, bemüht sie sich per Interrupt um einen *reconnect*, um die Daten ordungsgemäß abzuliefern. Diese sehr wichtige Fähigkeit bewahrt andere Geräte am Bus vor langen Wartezeiten und ermöglicht ein Quasi-Multitasking auch in Mitwirkung sehr langsamer Geräte (Streamer).

Scatter/Gather

Dieser Fähigkeit von Hostadaptern erlaubt es, mehrere gleichartige SCSI-Operationen in einen Zugriff zusammenzufassen. Soll z.B. eine SCSI-Schreiboperation **m** logisch aufeinanderfolgende Blöcke in **n** physikalisch *nicht* aufeinanderfolgende Buffer lesen, so müßte diese Operation ohne Scatter/Gather-Fähigkeit des Hostadapters in **n** Einzeloperationen zerlegt werden, da dann pro SCSI-Operation nur die Angabe einer einzigen physikalischen Adresse möglich ist. Mit Scatter/Gather wird eine sogenannte scatterlist gebildet, die Länge und Position aller zu beschreibender Puffer im Speicher aufnimmt. Die beschriebene Eigenschaft ist für SCSI-Operationen besonders wichtig, da jede neue SCSI-Operation einen großen Synchronisationsaufwand nach sich zieht und somit die Anzahl an Aufträgen gering gehalten werden sollte.

Übertragungsarten

Busübertragungen von Daten können *asynchron* oder *synchron* erfolgen, wobei letztere Art aus Performancegründen vorzuziehen ist. Steuerbefehle werden aus Kompatibilitätsgründen immer asynchron übertragen, das anzusprechende Target kann dann mit dem Initiator eine andere Übertragungsart vereinbaren. Ältere SCSI-Geräte beherrschen aber meist nur diese Übertragungsart.

Die Übertragung zwischen SCSI-Hostadapter und Hauptspeicher erfolgt meist per DMA. Dabei wird die CPU vom Übertragen großer Datenmengen freigestellt und kann in der Zwischenzeit andere Arbeiten erledigen (siehe **II.3: Busmastering**).

II.3 Gerätetreiber - Tor zur Hardware

Bei näherer Betrachtung stellt sich ein Gerätetreiber wie ein Abteilungsleiter dar: Für seinen Bereich ist er gänzlich allein verantwortlich, muß höheren Stellen Rede und Antwort stehen, verteilt eingehende Ressourcen gleichmäßig unter seinen Untergebenen und hat nur sehr spezielles Wissen von der Welt außerhalb seiner Dienststelle.

Ein Gerätetreiber ist speziell für seine Hardware angepaßt und muß diese vor höher liegenden Schichten repräsentieren. Dafür bereitet er die von der Hardware kommenden Daten auf, Befehle des Systems wandelt er in Steuerbefehle der Hardware um. Falls die Hardwarekomponente einen Wunsch hat (Interruptanforderung, I/O-Bereich-Anforderung), leitet der Gerätetreiber diesen an das System weiter, wo entschieden wird, dem Wunsch zu entsprechen oder zu widersprechen.

Aus dieser Sichtweise läßt sich ein allgemeines Schema für Gerätetreiber entwerfen. In Ergänzung zu [Stange 96] soll hier auf weitere wichtige Gegebenheiten eingegangen werden.

Übertragungsarten

Daten können auf zwei verschiedene Arten zwischen externen Komponenten und Hauptspeicher transportiert werden. Sie unterscheiden sich bezüglich Implementierungsaufwand und Leistungsfähigkeit.

Programmed I/O

Diese Übertragungsart ist sehr häufig anzutreffen. Ein Datum wird mittels I/O-Befehl der CPU (in/out) auf den Datenbus gelegt und entsprechend an ein externes Gerät geleitet. Ganze Datenblöcke werden mit Blockbefehlen übertragen. Die Arbeit erledigt in jedem Fall die CPU, jedes Datum muß einzeln übertragen werden. Das hat weitreichende Konsequenzen:

- Eine I/O-Übertragung darf nicht schneller erfolgen, als es die Hardware erlaubt. Speziell ältere Einsteckkarten können Daten nicht beliebig schnell lesen/schreiben und fügen bei Überschreitung einer Erhohlzeit im **µs-Bereich** *keine* Wartezyklen ein. Bei Bewegung von Datenblöcken kann es somit zu Datenverlust kommen. Dieses Problem tauchte erstmals bei Einführung des i486 von Intel auf [Stiller 93]. Diese CPU kann sehr viel schneller auf dem I/O-Bus operieren, als ihr Vorgänger, der i386. Mainboardhersteller reagierten darauf und ermöglichten ein Einfügen von Wartezyklen bei I/O-Operationen (Option „I/O-Recovery-Time“ im CMOS-Setup) durch den Chipsatz. Diese Einstellung gilt aber für alle I/O-Operationen und bremst auch Geräte aus, die in der Lage wären, schneller zu arbeiten.
- Es ist meist sehr gefährlich, I/O-Block-Operationen durch Interrupts zu unterbrechen: Tritt während eines solchen Transfer ein Interrupt auf, kann es zu Datenverlust kommen. Deshalb werden maskierbare Interrupts während Blockoperationen verboten, so z.B. auch beim ATA-Treiber von Linux 2.0 [Linux 96]. Folge: Schlechtere Multitasking- und Echtzeiteigenschaften.
- Die CPU-Belastung ist bei I/O-Operationen sehr hoch, alle Daten müssen durch diesen Flaschenhals. Die I/O-Daten stellen einen weiteren Byte-Strom durch die CPU dar und vermindern die Verfügbarkeit der Systemressource „CPU-Zeit“.

Busmastering

Einige Nachteile von Programmed I/O kann man mit Busmastering umgehen. Hierbei wird die Aufgabe des Datentransfers einer externen DMA-Einheit (*direct memory access*) übertragen. Die CPU reicht an diese Adresse und Länge des Datenblocks weiter und gibt dann den Startbefehl. Während nun die DMA den Datentransfer besorgt, kann sich die CPU anderen Aufgaben widmen. Dadurch ist bei Übertragungen dieser Art die CPU-Belastung wesentlich geringer als bei Programmed I/O. Insbesondere brauchen hier die Hardwareinterrupts nicht gesperrt werden, was sich positiv auf die Echtzeitfähigkeit der Komponente auswirkt. Wenn die DMA-Einheit die Arbeit beendet hat, signalisiert sie dieses der CPU durch einen Interrupt.

Ein Nachteil des DMA-Übertragungsverfahrens soll hier nicht unerwähnt bleiben: Als Tribut an die Kompatibilität verhalten sich die DMA-Bausteine vieler moderner Rechnersysteme genauso wie die in den ersten IBM-PCs. Der Adreßraum dieser DMA-Bausteine ist auf 16MB begrenzt, der Übertragungspuffer darf keine physische 64kB-Segmentgrenze überschreiten[1]. Bei Zugriff auf physische Adressen über der 16MB-Grenze muß eine Zwischenpufferung im unteren 16MB-Bereich erfolgen, da die Adressierung dieser alten Komponente nur mit 24 Bit stattfinden kann. Bereitstellung des Puffers bedeutet zusätzlichen Speicherverbrauch, Kopieren von/zum Zwischenspeicher heißt zusätzliche Taktzyklen der CPU. Die 16MB-Hürde wurde erstmalig mit der EISA-Architektur überwunden, verbesserte Hardware ohne diese Beschränkung ist im Low-Cost-Markt aber erst seit Einführung des PCI-Systemkonzepts anzutreffen.

Ein Schritt zur Seite

An diesen beiden Übertragungsarten erkennt man gut allgemeine Prinzipien der PC-Architektur. Das **Kompatibilitätsprinzip** ist eines der wichtigsten. Wenn irgend möglich, muß eine Hardwarekomponente, die im IBM PC Baujahr 1980 funktionierte auch im heutigen PC

[1] Ist der DMA-Puffer 16kB lang (0x4000), würde er z.B. bei einer Adreßlage von 0x5C100 die 64kB-Grenze bei 0x60000 schneiden. Das ist bei alten DMA-Bausteinen, wie sie in der ISA-Architektur verwendet werden, unzulässig.

mit Pentium Pro funktionieren[2]. Eine alte Festplatte mit 10MB Kapazität wird auch noch an einem modernen Controller mit ATA-Schnittstelle arbeiten. Dafür werden viele **Anpassungen** vorgenommen: Zusätzliche Wartezyklen für langsame externe Komponenten (die sich meist auch auf schnellere auswirken), Koexistenz von Komponenten verschiedener Busbreite und Geschwindigkeit am selben Bus, Emulation alter Betriebsmodi u.s.w.. Der Prozessor ist immer noch das Bauteil, das alle Daten verarbeiten muß und einen allgemeinen Flaschenhals darstellt. Um ihm soviel Arbeit wie möglich abzunehmen, wird die **Peripherie immer intelligenter**. Grafikkarten erhalten Beschleunigerfunktionen und Eigenintelligenz. Controller für die Ansteuerung von externen Datenspeichern lassen die Daten nicht von der CPU, sondern von der DMA übertragen (*Busmastering*). Daten werden sehr oft zwischengespeichert (*Caching*).

Gleichzeitig legt man wert auf absolut **minimale Kosten**: Fast alle Ressourcen eines PCs sind arg begrenzt. Verschiedene Geräte müssen sich einen Interrupt teilen, manchmal sogar so, daß nicht beide Geräte zugleich in Betrieb sein dürfen.

Auf diese besonderen Umstände gilt es gerade bei der Entwicklung von hardwarenahen Programmen einzugehen. Weil die PC-Architektur kein durchgängig kompromißloses Konzept darstellt, muß die Software intelligent implementiert werden und sich der Hardware anpassen.

II.4 Gerätetreiber unter L3

Der SCSI-Gerätetreiber

Unter L3 ist allgemein der Festplattentreiber und speziell der SCSI-Treiber Bestandteil des Kerns. Grund dafür ist, daß die Dienste dieser Schicht sehr zeitig beim Bootvorgang benötigt werden, das System wird über diesen Treiber von der Festplatte geladen. Deshalb nimmt hier der SCSI-Treiber eine Sonderstellung ein.

Die Systeminitialisierung erfolgt unter L3 so: Zuerst wird geprüft, ob im System mindestens eine Festplatte mit IDE-Interface eingebaut ist. Falls ja, werden soviel Prozesse gestartet, wie Partitionen auf IDE-Festplatten vorhanden sind. Dann wird der SCSI-Treiber ebenfalls gestartet und alle Partitionen auf SCSI-Platten überprüft. Der Backingstore Manager (bsdman) findet nun die Prozesse BSD1...BSDx vor und inititialisiert diese. Dann folgen weitere Schritte der Systeminitialisierung.

Nachdem der bsdman alle Swap-Geräte unter Kontrolle hat, ist es nicht möglich, nachträglich weitere Geräte einzubinden.

II.5 Gerätetreiber unter Linux 2.0

Grundlegendes

Aufgrund des monolithischen Kerns fällt es sehr schwer, einzelne Treiber zu isolieren. Jede Funktion kann grundsätzlich jede andere Funktion aufrufen. Erschwert wird die Strukturierung durch ständige Veränderungen, die am Kern vorgenommen werden. Oft wird es nötig, bestimmte Abschnitte komplett neu zu schreiben, weil einfach zu viele Änderungen daran vorgenommen wurden. Zum aktuellen Zeitpunkt spielt man mit dem Gedanken, den SCSI-Teil von

[2] Tatsächlich gilt das Gesagte auch für Software. Heutige Hochleistungs-PCs mit Chips der neuesten Intel-Generation beherrschen immer noch den Realmodus, damit z.B. MS-DOS funktioniert. Solcher Software wird einfach ein superschneller IBM-PC vorgegaukelt. Andererseits liegen damit viele Neuerungen der Hardwarearchitektur brach.

Linux neu zu implementieren. Grund sind immer neue Änderungen am Treiber, die den Verlust der Übersichtlichkeit bedeuten.

Andererseits stellt gerade die große Dynamik des Linux-Projektes eine reiche Auswahl an Gerätetreibern sicher. Alle Informationen über Linux in diesem Dokument beziehen sich auf die Version 2.0.23, welche momentan (Oktober '96) als letzte stabile Version des 2.0er Kerns gilt. Beinahe alle Informationen wurden aus dem Quelltext entnommen, Hinweise finden sich aber auch in [Beck 95] und [Johnson 95].

Zum weiteren Verständnis wird auch hier auf [Stange 96] verwiesen. Dennoch einige allgemeine Worte: Für der Programmierer stellt sich der Linux-Kern als eine Ansammlung von C- und Assembler-Quelltexten dar. Für viele Funktionen aus Standard-C-Bibliotheken gibt es hier Entsprechungen.

Ein Linux-Geräetetreiber läuft im selben Adreßraum wie der Kernel und kann somit auf alle Funktionen und Variablen des Kernels zurückgreifen. Die Schnittstelle zum Kernel besteht aus Funktions-Prototypen und Datenstruktur-Definitionen in C-Headerdateien. Ein Treiber kann **fest** in den Kernel **eincompiliert** oder in **Modulform** bei Bedarf während der Laufzeit des Systems nachgeladen werden. Ein Hinzufügen eines Treibers bedeutet dabei im ersten Fall immer eine Neucompilierung des gesamten Kernels, im zweiten Fall nicht. Immer mehr Gerätetreiber bieten die Möglichkeit einer nachträglichen Konfiguration mittels Parameter bei Systemstart oder Systemaufrufe (*ioctls*).

Hardwareinterrupts

Das PCI-System kennt neben normalen Hardwareinterrupts sogenannte *Shared* Interrupts. Hierbei teilen sich mehrere Hardwarekomponenten einen Interrupt. Bei Auftreten einer Unterbrechung prüft die Interruptbehandlungsroutine, für welches Gerät der Interrupt gedacht ist und leitet ihn entsprechend weiter. Seit Version 2.0 wird diese Arbeitsweise auch in Linux unterstützt. Sie dient vor allem der Einsparung der knappen Ressource „Hardwareinterrupt", von der uns der PC nur 15 zu bieten hat.

Arten von Geräten

Linux kennt verschiedene Arten von Geräten: *character, block, SCSI, net, sound*. SCSI-Geräte sind dabei Blockgeräte, können aber auch als Charaktergeräte angesprochen werden (Komponente generic).

Netzgerätetreiber

Netzkarten stellen reine zeichenorientierte Geräte (*character devices*) dar. Aufgrund des zu bewältigenden konstanten Datenstroms, stellt sich *Caching* hier anderes dar, als z.B. bei Blockgeräten. Alle Zwischenspeicher haben den Aufbau von Warteschlangen nach FIFO-Prinzip. Auf dem Weg zu Linux 2.0 wurde die Netzwerk-Schicht optimiert, was allerdings kaum Auswirkung auf die verwendeten Hardwaretreiber hat. Einzig Aufbau und Verwaltung der universellen *Netzpuffer* haben sich leicht geändert (vergl. [Stange 96]):

```
struct sk_buff_head {
     struct sk_buff *next;
     struct sk_buff *prev;
     unsigned int size;
};

struct net_buff {
     struct sk_buff *next;
     struct sk_buff *prev;
     struct sk_buff_head *list;

     struct device   *dev;
     unsigned long   len;

     unsigned short protocol;
     unsigned short  truesize;

     unsigned char *head;
     unsigned char *data;
     unsigned char *tail;
     unsigned char *end;
};

struct sk_buff *alloc_buff(unsigned int length);
void free_buff(struct sk_buff *buff);
void queue_head_init(struct sk_buff_head *list);
void queue_head(struct sk_buff_head *list_, struct sk_buff *buff);
void queue_tail(struct sk_buff_head *list_, struct sk_buff *buff);
struct sk_buff *dequeue(struct sk_buff_head *list_);
int queue_empty(struct sk_buff_head *list);
```

Abbildung II-2: Neue sk_buff-Struktur mit Hilfsroutinen

Insbesondere die Veränderung des Aufbaus muß in der Implementierungsphase berücksichtigt werden.

Blockgerätetreiber

Ein Blockgerät zeigt sich dem System als eine lineare Ansammlung von logischen Blöcken gleicher Größe. Die logische Blockgröße kann bei verschiedenen Geräten differieren und ist abhängig von der physikalischen Blockgröße, mit der die Medien hardwareseitig formatiert sind. CD-ROM-Laufwerke besitzen eine logische und physische Blockgröße von 2048 Byte, Festplatten eine physische Blockgröße von 512 Byte, aber eine logische Blockgröße von 1024 Byte. Manche MODs (*magnet optical drives*) haben auch eine physikalische Blockgröße von 2048 Byte und können in diesem Fall von Linux nicht angesprochen werden.[3]

Der Datenaustausch mit den Blockgeräten erfolgt unter Linux über eine einzige Schnittstelle, die Blockgeräteverwaltung. Die Funktion `ll_rw_block` *(all read/write block)* nimmt dabei alle Lese- und Schreibanfragen an jegliche Blockgeräte entgegen. Diesen Dienst nutzen das VFS *(virtual file system)*, die Direktansteuerung für Blockgeräte und die Swapping/Paging-Mechanismen.

[3] SCSI-MOD's werden unter Linux wie Festplatten angesprochen, der verantwortliche Treiber *(sd - scsi disk)* verarbeitet unter Linux 2.0 nur Blockgrößen von 1024 Byte. Grund dürften die für beschreibbare Medien verwendeten Dateisysteme sein, die von eben dieser Annahme ausgehen.

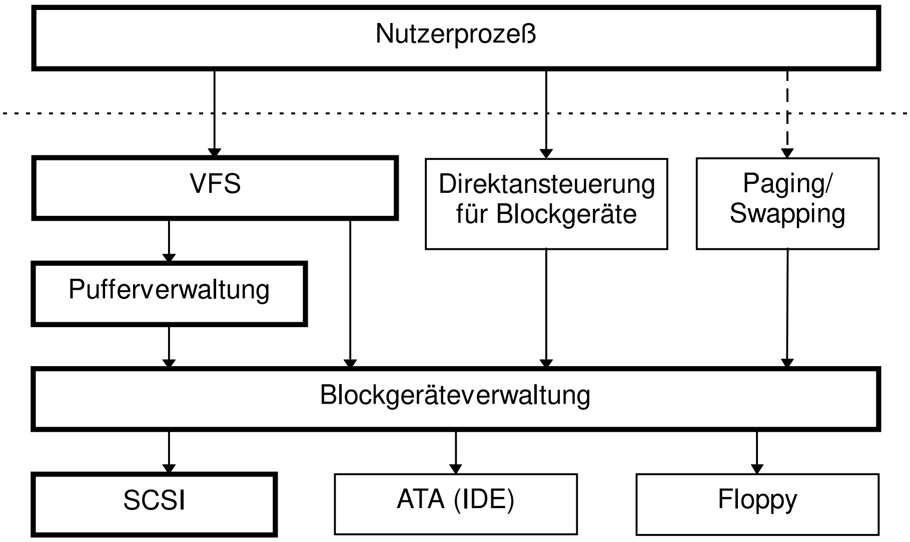

Abbildung II-3: Blockgeräte im Linux-Kern

Eine Steuerung des *Verhaltens* der Blockgerätetreiber ist über ioctl-Kommandos (*I/O control*) möglich. Damit wird eine CD-Lade verriegelt oder kann darauf reagiert werden, wenn ein Wechselmedium aus dem Laufwerk genommen wurde.

Gekennzeichnet werden Blockgeräte durch *Major-* und *Minor-*Gerätenummern. Die Major-Gerätenummer bezieht sich auf die Geräteklasse, die Minor-Nummer auf die physisch vorhandenen Laufwerke und deren logische Einteilung in Partitionen. Pro SCSI Festplatte sind maximal 16 Partitionen vorgesehen. Minor- und Major-Nummer haben unter Linux momentan eine Datenbreite von jeweils acht Bit.

Der SCSI-Gerätetreiber

Die Unterstützung für SCSI-Geräte hat unter Linux mittlerweile einen hohen Stand erreicht. Es wird eine große Zahl von Hostadaptern unterstützt, eine Reihe von Features, z.B. nachträglich an- und abzumeldende SCSI-Geräte [SCSI-HowTo], sucht man unter anderen Betriebssystemen erfolglos.

Der SCSI-Treiber von Linux besteht aus drei Schichten: Der unteren, mittleren und oberen Schicht.

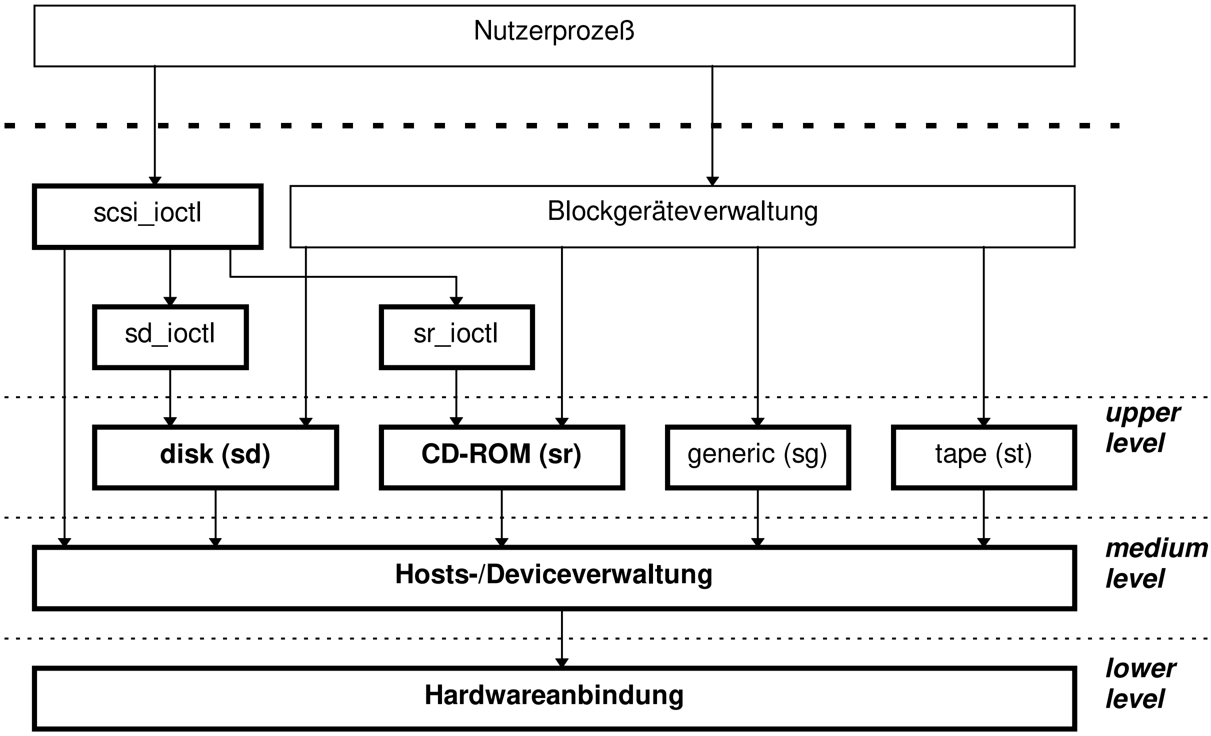

Abbildung II-4: Der Linux-SCSI-Gerätetreiber aus der Sicht des Nutzerprozesses

- Die untere Schicht *(lower level)* ist für die hardwaremäßige Anbindung des Hostadapters an das System verantwortlich und die einzige Schicht, die direkt mit der Hardware kommuniziert. Diese Schicht wird an den jeweiligen Hostadapter angepaßt.
- Die mittlere Schicht *(medium level)* ist für das Erkennen und Einbinden der Hostadapter sowie für generische Operationen über den SCSI-Geräten verantwortlich. Beim Inititalisierungsvorgang versucht sie der Reihe nach, alle in den Kernel compilierten Treiber zu starten. Ein Gerätetreiber, der seinen Hostadapter findet, registriert sich bei der mittleren Schicht per `scsi_register`. Damit ist er der dieser Schicht bekannt und kann angesprochen werden. Umgekehrt fürhrt das Auffinden eines SCSI-Gerätes an einem Hostadapter zum Aufruf von `scsi_register_device`.
- Die obere Schicht *(upper level)* ist für die Behandlung der verschiedenen SCSI-Geräte verantwortlich. Diese Schicht besteht wie die unterste Schicht aus mehreren Teilen, die dem System die SCSI-Geräte unter ihren Namen zugänglich machen: sd *(SCSI disk)*, sr *(SCSI cdrom)*, st *(SCSI tape)*, sg *(SCSI generic)*.

Datenfluß im Blockgerätetreiber

Folgend wird ausführlich der Weg der Daten über die Bufferverwaltung ausgehend von einer Leseanforderung eines Nutzerprozesses an eine SCSI-Komponente beschrieben. Diese Beschreibung dient dem Verständnis der Vorgehensweisen in der Entwurftsphase, die Informationen wurden ausschließlich aus den Quelltexten des Kerns entnommen.

VFS

Eine Lese-/Schreibanforderung von Daten einer Datei wird an das VFS (*virtual file system*) übermittelt. Diese Komponente möchte ein oder mehrere Datenblöcke vom Blockdevice lesen oder auf selbiges schreiben. Es stellt diese Forderung an die *Bufferverwaltung* durch Angabe von **Gerätenummer**, logischer **Blocknummer** und **Blockanzahl**.

Bufferverwaltung - Cache dynamischer Größe

Es werden ständig Teile des Hauptspeichers für die Pufferverwaltung reserviert. Ein Puffer entspricht genau einem logischen Block auf einem bestimmten Gerät (**Abbildung II-5**). Das bedeutet insbesondere: Soll ein Block eines Blockgerätes geschrieben werden, so muß, wenn es sich um den ersten Zugriff auf diesen Block handelt, dafür erst ein entsprechender Buffer eingerichtet werden, in den dann die zu schreibenden Daten abgelegt werden.

```
struct buffer_head {
      unsigned long b_blocknr;          /* Blocknummer */
      kdev_t b_dev;                     /* Gerätenummer */
      kdev_t b_rdev;                    /* reales Gerät */
      unsigned long b_rsector;          /* reale Pufferpostition auf Medium */
      struct buffer_head *b_next;       /* Hashliste */
      struct buffer_head *b_this_page   /* Pufferliste auf einer Seite */
      unsigned long b_state;            /* Status Attribute */
      struct buffer_head *b_next_free
      unsigned int b_count;             /* Nutzeranzahl des Puffers */
      unsigned long b_size;             /* Blockgöße [Byte] */
      char *b_data;                     /* Zeiger zu Datenblock */
      unsigned int b_list;
      unsigned long b_flushtime;        /* Rückspeicherzeit wenn geändert */
      unsigned long b_lru_time;         /* Zeit der letzten Nutzung */
      struct wait_queue *b_wait;
      struct buffer_head *b_prev;
      struct buffer_head *b_prev_free;
      struct buffer_head *b_reqnext;    /* nächster Puffer im Auftrag */
};
```

Abbildung II-5: Struktur der Linux-Puffer (Blockgeräte-Caching)

Ein Buffer besitzt verschiedene Attribute, die seinen Zustand kennzeichnen. Für diese Arbeit interessieren uns speziell die ersten beiden in **Tabelle II.2.** Diese werden von der Blockgerätesteuerung verändert, die anderen Attribute unterliegen einer Änderung durch die Linux-Pufferverwaltung.

Attribut	Bedeutung, wenn Attribut gesetzt
BH_Uptodate	Der Puffer enthält gültige Werte. Dieses Flag wird nach einem erfolgreich ausgeführten Zugriff auf den zugehörigen Block gesetzt.
BH_Lock	Puffer ist gesperrt, da er gerade von einem Blockgerät gelesen oder geschrieben wird. Solange dieses Flag gesetzt ist, darf der Block nicht fregegeben werden.
BH_Dirty	Pufferinhalt wurde verändert
BH_Req	Puffer soll gelesen oder geschrieben werden
BH_Touched	Auf diesen Puffer wurde mindestens einmal zugegriffen
BH_Has_aged	Puffer ist veraltet und wird verworfen
BH_Protected	Puffer ist geschützt
BH_FreeOnIO	Der zugehörige Puffer-Header kann nach einer I/O-Operation freigegeben werden

Tabelle II.2: Attribute eines Puffers

Wird ein Puffer neu angelegt, so ist er nicht „up-to-date“, d.h. er enthält keine gültigen Daten (*BH_Uptodate* gelöscht). Wenn dann der entsprechende Block des Blockgerätes in diesen Buffer gelesen wird, bekommt selbiger das Attribut *BH_Uptodate*. Eine darauffolgende Leseanforderung für den gleichen Block hat keine physische Leseoperation zur Folge, da der Inhalt noch vom Buffer gehalten wird. Nach einer Schreib-Operation auf den Block ist der Buffer ebenfalls mit *BH_Uptodate* versehen. Er bleibt das praktisch solange, bis ein Fehler beim Schreiben auf das physische Medium auftritt oder der Buffer invalidiert wurde.

Durch einen periodisch aktivierten Prozeß (*bdflush*) werden alle veränderten Blöcke auf die Medien geschrieben. Befindet sich ein zu lesender Block bereits in einem Puffer (zu erkennen am Blockattribut *BH_Uptodate*), so wird dessen Inhalt sofort geliefert, anderenfalls wird eine Anfrage an die Blockgeräteverwaltung gestellt. Die Verwaltung der Buffer geschieht über eine Hash-Tabelle, die Anzahl der Butter variiert ständig und hängt von der Systemauslastung und dem vorhandenen Hauptspeicher ab.

Unter Linux laufen fast alle Zugriffe auf Blockgeräte über die Bufferverwaltung, es sei denn, der Datenaustausch soll synchron erfolgen (alle Lese- und Schreiboperationen werden ohne Pufferung an das Blockgerät durchgereicht), in diesem Fall wird die Blockgeräteverwaltung direkt angesprochen.

Blockgeräteverwaltung

An diese Treiberkomponente richten sich alle Anfragen nach Daten von Blockgeräten, ausgehend von der Bufferverwaltung oder anderen Linux-Komponenten. Neben den logischen Positionsangaben wird hier auch ein Feld von Buffern sowie ein Operationsbefehl aus der Menge *{read, write, read_ahead, write_ahead}* als Parameter erwartet. Alle Anforderungen für Blockgeräte werden in eine Auftragswarteschlange eingereiht, deren Länge momentan 64 beträgt. Ein Eintrag in dieser Warteschlange gilt als frei, wenn er das Attribut *RQ_INACTIVE* trägt.

Die Funktion `ll_rw_block()`

- überprüft die richtige Blockgröße für das geforderte Blockgerät
- schließt Schreiboperationen auf schreibgeschützte Medien aus
- setzt das *BH_Req*-Bit jedes Buffers und
- läßt von `make_request()` für jeden Buffer eine Anfrage aufsetzen

Anzumerken ist hier, daß `ll_rw_block()` als Parameter ein Feld von Blöcken erwartet. Auf jeden Fall werden mehrere übergebene Puffer **einzeln** in die Warteschlange eingeordnet. Im ersten Moment erscheint es sinnlos, einen großen zu lesenden Datenblock in viele kleine Blöcke zu unterteilen, diese Zerstückelung wird aber beim Einordnen wieder aufgehoben, wenn nämlich genau geprüft wird, ob sich ein neu einzuordnender Block an irgendeine Geräteanfrage anfügen läßt.

Plug-/Unplug-Mechanismus - Warten auf Auftragsansammlungen

`Make_request()` setzt zuerst das Bit *BH_Lock* des Buffers und überprüft dann, ob die Forderung im gültigen Datenbereich des Mediums liegt. Sodann wird überprüft, ob sich für das anzusprechende Gerät bereits eine Anfrage in der Warteschlange befindet. Wenn nicht, wird die Funktion `plug_device()` aufgerufen. Diese legt in die Warteschlange einen dummy-Eintrag (Attribut *RQ_INACTIVE*!) nieder und nimmt das gemeinte Gerät in eine *task_queue* auf. Dieser Vorgang verhindert, daß eine Anforderung sofort nach Eintreffen bedient wird, sondern erst, wenn explizit auf das Ergebnis gewartet wird (`wait_on_buffer()`) oder kein Platz mehr in der Request-Queue frei ist (`get_request_wait`). Denn dann wird `unplug_device` aufgerufen, woduch alle wartenden Aufträge angestoßen werden.

Wenn es sich doch nicht um den ersten Eintrag für ein Gerät handelt, versucht Linux, diese neue Anfrage nach einem Block an schon bestehene Anfragen anzufügen (direkt davor oder dahinter). Dieses Reordering wird nur bei relativ zugriffsschnellen Blockgeräten versucht und gelingt nur, wenn in einer Blockanfrage bereits direkter Vorgänger oder Nachfolger des nun gewünschten Blockes stehen.

Reordering - Umsortieren von Aufträgen

Gelingt dies nicht, wird eine neue Anfrage erzeugt, indem in der Warteschlange ein freier Eintrag gesucht wird. Dabei dürfen Leseoperationen den gesamten Wartebereich nach freien Einträgen absuchen, Schreiboperationen hingegen nur $^2/_3$ des Bereiches. **Lesen hat** somit **Vorrang vor Schreiben**. Wenn im Moment kein freier Eintrag zu finden ist, kehrt die Funktion fehlerlos zurück, falls die Operation *ahead* (vorrauslesend oder -schreibend) war. Ansonsten wird auf das Freiwerden eines Eintrages gewartet (`get_request_wait()`). Dazu wird der aktuell aktive (und die Anforderung auslösende) Prozeß in den Zustand *UNINTERRUPTIBLE* überführt und in die Warteschlange *wait_for_request* eingeordnet, woraufhin der Scheduler andere Prozesse ausführt, die ihrerseits nach Entfernen eines Requests von der Request-Queue die Funktion `wake_up(`*wait_for_request*`)` aufrufen. Dieser oft im Linux-Kern angewandte Mechanismus verhindert ein Busy-Waiting und damit Deadlock-Gefahr, wenn eine Forderung im Moment nicht bedient werden kann.

Neue Anfragen werden abschließend in `add_request()` nach dem *Elevator*-Prinzip in die Warteschlange eingeordnet.[4] Der Eintrag wird als *RQ_ACTIVE* gekennzeichnet und erhält einen Verweis auf den Datenpuffer. Handelt es sich um ein SCSI-Gerät, so wird sofort die zugehörige Behandlungsroutine aufgerufen. Diese wird feststellen, daß für das angesprochene Gerät **keine** Aufträge bereitstehen, da der *plug/unplug*-Mechanismus es so will (Warten, bis einige Anfragen in der Schlange stehen, um Optimierungen vornehmen zu können). Beim *unplug*-Vorgang wird die Behandlungsroutine für alle diese gesammelten Aufträge einzeln aufgerufen.

Fehlerbehandlung

Ein Block wird, sobald er in die Warteschlange für Blockgeräte (Request Queue) eingeordnet wurde, mit dem Flag *BH_Lock* versehen. Das Flag *BH_Uptodate* wird gelöscht. Wenn ein Prozeß auf das Ergebnis eines Lese-/Schreibvorganges warten möchte, wird die Funktion `wait_on_buffer()` aufgerufen, die erst dann zurückkehrt, wenn das Flag *BH_Lock* gelöscht ist. Tritt während der Bearbeitung ein Fehler auf (z.B. Block nicht gefunden, Schreib- oder Lesefehler), so wird das Bit *BH_Uptodate* nicht gesetzt und der Auftragsprozeß kann geeignet darauf reagieren. Die Fehlerursache wird nicht als Nummer zurückgegeben, Linux gibt stattdessen eine entsprechende Fehlermeldung auf die Fehlerconsole aus.

Der festgestellte Fehler gilt immer für einen ganzen Buffer, also 1024 Byte für Festplatten und 2048 Byte für CD-ROM. Eine genauere Fehlerlokation (Byte-Ebene) ist nicht möglich.

[4] Dabei wird davon ausgegangen, daß die Kosten für einen Zugriff auf einen logischen Block von seiner Entfernung vom vorher zu lesenden/schreibenden logsichen Block abhängen. Dies bedeutet eine Vereinfachung der Sichtweise auf Blockgeräte, welche im Allgemeinen zu richtigen Entscheidungen führt, da die logische Blockanordnung (Cylinder/Head/Sector-Mapping) auf Datenträgern vom Hersteller nach optimalen Zugriffszeiten gewählt wird.

SCSI als Repräsentant von Blockgeräten

Wenn `unplug_device()` durch das Warten auf einen Puffer oder freien Auftrag aufgerufen wird, ruft diese Funktion die zuständige Behandlungsroutine des Blockgerätes für alle in der Auftragsschlange anstehenden Aufträge auf (z.B. im Fall der SCSI-Festplatte `do_sd_request()`). Diese Funktion überprüft, ob der SCSI-Hostadapter bereit ist (Aufruf der mittleren Schicht: Funktion `request_queueable()`), weitere Aufträge entgegenzunehmen und leitet ggf. den Einordnungsprozeß in die Warteschlange des Adapters ein (Aufruf von `requeue_sd_request()`).

Dabei muß nicht der gesamte Auftrag im Stück in die in diese Warteschlange eingefügt werden. Handelt es sich z.B. um einen Hostadapter ohne *scatter/gather*-Fähigkeit (siehe **S. 10: Scatter/Gather**) oder ist dessen *scatterlist* übergelaufen, so wird der Auftrag an der Grenzen der physischen Adressen der einzelnen Puffer in Einzelaufträge zerstückelt und so eingeordnet. Der ürsprüngliche Auftrag wird erst dann von der Warteschlange für Blockgeräte entfernt, wenn sich alle Teilaufträge in der Warteschlange des Hosts befinden.

Ein Auftrag steht nun in der Warteschlange des Hosts, an den das SCSI-Gerät angeschlossen ist und wird zu gegebener Zeit vom Host ausgeführt. SCSI-Geräte führen mitunter selbst noch einmal einen Umordnungsprozeß durch, um Operationen über Daten an die physischen Gegebenheiten anzupassen, die aber dem Betriebssystem verborgen bleiben.

Gesamtdatenweg

Zusammenfassend präsentiert sich der Weg von einer Datenanforderung eines Nutzerprozesses zum SCSI-Gerät wie in **Abbildung II-6**.

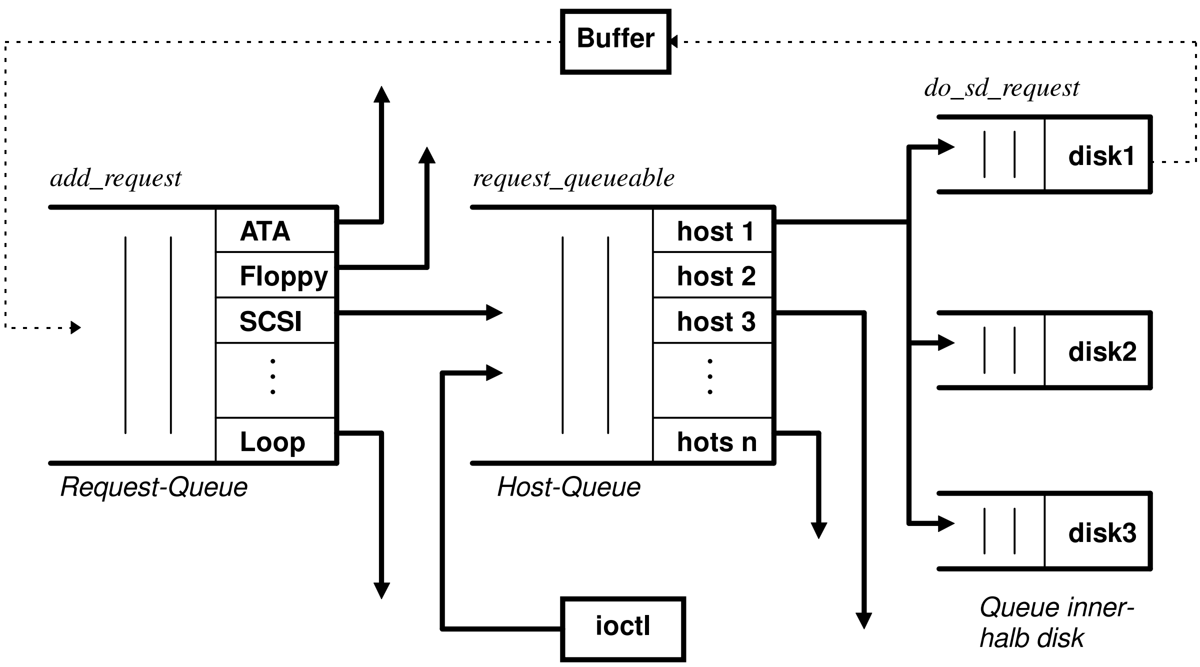

Abbildung II-6: Warteschlangen im Linux-Blockgerätetreiber

Blockierung von Prozessen

Nutzerprozesse können in Linux von Gerätetreibern blockiert werden. Das kann beim SCSI-Treiber dann auftreten, wenn eine von drei Möglichkeiten erfüllt ist (**Tabelle II.3**).

Blockade	**Warteschlange**
Ein neuer Auftrag soll in die Auftragswarteschlange eingefügt werden, aber im Moment ist kein freier Eintrag verfügbar.	*wait_for_request*
Ein SCSI-Hostadapter kann keine weitere Anforderung aufnehmen, da er selbst blockiert ist (Warteschlange für Host-Kommandos voll).	*host->host_wait*
Ein SCSI-Gerät kann keine weitere Anforderung aufnehmen, da es selbst gerade beschäftigt ist (interne Warteschlange voll).	*device->device_wait*

Tabelle II.3: mögliche Blockierungen und Warteschlangen, in die der aktuelle Prozeß bei Auftreten einer solchen eingeordnet wird

Für jedes eingebundene Gerät existiert eine Warteschlange *device_wait*, für jeden Hostadapter eine Warteschlange *host_wait*.

Der aktuelle Prozeß, der eine Arbeit mit einem blockierten SCSI-Gerät wünscht, wird bei Erreichen einer der genannten Bedingungen in eine zugehörige Warteschlange eingeordnet und als "nicht unterbrechbar" gekennzeichnet. Sodann wird der Linux-Scheduler aufgerufen, der die Prozessorzeit anderen Prozessen zuteilt.

```
#define SCSI_SLEEP(QUEUE, CONDITION) {                                          \
     if (CONDITION) {                                                           \
          struct wait_queue wait = {current, NULL};                             \
          add_wait_queue(QUEUE, &wait);                                         \
          for(;;) {                                                             \
               current->state = TASK_UNINTERRUPTIBLE;                           \
               if (CONDITION) {                                                 \
                    if (intr_count)                                             \
                         panic("scsi: trying to call schedule() "               \
                                      "in interrupt\n",                         \
                    schedule();                                                 \
               }                                                                \
               else                                                             \
                    break;                                                      \
          }                                                                     \
          remove_wait_queue(QUEUE, &wait);                                      \
          current->state = TASK_RUNNING;                                        \
     };                                                                         \
}
```

Abbildung II-7: Der SCSI-Treiber blockiert den aktuellen Prozeß (current), falls Condition wahr ist

Wird an anderer Stelle im Treiber eine der oben genannten Zustände entschärft, dann wird die entsprechende Warteschlange durchlaufen und jeder in der Schlange enthaltene Prozeß wieder aufgeweckt. Ein Deadlock könnte nun auftreten, wenn Prozesse auf das Ende einer SCSI-Operation warten, aber die Operation nicht ausgeführt wird (z.B. technischer Defekt des Gerätes). Tritt dieser Fall ein, verhindert ein SCSI-Timer den Stillstand des gesamten Systems.

III Entwurf

Folgend aus dem vorangegangenen Kapitel sollen hier einzelne Problemstellungen beim Schreiben des SCSI-Treibers besprochen werden.

III.1 Ausgangspunkte

Die in dieser Arbeit entwickelten Gerätetreiber sollen als Beitrag eines Echtzeitsystems basierend auf L4 gelten. Der SCSI-Treiber soll als Gerätetreiber die Verbindung zwischen SCSI-Geräten und einem noch zu entwickelnden echtzeitfähigen Dateisystem stehen. Der SCSI-Treiber selbst sollte auch echtzeitfähig sein. Weil es noch kein Dateisystem dieser Art gibt, sollen zwar im Treiber Vorraussetzungen für den Echtzeitbetrieb geschaffen werden, das Testen und Bewerten des Treibers wurde allerdings unter L3 vorgenommen. Daraus resultiert:

- Entwicklung eines auf dem L3-System lauffähigen Treibers
- Laufzeitumgebung des Treibers bildet ein Nutzerprozeß mit hoher Priorität
- Kommunikation des Prozesses mit anderen über GDP (*general driver protocol*)
- Realisierung des hardwareabhängigen Teils des SCSI-Treibers durch direkte Einbindung eines Linux-SCSI-Treibers (*lower level*) in Quellcodeform
- Bereitstellung der wichtigsten SCSI-Dienste
- möglichst unveränderte Übernahme des Linux-Quellcodes
- möglichst keine Änderung am L3-Systemkern

Daraus ergeben sich einzelne Arbeitsabschnitte, worauf in den nächsten Abschnitten eingegangen werden soll:

1. Optimale Abbildung der Nebenläufigkeiten im Linux-Kern auf L3-Threads
2. Nachbildung der Speicherverwaltung des Linux-Kerns
3. Abbildung der L3-Treiberschnittstelle (GDP) auf die Funktionen der Linux-Schnittstelle
4. Emulation der Dienste des Linux-Kerns

III.2 Entwurf der Thread-Struktur

Für das Verständnis der folgenden Abschnitte wird wieder auf [Stange 96] verwiesen. Der Ablauf innerhalb des SCSI-Treibers unterscheidet sich allerdings ein wenig von dem im Netzwerktreiber. Während bei genannter Arbeit der eigentliche Hardware-Gerätetreiber vollständig aus Linux herausgelöst wurde, ist es hier notwendig, neben der eigentlichen Hardwareansteuerung die gesamte SCSI-Funktionalität mit zu übernehmen. Das beinhaltet beispielsweise die getrennte Behandlung von SCSI-Disk- und SCSI-CD-ROM-Medien oder auch das *ioctl*-Protokoll.

Welche Nebenläufigkeiten existieren?

Der Zugriff auf den SCSI-Treiber erfolgt in Linux durch die Blockgeräte-Schicht. Bei Betrachtung von Datenlesen und -schreiben ergeben sich folgende Ausführungspfade (**Abbildung III-1**):

1. Die Funktion `ll_rw_block()` ist für die Einordnung der Aufträge in die Auftragswarteschlange verantwortlich und ruft die Behandlungsroutine `dev_request_fn()` (z.B. im Fall einer Festplatte `do_sd_request()`) auf.
2. Die Behandlungsroutine des entsprechenden SCSI-Gerätes arbeitet den Auftrag auf und läßt letztendlich von der mittleren Schicht (*medium level*) einen SCSI-Befehl in die Warteschlange des Hosts aufnehmen. Während dieser Zeit können bereits weitere Aufträge unter 1. eingehen.
3. Beim Auftreten eines Interrupts wird die vom SCSI-Gerätetreiber installierte Interruptbehandlungsroutine abgearbeitet.

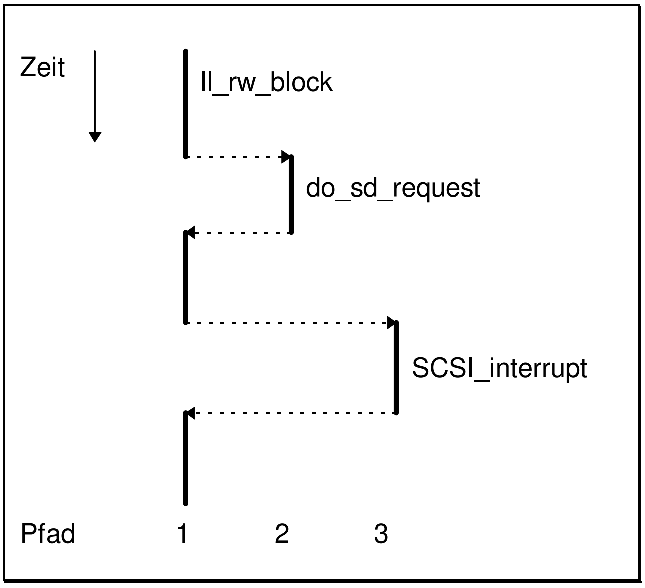

Abbildung III-1: Ausführungspfade im SCSI-Treiber

Pfad 1 kann blockieren, falls kein freier Eintrag mehr in der Auftragswarteschlange für Blockgeräte frei ist, *Pfad 2* blockiert, wenn entweder Hostadapter oder SCSI-Gerät keinen weiteren Befehl in die zugehörige Warteschlange aufnehmen kann (siehe **S. 20: Blockierung von Prozessen**). Der Stop des zugehörigen Threads würde aber die anderen Abarbeitungspfade nicht beeinflussen.

Möchte man dieses Verhalten unter L3 genau nachbilden, so bietet sich auf den ersten Blick eine Lösung mit drei Threads an (**Abbildung III-2**): Der erste Thread wartet auf neue Aufträge und stellt sie in die allgemeine Warteschlange für Aufträge. Thread zwei leert diese Warteschlange und füllt die Warteschlange für den SCSI-Hostadapter. Thread drei übernimmt die Reaktion auf Unterbrechungen.

```
gdp_thread()
{
      while(1)
      {
            ipc_receive(any_thread);
            ...
            ll_rw_block(cmd, dev, blocknr, size);
            ...
      }
}

do_request_thread()
{
      while(1)
      {
            if(new_request && (request_queueable(host))
                  requeue_request(request, host);
      }
}

interrupt_thread()
{
      while(1)
      {
            ipc_receive_interrupt();

            block(gdp_thread);
            do_scsi_interrupt();
            unblock(gdp_thread);
      }
}
```

Abbildung III-2: Abbildung der Nebenläufigkeiten auf drei Threads

Ist eine Ein-Thread-Lösung möglich?

Zusätzlich zu den in [Stange 96] angesprochenen Punkten, warum die Verwendung von drei Threads unangebracht ist (Operationszeit der einzelnen Threads gering, ohnehin Serialisierung der Threads durch Scheduler) liegt die Begründung in der Struktur des SCSI-Treibers im Linux-Kern: Eine Aufteilung des SCSI-Treibers in drei Threads ist nur mit erhöhtem Arbeitsaufwand möglich.

Bei genauerer Betrachtung stellen wir übrigens fest, daß selbst drei Threads nur für **eine** SCSI-Geräteart, z.B. SCSI-disk, ausreichen würde. Da aber unter Linux vier verschiedene SCSI-Geräteklassen existieren (disk, CD-ROM, tape, generic), müßte man sogar insgesamt sechs Threads einführen, da jedes dieser Blockgeräte anstehende Aufträge mit einer eigenen Funktion `dev_request_fn()` behandelt. Mit dieser Thread-Anzahl ginge ein nicht unerheblicher Verwaltungsaufwand einher. Deshalb soll versucht werden, auch hier die Ein-Thread-Lösung zu verwenden, allerdings abgewandelt (siehe **Abbildung III-3**).

```
single_thread()
{
      while(1)
      {
            ipc_receive(any_thread_or_interrupt);
            if(is_interrupt)
            {
                  do_scsi_interrupt();
            }
            else
            {
                  handle_gdp_order();
            }
      }
}
```

Abbildung III-3: Die Ein-Thread-Lösung ist auch hier gangbar

Im SCSI-Treiber von Linux gibt es keine direkte Behandlung einer Bottom-Half-Routine wie im Netzwerktreiber, die Trennung der Interrupt-Behandlungsroutine in Top- und Bottom-Half etfällt hier.

Dafür muß, wie oben erwähnt, das Problem der Blockierung von Prozessen gelöst werden (siehe **S. 20: Blockierung von Prozessen**). Denkbar wäre hier der Einsatz von mehreren Threads. Betrachten wir aber noch einmal die Vorgehensweise von Linux: Bei Notwendigkeit einer Blockierung wird der aktuell ablaufende Prozeß in eine Warteschlange eingeordnet und der Linux-Scheduler aufgerufen. Den L3-Scheduler kann man nicht direkt aufrufen. Statt dessen bietet sich folgende Lösung an (siehe **Abbildung III-4**): Der SCSI-Treiber ruft die Funktion `Wait(thread, order, timeout)` auf. Falls innerhalb von `timeout` eine `order` vom Prozeß `thread` für den SCSI-Treiber eintrifft, wird diese behandelt. Während der Ausführung von `Wait()` verbraucht der Treiberprozeß praktisch keine CPU-Rechenzeit.

```
#define SCSI_SLEEP(QUEUE, CONDITION) {                                  \
      IF(CONDITION)                                                     \
      {                                                                 \
            ipc_receive(any_thread_or_interrupt, timeout);              \
            if(not timeout)                                             \
            {                                                           \
                  if(is_interrupt)                                      \
                  {                                                     \
                        do_scsi_interrupt();                            \
                  }                                                     \
                  else                                                  \
                  {                                                     \
                        handle_gdp_order();                             \
                  }                                                     \
            }                                                           \
      }                                                                 \
}
```

Abbildung III-4: Linux-Scheduler-Aufrufe werden durch Warten auf ipc-Ereignisse ersetzt

Diese Lösung macht Sinn, denn so führt der SCSI-Treiber kein *Busy-Waiting* durch, sondern wartet auf ein Ereignis, speziell auf einen SCSI-Interrupt, der notwendig ist, um die Blockierung des Treibers wieder aufzuheben. Da immer nur ein Prozeß direkt mit dem SCSI-Treiber kommuniziert, erübrigt sich auch das Aufstellen einer Prozeß-Warteliste.

III.3 Zur Speicherverwaltung

Für die Verwaltung freier Speicherblöcke, für die ein L3-Programm selbst zuständig ist, wurde der Einfachheit halber der Algorithmus von Linux (`kmalloc`) übernommen. Die Funktion

```
void *malloc(unsigned int size)
```

stellt einen freien Speicherbereich zur Verfügung, ungenutzter Speicher kann durch

```
void free(void *ptr)
```

wieder freigegeben werden. Die Verwaltung der freien Speicherplätze ähnelt dem Buddy-Verfahren. Die Verwendung hat sich schon in [Stange 96] bewährt und soll deshalb hier nicht näher besprochen werden.

Im Unterschied zum Netzgerätetreiber hat der SCSI-Treiber einen relativ großen Speicherbedarf. Einerseits verwaltet jeder Host einen eigenen Speicherbereich, wo z.B. im Falle des Hostadapters der NCR53c8xx-Klasse der **SCSI-Code** untergebracht ist. Andererseits wird für die Erstellung der *scatterlists* (siehe **S. 10: Scatter/Gather**) Speicher benötigt (im ungünstigsten Fall eine Liste pro Auftrag). Im Vergleich zum Netzwerktreiber werden hier auch größere Bereiche als 4080 Byte alloziert (z.B. Länge des SCSI-Scripts vom NCR-Controller: ca. 13KB). Da eine L3-Speicherseite aber nur 4096 Byte lang ist, macht sich die Verwendung einer Seitenverwaltung notwendig, die Aufträge nach mehreren *physisch zusammenhängenden* Seiten bedienen kann. Dafür bietet sich die Seitenverwaltung (*page allocation*) von Linux an. Die Funktion

```
_get_free_pages(unsigned long order)
```

liefert ein order mehrere aufeinanderfolgende Seiten auf Anforderung von `malloc()`. Die Verwaltung der freien Speicherseiten erfolgt durch Freispeicher-Bitmaps. Dabei wird versucht zu verhindern, daß viele Anforderungen für eine Speicherseite dazu führen, daß keine zusammenhängenden Seiten mehr geliefert werden können. Ausgeschlossen werden kann dies aber nicht. Wenn der Fall in Linux auftritt, wird über einen mehrstufigen Prozeß versucht, mehr verfügbare Seiten zu erzeugen (Swapping, Memory Management). Dieser Weg ist hier nicht gangbar. Insofern stellt diese Lösung nur einen eingeschränkt sicheren Weg dar.

Für die eigentliche Verwendung dieses Treibers unter L4 ist sie dennoch tragbar, da dann einmalig Seiten beim Initialisierungsvorgang belegt werden und später nie wieder, da dann keine Lese-/Schreibpuffer erzeugt werden müssen, wie bei L3 und der Kommunikation durch das GDP (Bereitstellung von Puffern durch Speicher-Mapping-Funktionen des L4-Kerns).

III.4 GDP-Interface

Für den Betrieb unter L3 ist ein Interface notwendig, welches sich in den Grundzügen an die GDP-Richtlinien hält.

Behandlung von Early-Nachrichten

Insbesondere die Behandlung von *early in* und *early out orders* für die asynchrone Übertragung zwingt zu einigen Überlegungen.

[Hohmuth 96a] beschreibt die grundlegende Behandlungsweise von orders mit dem *early* Attribut. Eine *early in order* (Leseanforderung) soll demnach sofort mit einem *NIL reply* beantwortet werden und die Daten bei späterer Anfrage nach dem gleichen Block geliefert werden. Die Behandlung einer *early out order* (Schreibanweisung) funktioniert im Prinzip genauso, nur daß normalerweise keine Rückmeldung über den erfolgreichen Schreibvorgang gesendet wird. Falls hier allerdings ein Fehler auftritt, wird dies dem aufrufenden Prozeß bei einer späteren order mitgeteilt. Dieser muß dann davon ausgehen, daß alle Schreibaufträge zwischen dem Senden der *early out order* und der letzten order erfolglos waren.

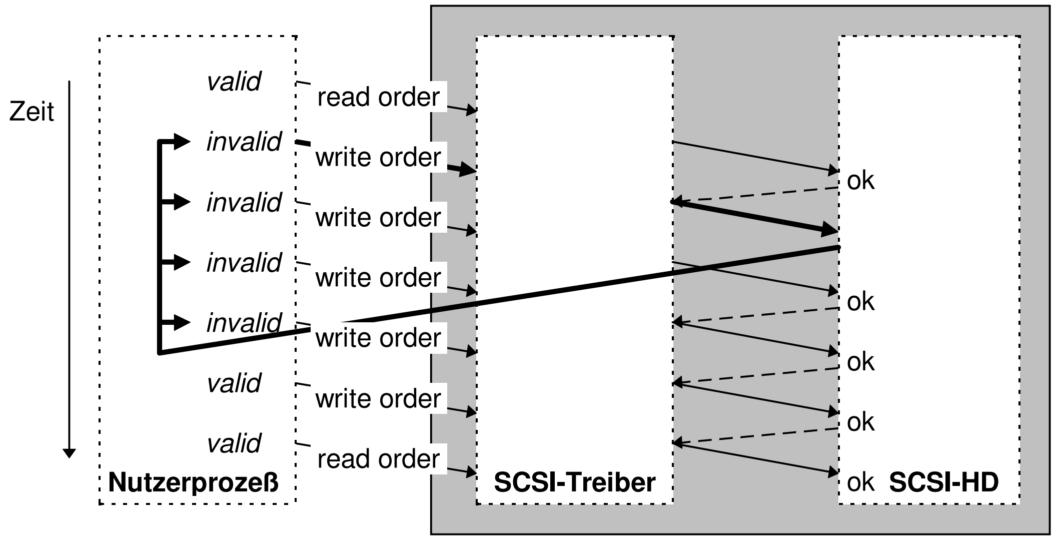

Abbildung III-5: Eine *early out order* kann im Fehlerfall viele vorher gelesene Blöcke invalidieren

Diese Behandlung mag für Netzgerätetreiber angebracht sein, für Prozesse, die Daten durch einen Blocktreiber schreiben wollen bedeutet das aber nur ungenaue Fehlerangaben.

Gesucht wird eine Lösung für ein Protokoll, das möglichst alle fehlerhaft geschriebenen Blöcke einzeln meldet. Andererseits brauchen korrekt geschriebene Blöcke nicht zurückgemeldet werden.

Durchführbar erscheint die folgende Lösung (**Abbildung III-6**): Durch eine leine Abwandlung des GDP-Protokolls: Das positive Ergebnis einer erfolgreich beendeten early out order wird nicht zurückgemeldet. Tritt ein Fehler beim Schreiben eines Blocks auf, so wird bei der nächsten order die Blocknummer genau dieses Blockes zurückgemeldet. Daraufhin wiederholt der aufrufende Prozeß seine order. Dieser Vorgang des Fehlerrückmeldens wird solange durchlaufen, bis alle Fehler gemeldet sind. Sollten mehrere Schreibfehler kurz nacheinander auftreten (z.B. Zugriff auf viele Sektoren hinter Geräte-Ende), so wird die Schleife entsprechend oft durchlaufen. Somit kann die Übertragung einerseits asynchron erfolgen, andererseits werden alle eventuell auftretenden Fehler gemeldet.

Diese Änderung weicht zwar ein wenig vom in [Hohmuth 96a] beschriebenen GDP-Protokoll ab, macht aber keine Anpassungen am L3-System notwendig. Einzig die mit dem SCSI-Treiber kommunizierenden Prozesse müssen sich nach dieser Handlungsweise richten. Diese Vorgehensweise ist korrekt, da der Aufbau der Schnittstelle letztlich als Definitionsfrage in Grenzen variierbar ist.

```
gdp_order()
{
     switch(order)
     case READ_ORDER:
          if(was_write_error)
          {
               send_write_error(partner);
          }
          else
          {
               if(already_read_block(block)
               {
                    send_block(partner);
                    release_buffer(block);
               }
               else if(early)
                    {
                         reserve_buffer(block);
                         queue_read_block(block);
                         ...
                         send_nil(partner);
                    }
                    else
                    {
                         reserve_buffer(block);
                         queue_read_block(block);
                         wait_on_buffer(block);
                         send_block(partner);
                         release_buffer(block);
                    }
               }
          }

     case WRITE_ORDER:
          if(was_write_error)
          {
               send_write_error(partner);
          }
          else
          {
               ...
          }
```

Abbildung III-6: Early-Behandlung mit Fehlerrückmeldung

Erstellung von Aufträgen

Wie in auf **S. 18 (Blockgeräteverwaltung)** erläutert, erwartet der Blocktreiber von Linux eingehende Aufträge in Form eines Feldes von zu lesenden/schreibenden Puffern. Um die Erstellung des GDP-Interfaces nicht zu komplizieren, soll der SCSI-Treiber bei *in/out-orders* nicht auf die Übergabe eines solchen Feldes angewiesen sein. Vielmehr sollen die übergebenen Parameter aus **einer** Puffer-Adresse nebst Geräte- und Blocknummer sowie Blocklänge bestehen.

Um die Sprache der Blockgeräteverwaltung von Linux zu sprechen, muß der übergebene Puffer aber dennoch in einzelne Bereiche zerlegt werden, allerdings erst im SCSI-Treiber selbst, der aufrufende Partnerprozeß merkt davon nichts.

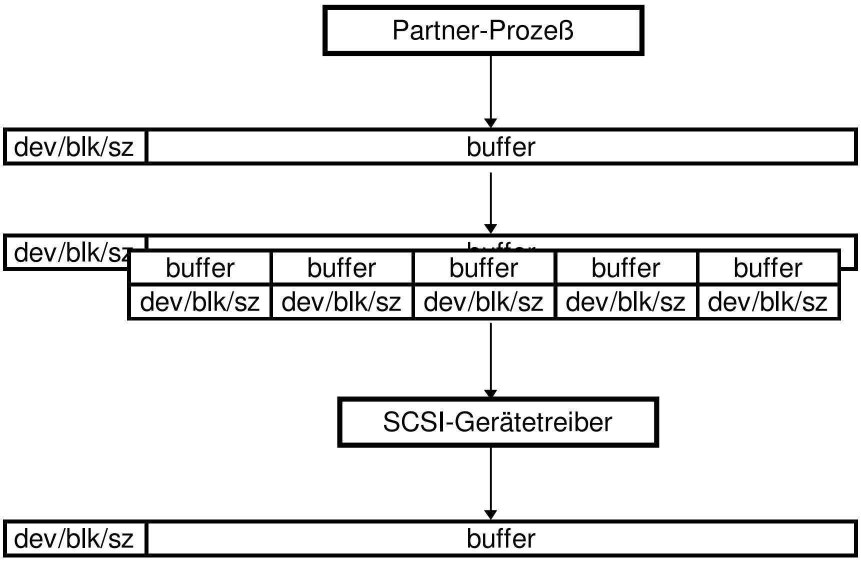

Abbildung III-7: Der SCSI-Treiber bekommt den Puffer in Form von Puffer-Arrays übergeben

Die Größe der zu bildenen Buffer hängt von der im Treiber festgelegten Größe für das gewählte Blockgerät ab (disk = 1024 Byte, CD-ROM = 2048 Byte).

III.5 Nachbildung der Linux-Kern-Dienste

PCI–Unterstützung

Die L3-Emulation richtet für den SCSI-Treiber ist für Hostadapter gedacht, die über den PCI-Bus an das System angebunden sind. Jeder PC mit PCI-Bus besitzt im ROM ein PCI-BIOS, welches sich sowohl vom Real Mode als auch vom Protected Mode aus ansprechen läßt. Linux beinhaltet einen PCI-Treiber für die Unterstützung dieses 32-Bit-BIOSes. Dieser Treiber wurde mit kleinen Änderungen bereits im Netzgerätetreiber genutzt und soll auch in den SCSI-Treiber eingebunden werden.

IV Implementierung

Diese Phase verlief in zweiphasig. Zuerst wurde die Netzgeräteemulation busmasterfähig gemacht, dann, mit Hilfe der gewonnenen Erkenntnisse, die SCSI-Geräteemulation implementiert. Da der erste Abschnitt weit mehr Zeit erforderte, als geplant, mußten Abstriche an der Vollständigkeit der Implementierung des SCSI-Treibers gemacht werden.

IV.1 Erweiterung des Netzgerätetreibers um Busmasterfähigkeit

Neue Version des Linux-Treibers

Die Entwicklung des Netztreibers für L3 wurde anhand des schon von Rene Stange benutzten Treibers für eine 3Com Netzkarte 3c595 vorgenommen. Mittlerweile (Oktober '96) gibt es den Linux-Treiber für die Geräteklasse 3c95x in der Version 0.28c [Becker 96]. Dieser unterstützt nun auch neben seperater Konfiguration verschiedener im PC eingebauter Netzkarten und automatischer Medien-Erkennung den DMA-Betrieb für das Senden von Datagrammen. Für den Empfang wird weiterhin der I/O-Betrieb genutzt.

DMA-Betrieb und physische/virtuelle Adressen

Ein Treiber benötigt für DMA-Betrieb die Kenntnis der physischen Adresse eines Datums (die DMA-Einheit kennt die virtuellen Adressen des Prozessors nicht). Unter Linux gehen die meisten Gerätetreiber von einer Übereinstimmung zwischen logischer und physischer Adresse aus, unter L3 findet aber eine andere Zuordnung statt. Da das Ziel darin bestand, den Linux-Treiber möglichst ohne Änderungen in das L3-System zu übernehmen, müssen auch die logischen Adressen, die vom L3-System an den Linux-Treiber geliefert werden, mit den physischen übereinstimmen.

Diese Forderung konnte nur mit einer L3-Kernänderung erfüllt werden. Dazu wurde eine neue Kernelfunktion `get_dma_mem()` implementiert, die die logische Adresse eines Speicherbereiches zurückliefert, für den logische und physische Adressen übereinstimmen. Busmasterbetrieb funktioniert somit nur mit L3-Kernels, die diese neue Funktion unterstützen!

Als schnellste Lösung zur Implementierung der Adreßverwaltung bot sich an, alle freien Speicherseiten (20) auf einen vorher von L3 angeforderten Speicherbereich zu legen, wo physische und logische Adressen übereinstimmen.

Bei einer späteren Nutzung unter L4 gelten die gleichen Vorraussetzungen, die aber mit Hilfe eines externen Pagers erfüllt werden könnten.

Automatische Medien-Erkennung

Im neuen Linux-Treiber für 3Com-Katen ist eine automatische Medienerkennung integriert. Diese wird von der Treiber-Emulation nicht unterstützt, es sollte sich aber kein Problem bei einem nachträglichem Einbau ergeben.

Die Erkennung beruht auf die Verwendung eines installierten Timers (Funktion `add_timer()`). Linux unterstützt mehrere benutzerdefinierte Timer, die in einer Liste angeordnet sind (**Abbildung IV-1**).

```
struct timer_list {
     struct timer_list *next;
     struct timer_list *prev;
     unsigned long expires;
     unsigned long data;
     void (*function)(unsigned long);
};
```

Abbildung IV-1: Benutzerdefinierte Timer werden in einer Liste verwaltet

Ein Treiber kann einen Timer installieren und dabei vorgeben, *wann* der Timer ausgelöst werden soll (`expires`) und *welche Funktion* dann ausgeführt werden muß (`*function`). In der Linux-Behandlungsroutine des Hardwareinterrupts Nr. 0 (Timer-Interrupt) wird die Systemzeit (`jiffies`-Variable) inkrementiert und jede Timerbehandlungsroutine ausgeführt, deren zugehöriger Zähler abgelaufen ist. Dieser Aufruf des Timers ließe sich auch in der Hauptschleife der L3-Emulation unterbringen. Als Muster kann die Implementierung des SCSI-Timers dienen.

Stolpersteine

Als Entwicklungsplatform wurde ein Pentium-PC mit Netzkarte 3c595 gewählt. Für diesen Netzkartentyp war kurz vor Beginn dieser Belegarbeit ein busmasterfähiger Treiber für Linux erschienen [Becker 96]. Dieser sollte den bis dahin auf dem L3-System eingesetzten Treiber mit Programmed I/O-Datenübertragung ablösen. Dabei stellte sich nach beträchtlichem Zeitaufwand heraus, daß gerade die interessierende Fähigkeit des Treibers, nämlich die Übertragung per DMA, fehlerhaft implementiert war.[5]

Diese Erfahrung bestätigt die Notwendigkeit der vollständigen Testung eines zu portierenden Treibers auf der Ausgangsplattform, bevor er für die Zielplattform verwendet werden kann.

IV.2 SCSI-Gerätetreiber

Konfiguration

Grundlegende Einstellungen (belegter Hardwareinterrupt, I/O-Basisadresse) des Treibers müssen schon bei der Programmgenerierung bekannt sein. Zwar sollte der PCI-Interrupt und der belegte I/O-Bereich direkt vom PCI-BIOS abgefragt werden. Eine Einbindung des Treibers in ein L3-System macht aber ein Wissen über diese beiden Einstellungen schon zur Zeit der Programmerstellung notwendig, der *hardware configurator* von L3 muß dem SCSI-Treiber bei Einbindung ins System den Hardwareinterrupt zuweisen. Die Datei driveconf.h (**Abbildung IV-2**) enthält deshalb auch diese Informationen.

Compilerdirektiven

Zu Testzwecken wurde eine Reihe von Wahlmöglichkeiten vorgesehen, die das Verhalten des SCSI-Treibers beeinflussen. Damit lassen sich Tests mit verschiedenen Zugriffsstrategien durchführen. In **Abschnitt V** wird deren Auswirkung auf die Übertragungsrate in Abhängigkeit vom gewählten Zugriffsverfahren (linear oder zufällig, lesend oder schreibend) untersucht.

---

[5] Es gab zwei Fehler: Erstens war das Register, welches den Interrupt bei Beendigung eines DMA-Transfers freigibt, fehlerhaft ausmaskiert, zweitens wurde ein Puffer nach Übertragung eines Datagramms nicht wieder freigegeben. DMA-Betrieb war so auch unter Linux unmöglich. Inzwischen wurde der Fehler im Linux-Treiber behoben.

```
#define CONFIG_NAME                    "L3.BIOS.NCR53C8XX"
#define CONFIG_VERSION                 30014
#define CONFIG_DATE                    "96-10-30"

#define CONFIG_PCI                     1
#define CONFIG_SCSI_NCR53C7xx          1
#define CONFIG_SCSI_NCR53C7xx_sync     1
#define CONFIG_SCSI_CONSTANTS          1
#define CONFIG_BLK_DEV_SD              1
#define CONFIG_BLK_DEV_SR              1

#define CONFIG_RESERVED_PAGES          192
#define CONFIG_MAX_BUFFERS             100
#define CONFIG_MAX_IO                  31*1024
#define CONFIG_MAX_EARLY_RET           20

#define CONFIG_IRQ                     5
#define CONFIG_BASE_ADDR               0xe800

#define CONFIG_ELEV_REQUESTS           1
#define CONFIG_JOIN_REQUESTS           1
#undef  CONFIG_TQUEUE
```

Abbildung IV-2: Treiberabhängige Konfigurationsdatei

Folgende Wahlmöglichkeiten wurden im Quelltext vorgesehen, um Aussagen über mögliche Optimierungen zu treffen:

CONFIG_ELEV_REQUESTS	Wenn dieses Symbol definiert ist, wird der von Linux vorgesehene "Fahrstuhl"-Algorithmus zum Einfügen von neuen Aufträgen in die Auftragswarteschlange für Blockgeräte benutzt und damit eine Umordnung der Aufträge zugelassen.
CONFIG_JOIN_REQUESTS	Die Aktivierung dieses Symbols erlaubt das **Zusammenfügen** von Aufträgen. Damit werden zwar einzelne Aufträge umfangreicher, die Anzahl an auszuführenden Aufträgen und damit die Anzahl an auftretenden Busarbitrierungen und Interrupts wird aber geringer. **Beispiel:** Es sollen die Blöcke 143 bis 152 geschrieben werden. Es wird angenommen, in der Warteschlange befinden sich bereits Aufträge, einer von ihnen soll die Blöcke 153 und 154 schreiben. Der alte Auftrag erhält als Vorspann die neue Anforderung, somit braucht kein neuer Auftrag erstellt zu werden.
CONFIG_TQUEUE	Die Nutzung dieses Symbols sollte nur zu Testzwecken geschehen. Aktiviert werden kann hier ein drittes Feature von Linux, nämlich erst solange mit der Abarbeitung von anstehenden Aufträgen zu warten, bis die Auftragsschlange voll ist oder ein Nutzerprozeß explizit auf die Erfüllung eines Auftrages wartet. Das ergibt deshalb einen Sinn, weil so mehr Möglichkeiten zum Zusammenfügen von Aufträgen gegeben sind. Für die Echtzeitfähigkeit des Treibers ist dieses Verhalten eher unangebracht und wird deshalb im Folgenden auch nicht näher betrachtet.

PCI-BIOS-Abfrage

Die PCI-Unterstützung wird, wie beim Netzgerätetreiber, direkt zum SCSI-Treiber hinzugelinkt. Für spätere Arbeiten ist vorgesehen, **einen** PCI-Treiber für das gesamte L3/L4-System zu schreiben, auf den alle Prozesse über IPC zugreifen können. Das PCI-BIOS liefert dann dem Prozeß wichtige Informationen über belegte Interrupts und Adressen.

Behandlung von GDP-Aufträgen

Für den Testbetrieb unter L3 wurde ein Interface entwickelt, welches sich an die GDP-Richtlinien hält (siehe **III.4**). Dabei wurde es notwendig, eine Art Pufferverwaltung in den Treiber zu integrieren. Grund dafür ist, daß der Linux-SCSI-Treiber mit Aufträgen arbeitet, in denen Verweise auf verkettete Puffer stehen. Diese Implementierung von Puffern ist nur für Tests unter L3 vorgesehen und sollte unter L4 überflüssig sein, da die Pufferverwaltung dann im Dateisystem integriert ist und der SCSI-Treiber mit Speicheradressen dieses Prozesses umgeht.

Erzeugen von Anfragen

Die im Entwurf erwähnte Aufteilung des vom Partner übergebenen Speicherbereiches in Puffer der Blockgröße des jeweiligen Blockgerätes ist **nicht** erforderlich, der SCSI-Treiber kann, wie sich herausstellte, auch mit anderen Blockgrößen als der fest für das jeweilige Gerät vorgegebenen umgehen. Auf den übergebenen Datenbereich wird nur **ein** Puffer mit einer vielfachen Länge der Blockgröße des Gerätes (1024 Byte für disk, 2048 für CD-ROM) gelegt. Diese Vereinfachung stellt eine Verringerung des Verwaltungsaufwandes dar und sollte sich positiv auf die Gesamtperformance auswirken.

Nanosleep

Viele SCSI-Gerätetreiber (so auch der für den NCR53c8xx) benötigen eine Verzögerungsfunktion, die den Treiber für den Bruchteil einer µs warten läßt. Aus Aufwandsgründen wurde hier einfach die Funktion von Linux übernommen, die für diesen Zeitraum ein *Busy-Waiting* durchführt. Dabei wird eine Schleife **n**-mal durchlaufen, wobei **n** beim Initialisierungsvorgang anhand der Rechnergeschwindigkeit bestimmt wurde (*BogoMips*-Zahl). Diese Sleep-Funktion wird nur sehr selten verwendet und sollte den normalen Programmablauf nicht stören. Die größte im Treiber verwendete Zeitverzögerung beträgt 25µs.

Die Initialisierung der Warteschleife (Anpassung der Durchlaufanzahl an die Rechnergeschwindigkeit) erfolgt in der Funktion `calibrate_delay_loop()`, die ebenfalls unverändert von Linux übernommen wurde.

Barrier

Komplizierter wird es, wenn Geräte durch *Busy-Waiting* auf einen *Hardwareinterupt* warten. Aufgrund der gewählten Ein-Thread-Struktur ist Busy-Waiting auf Ereignisse verboten und muß gesondert behandelt werden. Dazu wurde die Funktion

```
handle_one_order(unsigned int timeout)
```

definiert. Diese Funktion wartet auf eine order vom L3-System, welche einen Interrupt oder eine IPC-Botschaft eines anderen Prozesses bedeuten kann. Die Funktion kehrt spätestens nach Ablauf des Timouts zurück, egal ob eine Order einging oder nicht. Falls aber ein Interrupt auftritt, so wird dieser von der entsprechenden Behandlungsroutine verarbeitet und der Treiber registriert das eingetroffene Ereignis. In der Praxis werden für timeout Werte von 10ms gewählt. Dieser Wert garantiert einerseits ausreichend Wartezeit (sowie Entlastung für den Systemkern) und verhindert andererseits eine zu lange Wartezeit, die sich negativ auf das Laufzeitverhalten des SCSI-Treibers auswirken könnte.

```
timeout = get_time() + 500ms;
while ((test_completed == -1) && get_time < timeout)
     barrier();
```

Abbildung IV-3: Manchmal führt der Linux-Treiber Busy-Waiting auf ein Ereignis durch

Die Funktion barrier, die innerhalb der Busy-Wait-Schleife aufgerufen wird, ist nichts anderes als ein Alias für `handle_one_order()`. Unter Linux besteht sie aus einer leeren Anweisung. Sie ist deshalb notwendig, damit der C-Compiler sie beim Übersetzungsvorgang nicht "wegoptimiert", da diese Schleife im Grunde genommen "nichts" macht.

Lösung des Scheduler-Problems

Unter L3 kommuniziert immer nur eine (auch als Partner bezeichnete) Task mit einem Gerätetreiber. Aufgrund dieser Tatsache währe ein schedule()-Aufruf wie in Linux unsinnig, wenn er denn machbar wäre. Kommt es zu einer Blockade des SCSI-Gerätetreibers, weil z.B. gerade keine Einträge mehr in der Request-Queue frei sind, wird als Konsequenz daraus auch hier die Funktion handle_one_order aufgerufen, denn dieser Fall ist im Grunde genommen nichts anderes als das eben besprochene Busy-Waiting auf ein Ereignis in der `barrier()`-Schleife.

Aufgrund dieser Betrachtung läßt sich auch das auf **S. 20 (Blockierung von Prozessen)** behandelte Codefragment korrekt in den L3-Treiber integrieren (**Abbildung IV-4**):

```
#define SCSI_SLEEP(QUEUE, CONDITION) {                                   \
     if (CONDITION) {                                                    \
          for(;;) {                                                      \
               if (CONDITION) {                                          \
                    if (intr_count)                                      \
                         panic("scsi: trying to call schedule() "        \
                                       "in interrupt\n",                 \
                    handle_one_order(WAIT_DEFAULT);                      \
               }                                                         \
               else                                                      \
                    break;                                               \
          }                                                              \
     };                                                                  \
}
```

Abbildung IV-4: So kann man auf blockierte SCSI-Geräte reagieren

SCSI-Timeouts

Durch Fehlfunktion könnte ein SCSI-Gerät den Prozeß, der auf dieses Gerät zugreifen möchte, blockieren. Um das zu verhindern wird unter L3 der SCSI-Timer nachgebildet. Bei Initialisierung definiert der Treiber eine Funktion die ausgeführt werden soll, wenn der SCSI-Timer abgelaufen ist. Die Funktion `check_scsi_timer` die am Ende der Funktion `handle_one_order()` aufgerufen, nimmt den Test auf Ablauf des SCSI-Timers vor und ruft ggf. die Behandlungsroutine für timeouts auf.

V Leistungsbewertung

V.1 Busmasterfähiger Netzwerktreiber

Für die Beurteilung der Leistung der Lösung wurden Messungen unter Linux und L3 in den Betriebsarten Programmed I/O und Busmastered I/O vorgenommen. Hier wurde das gleiche Meßverfahren wie in [Stange 96] verwendet. Aufgrund der vielen notwendigen Messungen wurde der Meßzyklus von 500000 auf 200000 Datagramme reduziert.

Die wichtigsten Daten der für den Test verwendeten Rechner. lassen sich aus Tabelle V.1 ablesen. PC1 und PC2 wurden über einen Hub (100MBit/s) verbunden. Sowohl unter Linux als auch L3 hatten die Rechner keine weitere Verbindung zur Außenwelt.

	PC 1	**PC 2**
CPU	Pentium 100MHz	Pentium 90MHz
Hauptspeicher	32MB	16MB
Netzwerkkarte	3Com 3c595 PCI	3Com 3c595 PCI

Tabelle V.1: Verwendete Testplattform für Test des Netzgerätetreibers

Der Test begann jeweils dann, wenn keine Hintergrundaktivitäten mehr verzeichnet wurden. Im Testverlauf sendete PC1 entsprechend dem Test entweder 100000 mal 1 Datagramm bzw. 5000 mal 20 Datagramme. Nach dem Senden jeden Packetes wartete PC1 auf die Rücksendung des Packetes durch PC2.

Größe	**L3 ohne DMA**		**L3 mit DMA**		**Linux ohne DMA**		**Linux mit DMA**	
[Byte]	[s]	[MBit/s]	[s]	[MBit/s]	[s]	[MBit/s]	[s]	[MBit/s]
100	51	(3,1)	50	(3,2)	50	(3,2)	50	(3,2)
200	61	(5,2)	60	(5,3)	58	(5,5)	54	(5,9)
300	66	(7,3)	63	(7,6)	65	(7,4)	61	(7,9)
400	73	(8,8)	69	(9,3)	68	(9,4)	64	(10,0)
500	81	(9,9)	74	(10,8)	74	(10,8)	69	(11,6)
600	87	(11,0)	81	(11,9)	81	(11,9)	74	(13,0)
700	94	(11,9)	86	(13,0)	88	(12,7)	81	(13,8)
800	100	(12,8)	92	(13,9)	94	(13,6)	86	(14,9)
900	107	(13,5)	97	(14,8)	102	(14,1)	94	(15,3)
1000	108	(14,8)	98	(16,3)	110	(14,5)	100	(16,0)
1100	114	(15,4)	103	(17,1)	117	(15,0)	105	(16,8)
1200	120	(16,0)	108	(17,8)	125	(15,4)	112	(17,1)
1300	127	(16,4)	114	(18,2)	130	(16,0)	117	(17,8)
1400	133	(16,8)	120	(18,7)	138	(16,2)	123	(18,2)
1500	140	(17,1)	125	(19,2)	145	(16,6)	130	(18,5)

Tabelle V.2: Meßergebnisse für die Übertragung von 200000 x 1 Datagramm[6]

[6] 1MBit/s = 10^6Bit/s

Der leichte Geschwindigkeitsvorteil für L3 bei Datagrammgrößen ab 1000 Byte liegt an der Wahrnehmungsgrenze und ist wahrscheinlich auf den größeren Overhead unter Linux durch parallellaufende Prozesse zurückzuführen.

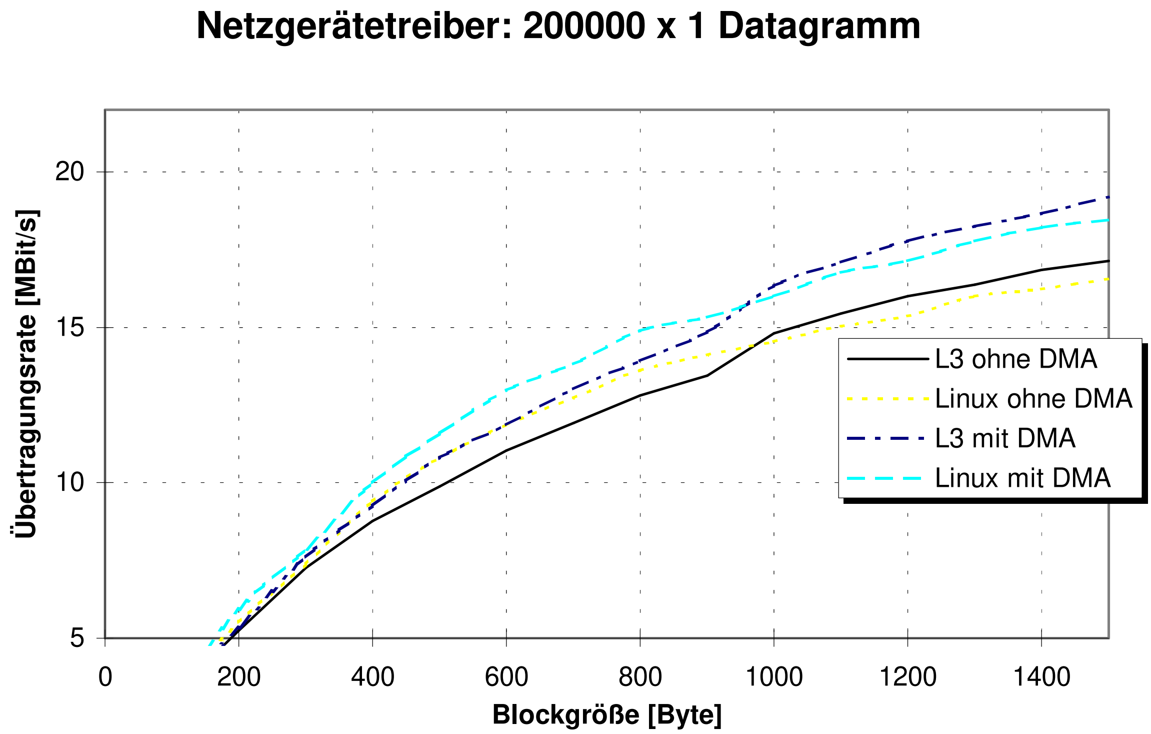

Abbildung V-1: Vermessung des Netzgerätetreibers (200000 x 1 Datagramm)

Grundsätzlich kann die in der ursprünglichen Arbeit [Stange 96] getroffene Aussage bestätigt werden, daß sich der Emulationscode unter L3 nicht negativ auf die Gesamtperformance auswirkt.

Weiterhin ist erkennbar, daß die Verwendung des DMA-Übertragungsmodus' einen leichten Performance-Gewinn erbringt, wichtiger sollte sich aber die verringerte CPU-Belastung auf das Gesamtsystem auswirken.

Größe [Byte]	**L3 ohne DMA** [s]	[MBit/s]	**L3 mit DMA** [s]	[MBit/s]	**Linux ohne DMA** [s]	[MBit/s]	**Linux mit DMA** [s]	[MBit/s]
100	34	(4,7)	35	(4,6)	31	(5,2)	31	(5,2)
200	38	(8,4)	38	(8,4)	35	(9,1)	34	(9,4)
300	41	(11,7)	41	(11,7)	37	(13,0)	38	(12,6)
400	45	(14,2)	45	(14,2)	41	(15,6)	41	(15,6)
500	48	(16,7)	48	(16,7)	45	(17,8)	44	(18,2)
600	51	(18,8)	51	(18,8)	48	(20,0)	48	(20,0)
700	55	(20,4)	55	(20,4)	52	(21,5)	52	(21,5)
800	58	(22,1)	57	(22,5)	55	(23,3)	54	(23,7)
900	61	(23,6)	61	(23,6)	60	(24,0)	59	(24,4)
1000	65	(24,6)	65	(24,6)	62	(25,8)	62	(25,8)
1100	69	(25,5)	69	(25,5)	65	(27,1)	65	(27,1)
1200	72	(26,7)	71	(27,0)	69	(27,8)	69	(27,8)
1300	75	(27,7)	75	(27,7)	73	(28,5)	72	(28,9)
1400	79	(28,4)	78	(28,7)	76	(29,5)	75	(29,9)
1500	82	(29,3)	81	(29,6)	80	(30,0)	79	(30,4)

Tabelle V.3: Meßergebnisse für die Übertragung von 10000 x 20 Datagrammen

Bei Senden von jeweils 20 Datagrammen ergibt sich praktisch keine Veränderung der Gesamtübertragungsrate ziwschen DMA- und Programmed-I/O-Betrieb.

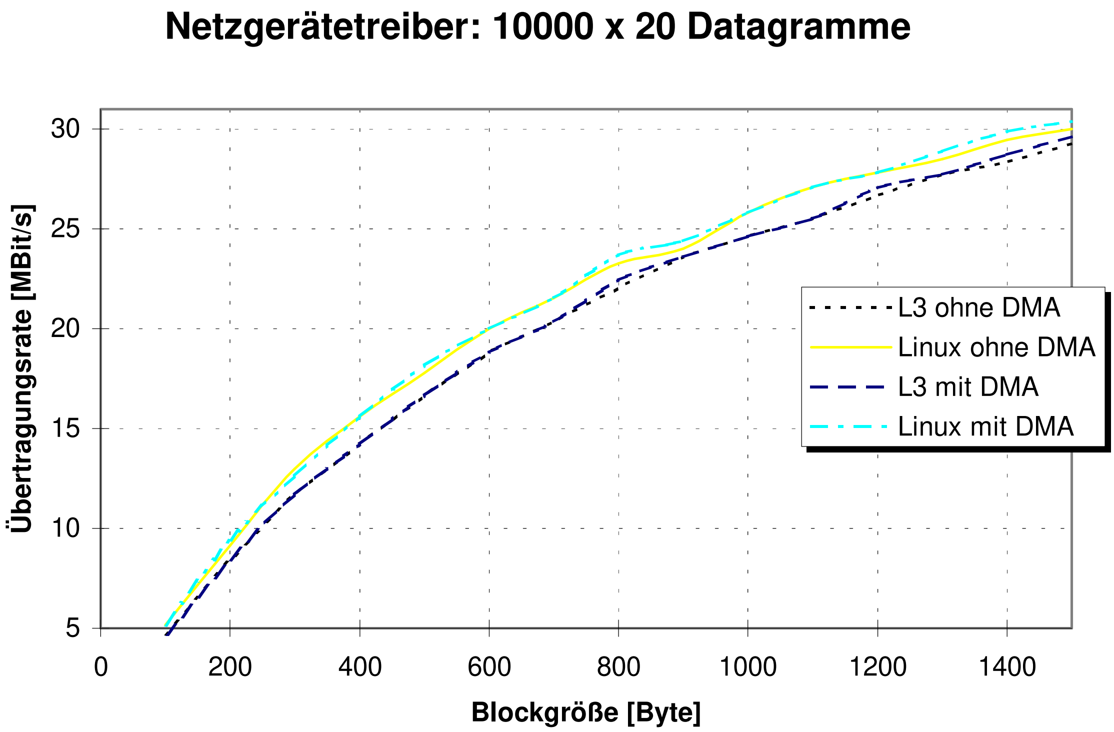

Abbildung V-2: Graphische Darstellung (10000 x 20 Datagramme)

Auf eine Messung der CPU-Belastung mußte aus Zeitgründen verzichtet werden, es wird erwartet, daß diese bei DMA-Betrieb geringer ist, als bei I/O-Betrieb.

V.2 SCSI-Treiber

Zum Ausmessen des SCSI-Gerätetreibers unter L3 und Vergleich mit Linux wurde ein L3-Programm entwickelt, das über das GDP Daten vom SCSI-Treiber liest und schreibt. Dabei wurde besonders auf die Implementierung von Orders mit Early Attribut (siehe **S. 26: Behandlung von Early-Nachrichten**) wert gelegt. Das Programm arbeitet (lesend oder schreibend) auf einem definierten Bereich mit linearem oder zufällig verstreutem Zugriff. Im zweiten Fall wurde mit einem Feld von Pseudozufallszahlen gearbeitet, um ein möglichst ungünstigen aber nicht unmöglichen Arbeitsfall zu simulieren.

Verwendet wurden zwei verschiedene Festplatten um einen Vergleich zwischen verschieden schnellen SCSI-Geräten zu ermöglichen. Die benutzte Hardware ist kurz in (**Tabelle V.4**) aufgelistet.

	Festplatte 1	**Festplatte 2**
Bezeichnung	Quantum LPS52S	IBM DORS 32160
Kapazität	52MB	2.1GB

Herstellungsjahr	ca. 1988	1996
Übertragungsart	asynchron 4.3Mhz *(SCSI-1)*	synchron 10Mhz *(SCSI-2)*

	Rechner
CPU	Pentium 100MHz
Hauptspeicher	32MB
SCSI-Controller	Asus SC-200 (mit NCR53C810) PCI

Tabelle V.4: Festplatten und Testrechner für Messungen am SCSI-Treiber

Zuerst wurde die Leistung des Treibers an einem schnellen SCSI-Gerät gemessen (Festplatte IBM DORS 32160 mit SCSI-2-Schnittstelle). Hier zeigen sich deutlich Einflüsse im Treiber

Größe	**Anzahl**	**Lesen linear**				**Lesen verstreut**			
		ohne Verknüpf.		**mit Verknüpf.**		**ohne Sort.**		**mit Sort.**	
[Byte]		[s]	[MB/s]	[s]	[MB/s]	[s]	[MB/s]	[s]	[MB/s]
1024	32768	35,17	(0,95)	17,41	(1,93)	342,91	(0,10)	273,89	(0,12)
2048	16384	17,95	(1,87)	9,77	(3,43)	176,81	(0,19)	141,57	(0,24)
4096	8192	9,41	(3,57)	6,21	(5,40)	96,87	(0,35)	73,86	(0,45)
6144	5632	7,27	(4,76)	6,12	(5,65)	63,40	(0,55)	52,30	(0,66)
8192	4096	5,99	(5,60)	5,93	(5,66)	48,19	(0,70)	40,62	(0,83)
12280	2816	6,10	(5,67)	6,11	(5,66)	35,02	(0,99)	29,12	(1,19)
16384	2048	5,92	(5,67)	5,93	(5,66)	26,88	(1,25)	24,46	(1,37)
20480	1536	5,55	(5,67)	5,56	(5,66)	21,21	(1,48)	19,17	(1,64)
24576	1280	5,55	(5,67)	5,55	(5,67)	18,59	(1,69)	16,97	(1,85)
28672	1024	5,19	(5,66)	5,18	(5,67)	15,45	(1,90)	13,80	(2,13)
31744	1024	5,74	(5,66)	5,73	(5,67)	16,35	(1,99)	14,85	(2,19)

Tabelle V.5: Messungen unter L3: Lesen von IBM DORS 32160[7]

implementierten Optimierungen:

- Wenn viele aufeinanderfolgende Aufträge an den SCSI-Treiber gereicht werden (linearer Zugriff), erhöht die Einstellung "**verknüpfen**" den Datendurchsatz, besonders bei kleineren Blockgrößen. Mehrere Anfragen werden dabei zu einer zusammengefaßt, wodurch sich die Anzahl der Unterbrechungen entscheidend senken kann. Größere Blöcke wirken sich besser auf die Relation "Daten zu Steuerdaten" (siehe **S. 9: Aufwendiges Busprotokoll - Lange Ausführungszeiten**) aus, hier bringt das Verknüpfen von Aufträgen kaum Vorteile.
- Ein **Sortieren** der eingehenden Aufträge nach dem "Fahrstuhl"-Prinzip vermindert die Kopfbewegungen der Festplatte und führt bei wahlfreiem Zugriff zu Verbesserungen im Zugriffsverhalten (auch deutlich am geringeren Geräuschpegel der Festplatte zu hören).

[7] 1MB/s = 10^6Byte/s

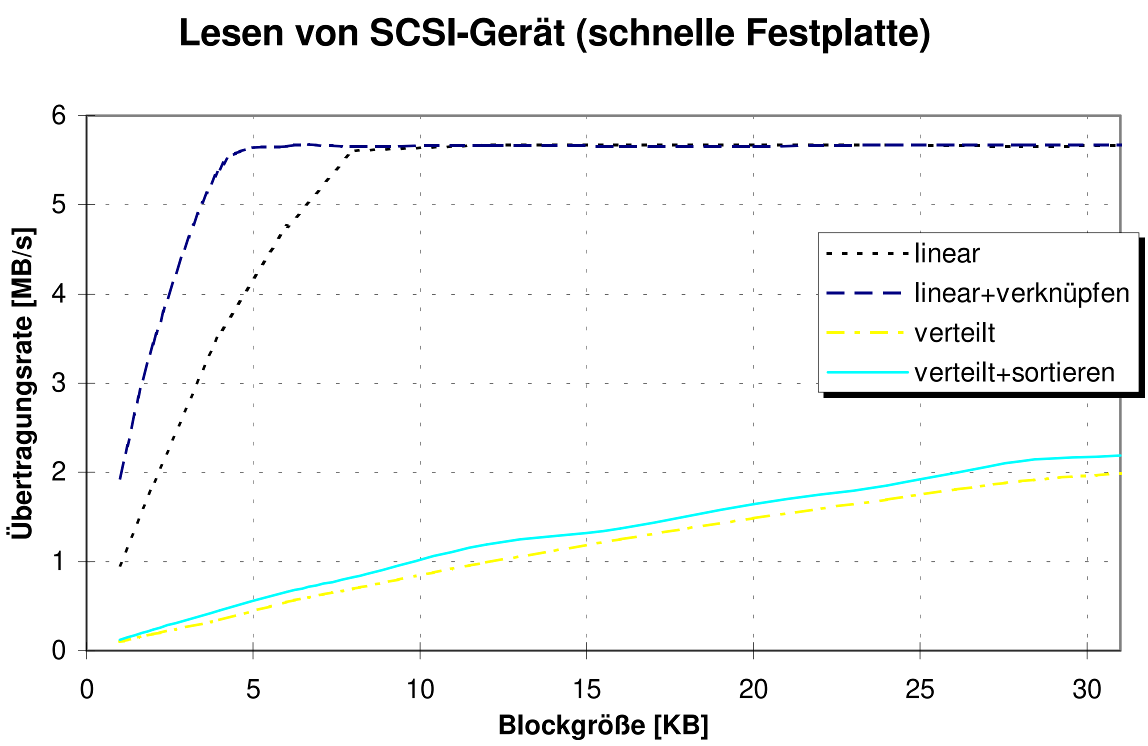

Abbildung V-3: Messungen unter L3: Lesen von IBM DORS 32160

Im SCSI-Treiber selbst unterscheiden sich Schreib- von Lesekommandos nur im verwendeten SCSI-Befehl, die sonstige Behandlung der Aufträge ist gleich - abgesehen davon, daß Lesen Vorrang vor Schreiben hat (siehe **S. 19: Reordering - Umsortieren von Aufträgen**). Der Vollständigkeit halber soll hier dennoch eine Überprüfung der Schreibleistung erfolgen.

Dabei erleben wir eine kleine Überraschung (**Abbildung V-4**): Bis zu einer Blockgröße von 7KB zeigen alle Kurven den erwarteten Verlauf. Bei größeren Blöcken kommt es plötzlich zu dramatischen Einbrüchen der Schreibgeschwindigkeit bei linearem Zugriff.

Der Grund zeigt sich, wenn man die hier zusätzlich eingezeichnete Kurve "**linear+sort+verkn.**" betrachtet, sie zeigt in etwa den erwarteten Verlauf. Diese Messpunkte wurde mit zusätzlich eingeschalteter Sortierung (nach Fahrstuhl-Prinzip) aufgenommen: Bei linearem Zugriff sollte sich dieses Umsortieren eigentlich erübrigen.

Daß sie hier dennoch notwendig ist, liegt an der Implementierung der Early-Out-Order (Schreibanforderung ohne Rückmeldung) des SCSI-Treibers und des Testprogramms. Wir erinnern uns (siehe **S. 26: Behandlung von Early-Nachrichten**), wie eine Early-Out-Order funktioniert: Der Auftraggeber sendet einen Datenblock nebst Statusinformationen zum SCSI-Treiber. Dieser reserviert sich einen Schreibpuffer und gibt bei Erfolg der Reservation sofort eine positive Rückmeldung an den Absender (obwohl der Block noch nicht geschrieben worden ist). Ist im Moment kein Speicher für einen Schreibpuffer verfügbar, muß das dem Partner mitgeteilt werden, dieser sendet die gleiche Anforderung später noch einmal.

Im Testprogramm wird solch ein erfolgloser Schreibauftrag an das Ende einer Warteschlange aufgenommen, welche zyklisch abgearbeitet wird, um die darin enthaltenen Aufträge zur Abarbeitung durch den SCSI-Treiber zu bringen. Offensichtlich kommt es hier oft dazu, daß aus dieser Warteschlange Aufträge in falscher Reihenfolge zum SCSI-Treiber gesendet werden.

Größe [Byte]	Anzahl	Schreiben linear ohne Verknüpf. [s]	[MB/s]	mit Verknüpf. [s]	[MB/s]	Schreiben verstreut ohne Sort. [s]	[MB/s]	mit Sort. [s]	[MB/s]
1024	32768	37,99	(0,88)	14,09	(2,38)	347,71	(0,10)	269,32	(0,12)
2048	16384	19,38	(1,73)	8,89	(3,77)	178,65	(0,19)	137,37	(0,24)
4096	8192	9,67	(3,47)	8,19	(4,10)	92,73	(0,36)	71,88	(0,47)
6144	5632	6,93	(4,99)	6,18	(5,60)	66,73	(0,52)	50,96	(0,68)
8192	4096	36,40	(0,92)	8,40	(3,99)	49,89	(0,67)	39,32	(0,85)
12280	2816	26,38	(1,31)	6,44	(5,37)	36,44	(0,95)	28,67	(1,21)
16384	2048	21,16	(1,59)	9,44	(3,55)	27,79	(1,21)	23,66	(1,42)
20480	1536	16,73	(1,88)	7,94	(3,96)	21,80	(1,44)	18,77	(1,68)
24576	1280	14,78	(2,13)	7,45	(4,22)	19,04	(1,65)	16,53	(1,90)
28672	1024	12,71	(2,31)	6,30	(4,66)	15,81	(1,86)	13,91	(2,11)
31744	1024	13,70	(2,37)	6,43	(5,06)	16,80	(1,93)	14,59	(2,23)

Tabelle V.6: Messungen unter L3: Schreiben auf IBM DORS 32160

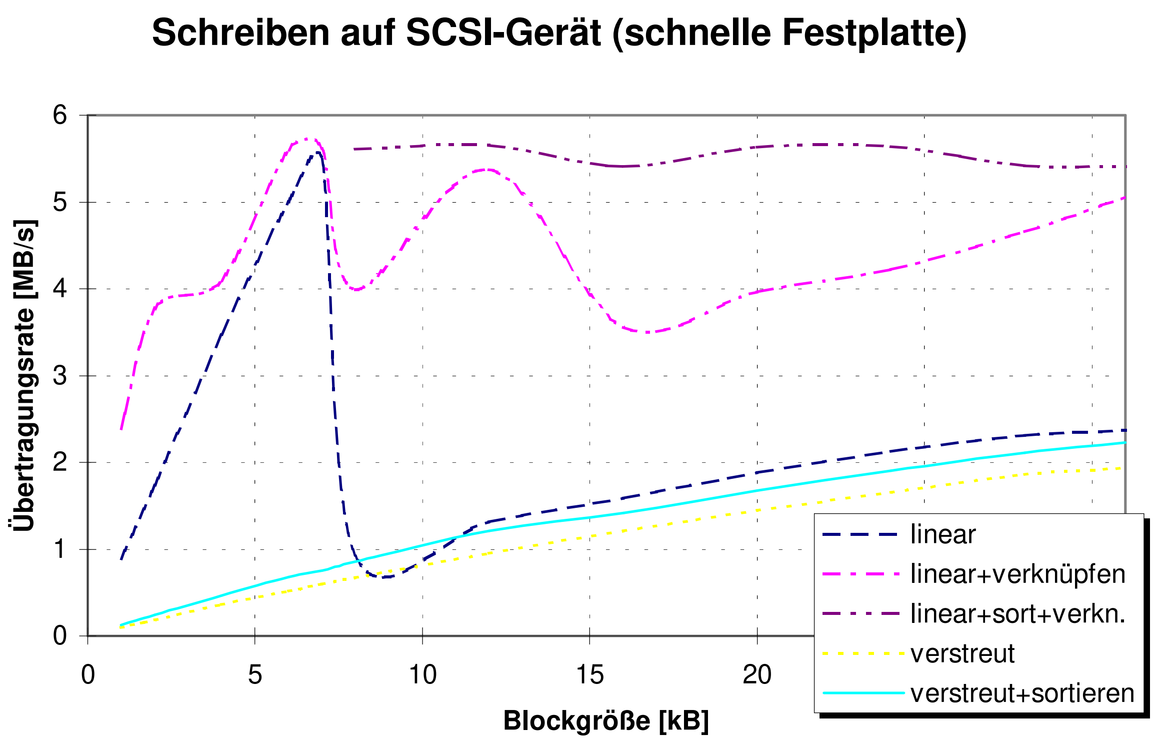

Abbildung V-4: Messungen unter L3: Schreiben auf IBM DORS 32160

Das liegt daran, daß auch dann, wenn der SCSI-Treiber im Moment keine neuen Aufträge entgegennehmen kann (kein Speicher mehr für Schreibpuffer verfügbar), immer die volle Warteschlange durchlaufen wird. Wenn Schreibauftrag 5 wegen Speichermangel erfolglos war, kann der SCSI-Treiber bei Auftrag 6 schon wieder ein wenig Platz haben und nimmt diesen Auftrag auf, obwohl erst Auftrag 5 an der Reihe wäre.

Hierbei handelt es sich ohne Zweifel um einen Fehler im Testprogramm, der andererseits hervorragend aufzeigt, wie fatal sich schon kleine Änderungen in der Auftragsreihenfolge auf die Gesamtperformance auswirken können. Aus linearen Zugriff werden wahlfreie Zugriffe, die erheblich längere Bearbeitungszeit erfordern.

Erschwert wird die Ausmessung der Schreibleistung übrigens durch den Einfluß des Schreibcaches der SCSI-Platten. Bei einer Benutzung des Treibers unter L4 sollten sich die Phänomene nicht einstellen, da dann die Pufferverwaltung nicht im SCSI-Treiber stattfindet.

Für eine grobe Abschätzung wurde eine Meßreihe vom Betrieb der IBM DORS 32160 unter Linux aufgenommen. Da sich die Pufferung von Blockgeräten nicht ohne weiteres abschalten läßt, muß auf Messung der Schreibrate verzichtet werden. Bei den Meßwerten für Lesen zeigen sich erwartete Ergebnisse. Die hohe Lesegeschwindigkeit bei kleinen Blöcken ist darauf zurückzuführen, daß Linux erst wartet, bis mehrere Aufträge in der Warteschlange vorhanden sind und somit leichter mehrere Zugriffe zusammenfassen kann. Daß die Leseraten insgesamt unter denen von L3 liegen, ist anhand der größeren Overheads bei Linux zu erklären (längerer Weg der Daten, Bufferverwaltung, andere parallellaufende Prozesse).

Größe [Byte]	Anzahl	Lesen linear [s]	[MB/s]
1024	32768	7,09	4,73
2048	16384	7,31	4,59
4096	8192	7,31	4,59
8192	4096	7,19	4,67
16384	4096	13,65	4,92
24576	4096	21,71	4,64
31744	4096	36,17	3,59
32768	8192	66,34	4,05

Tabelle V.7: Messungen unter Linux: Lesen von IBM DORS 32160

Zum Schluß sollte noch untersucht werden, wie sich der Treiber an einem langsameren SCSI-Gerät verhält. Die etwas betagte SCSI-Platte Quantum LPS52S beherrscht nur den asynchronen Übertragungsmodus.

Größe	Anzahl	Lesen linear				Lesen verstreut			
		ohne Verknüpf.		mit Verknüpf.		ohne Sort.		mit Sort.	
[Byte]		[s]	[MB/s]	[s]	[MB/s]	[s]	[MB/s]	[s]	[MB/s]
1024	8192	26,65	(0,31)	9,66	(0,87)	186,26	(0,05)	146,74	(0,06)
2048	4096	13,56	(0,62)	9,53	(0,88)	95,56	(0,09)	75,32	(0,11)
4096	2048	9,94	(0,84)	9,55	(0,88)	52,34	(0,16)	40,61	(0,21)
8192	1024	9,69	(0,87)	9,56	(0,88)	31,09	(0,27)	25,63	(0,33)
16384	512	9,60	(0,87)	9,65	(0,87)	20,20	(0,42)	17,96	(0,47)
24576	256	7,26	(0,87)	7,22	(0,87)	12,41	(0,51)	11,28	(0,56)
31744	256	13,34	(0,61)	10,63	(0,76)	14,47	(0,56)	14,00	(0,58)

Tabelle V.8: Messungen unter L3: Lesen von Quantum LPS52S

Hier zeigt sich, daß bei langsameren SCSI-Geräten die maximalen Datenrate schon bei kleineren Blockgrößen erreicht wird und der Einfluß der Zusammenfassung von Blöcken stärker ist.

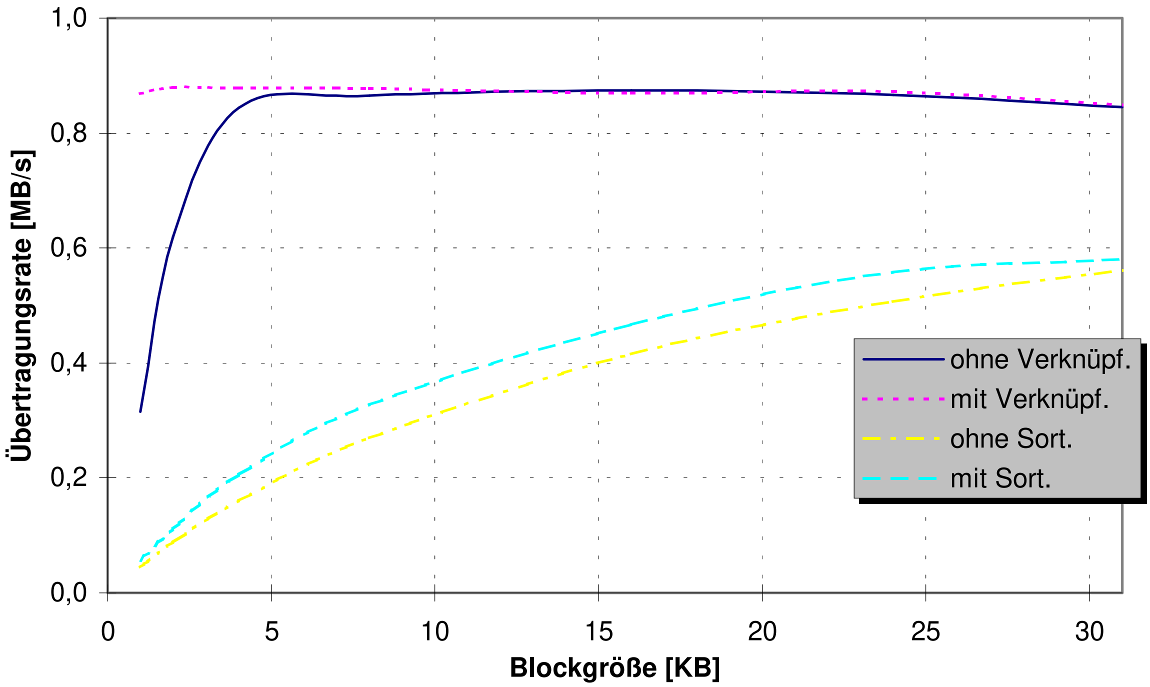

Abbildung V-5: Messungen unter L3: Lesen von Quantum LPS52S

V.3 Bewertung der Meßergebnisse

Alle Messungen an beiden Treibern zeigen die Brauchbarkeit des verwendeten Konzepts. Trotz Beschränkung auf einen ausführbaren Thread lassen sich keine gravierenden Nachteile im Vergleich mit Linux feststellen. Die einzelnen Meßreihen in Abhängigkeit der erlaubten Optimierungen des SCSI-Treibers und speziell der Einbruch beim Schreiben bekräftigen die Forderung nach Aufträgen mit großen zusammenhängenden Blöcken, um den Verwaltungsaufwand auf dem SCSI-Bus minimal zu halten.

VI Schlußfolgerungen und Ausblick

Das Ziel, die Portierung des Linux-SCSI-Treibers ohne Änderungen am hardwareabhängigen Teil wurde, wurde erreicht. Der SCSI-Treiber bewies seine Lauffähigkeit und Stabilität in der Testphase an zwei verschiedenen SCSI-Geräten. Da die Implementierung eigentlich auf L4 laufen soll, bleibt abzuwarten, ob sich beim Übergang auf diese Plattform Probleme einstellen. Wie im vorherigen Abschnitt zu erkennen war, ist der Definition der Schnittstelle größte Aufmerksamkeit zu widmen, um unnötige SCSI-Aktivitäten zu verhindern.

Noch nicht bis ins Detail geklärt ist das Zusammenwirken von mehreren SCSI-Geräten am selben Bus. Für die Beurteilung dessen sind umfangreichere Messungen erforderlich, welche sich aus Zeitgründen nicht durchführen ließen. Ebenfalls mußte auf einer Testung der bereits implementierten SCSI-CD-ROM-Ansteuerung verzichtet werden, eine einwandfreie Funktion ist aber aufgrund der Parallelen im Programmcode für Festplatten und CD-ROMs sehr wahrscheinlich.

Für diesen SCSI-Treiber wurde eine wichtige Vereinfachung im Vergleich zu Linux getroffen: Immer nur ein Partner ist in einem Zeitabschnitt mit dem SCSI-Treiber verbunden, im Gegensatz zu Linux, wo mehrere Prozesse zur selben Zeit auf dem gleichen SCSI-Gerät oder verschiedenen Geräten am gleichen Hostadapter agieren können. Unter L4 wird aber nur das echtzeitfähige Dateisystem direkten Kontakt mit dem SCSI-Treiber haben. Das Kooperationsverhalten des SCSI-Treibers im Zusammenhang mit mehreren SCSI-Geräten wird dann größtenteils von der Art der Implementierung der Schnittstelle abhängen. Das für den Test verwendete GDP-Interface unter L3 sollte jedenfalls verworfen werden, da eine Pufferverwaltung im SCSI-Treiber nichts zu suchen hat.

Ferner macht sich die vollständige Implementierung der Steuerung des SCSI-Treiber notwendig. Alle nötigen Vorbereitungen dazu sind getroffen, die benötigten Programmfragmente aus Linux sind bereits im Treiber enthalten und müssen nur noch angesteuert werden. Im einzelnen sind das

- die *ioctl*-Steuerung
- die Rückmeldung wichtiger Statusinformationen des Treibers an den Partner (Blockgröße des Mediums, Lage und Größe der Partitionen auf Festplatten)
- verbesserte Fehlerbehandlung

VII Zusammenfassung

Ziel dieser Belegarbeit war die Portierung des funktionstüchtigen SCSI-Treibers von Linux auf das Betriebssystem L3/L4. Die Lösung sollte universell sein, d.h. wenn möglich ohne Änderung der hardwareabhängigen Schicht.

Als Ausgangspunkt diente die Implementierung einer Emulation für einen Netzgerätetreiber [Stange 96]. Auf diese Arbeit aufbauend wurde zuerst die Busmasterfähigkeit des dort entwikkelten Netzgerätetreibers hergestellt. Dabei zeigte sich die Notwendigkeit der Kenntnis der physischen Adresse eines Datums, das von der DMA-Einheit übertragen werden soll. Da der verwendete Linux-Treiber von der Übereinstimmung von logischer und physischer Adresse ausgeht, wie sie in L3 nicht vorhanden ist, machte sich das Hinzufügen einer zusätzlichen L3-Kernfunktion notwendig. Bis auf diese Funktion war keine Änderung am L3-System notwendig.

Die mit der Netzgerätetreiber-Emulation gewonnenen Erfahrungen konnten bei der Entwicklung des SCSI-Treibers genutzt werden. Auch hier findet DMA-Betrieb statt, es gelten somit die gleichen Gesetze wie beim L3-Treiber. Erschwerend kam hier hinzu, daß der Linux-SCSI-Treiber sehr stark mit dem Kern verwachsen ist (Scheduler-Aufrufe) und daß mit einem Treiber mehrere verschiedene Blockgerätearten angesteuert werden können. Das machte leichte Änderungen am Ein-Thread-Konzept nötig. Der L3-SCSI-Treiber hat nun zwar prinzipiell den gleichen Aufbau wie der Netzwerktreiber (Hauptschleife mit Ereigniserfassung und -behandlung), unterscheidet sich aber im Detail (mehrere Warteschlangen zu verwalten, Ereignisbehandlung auch außerhalb der Hauptschleife).

Letztendlich waren dennoch nur *wenige* Änderungen am Linux-SCSI-Treiber bzw. *keine* Änderungen am hardwareabhängigen Teil notwendig. Der zum Testen verwendete Hostadapter Asus SC200 mit NCR53c810 wies zudem noch die Besonderheit des ladbaren SCSI-Programm-Codes auf. Selbst diese Spezialität erwies sich nicht als Fallstrick. Allerdings soll hier nicht unerwähnt bleiben, daß sich nicht alle Treiber für SCSI-Hostadapter ohne Änderung in den L3-Treiber integrieren lassen: Nimmt der Treiber unmittelbaren Bezug auf schwer nachzubildende Linux-Kernelvariablen, wird eine Anpassung der entsprechenden Stelle im Treiber notwendig. Dieser Fall sollte aber eher die Ausnahme darstellen.

VIII Glossar

ATA-Interface Mittlerweile offizielle Bezeichnung für den **IDE**-Standard zur Anbindung von Festplatten und CD-ROMs. Findet große Verbreitung speziell in billigen PCs.

Block aus der Sicht des Linux-Kerns kleinste Einheit eines Blockgerätes. Alle Blökke desselben Gerätes haben die gleiche Länge und sind logisch linear angeordnet.

blockorientiertes Gerät *block device*; Datenaustausch mit Gerät erfogt über Blöcke. Wahlfreier Zugriff auf die Daten. Blockorientierte Geräte erfahren unter Linux fast immer eine Zwischenspeicherung für lesenden und schreibenden Zugriff.

Direktansteuerung für Blockgeräte Unter Linux können Blockgeräte direkt, d.h. ohne Umweg über das Dateisystem, angesprochen werden. Der Zugriff erfolgt dabei blockweise und nicht dateiweise.

logische Blockgröße Größe jedes Blockes auf einem Blockgerät, wie sie Anwendungsprogramme benutzen. Ist immer ein Vielfaches der physischen Blockgröße.

Puffer Teil der Linux-Pufferverwaltung (siehe **S. 17:** Bufferverwaltung - Cache dynamischer Größe). Besteht aus Steuerblock (*buffer header*) und Datenblock. Die Steuerdaten beinhalten Blocknummer und Blockanzahl sowie einen Verweis auf den nächsten Buffer. Ein Puffer entspricht einem logischen Block auf dem Blockgerät.

Busy-Waiting Aktives Warten: Eine Schleife mit leerem Schleifenkörper wird solange durchlaufen, bis ein extern auftretendes Ereignis zum Abbruch führt. Dies ist für Echtzeit- und Multitaskingbetriebssysteme sehr schädlich, da die Prozessorzeit für "Nichtstun" verbraucht wird.

DMA *direct memory access*; Rechnerkomponente zum Kopieren von Daten zwischen Hauptspeicher und externen Komponenten ohne Mitwirkung der CPU

Hardwareinterrupt Unterbrechnung (*interrupt*) durch ein Ereignis der Hardware

IOCTL *input output control* Systemaufruf zum Setzen bestimmter Eigenschaften von Hardwarekomponenten

physische Blockgröße Größe jedes Blockes auf einem Blockgerät, mit dem das Medium seitens der Hardware formatiert ist.

Request Ein bestimmter **Auftrag** gekennzeichnet durch die Operation (Lesen, Schreiben, vorrausschauend Lesen oder Schreiben), die erste Blocknummer, die Anzahl der Blöcke und den Verweis auf einen Datenpuffer. Ein Request kann in mehreren Teilschritten ausgeführt werden.

SCSI *Small Computer System Interface*; Schnittstellensystem zur Verbindung von externen Komponenten unterschiedlichster Leistungsfähigkeit und Anschlußart; gilt im Gegensatz zum ATA als Profi-Lösung im PC-Bereich.

zeichenorientiertes Gerät *character device*; Datenaustausch mit dem Gerät erfolgt zeichenweise. Sequentieller Zugriff. Caching ist nach FIFO-Prinzip aufgebaut.

IX Literaturverzeichnis

[Beck 96] Beck, M., Böhme, H., Dziadzka, M., Kunitz, U., Magnus, R., Verworner, D.: Linux-Kernel-Programmierung, Algorithmen und Strukturen der Version 1.0; 2.Auflage; Addison-Wesley, Bonn 1995

[Becker 96] Becker, D.: 3Com 3c590/3c595 „Vortex“ Ethernet Driver for Linux; Quellcode, Version 0.28c, 1996; http://cesdis.gsfc.nasa.gov/linux/drivers/vortex.html

[Hohmuth 96a] Hohmuth, M., Schalm, M., Wolter, J.: L3 Documentation; TU-Dresden, 1996; http://www.inf.tu-dresden.de/~mh1/l3/l3doc/

[Hohmuth 96b] Hohmuth, M.: Linux-Emulation auf einem Mikrokern; TU-Dresden, 1996; http://www.inf.tu-dresden.de/~mh1/prj/linux-on-l4/diplom/

[Johnson 95] Johnson, M.: Writing Linux Device Drivers; from a talk given at Spring DECUS '95 in Washington, DC.; http://www.redhat.com/~johnsonm/devices.html

[Liedtke 96] Liedtke, J.: L4 Reference Manual Version 2.0; Arbeitspapier 1021, 4.9.1996; http://www.inf.tu-dresden.de/~mh1/l3/l4refx86.ps.gz

[Linux 96] Linux Kernel-Sourcen Version 2.0.23; ftp://ftp.inf.tu-dresden.de/pub/linux/kernel-funet/v2.0/

[Schmidt 93] Schmidt, F.: SCSI-Bus und IDE-Schnittstelle; Addison-Wesly, Bonn 1994

[Stange 96] Stange, R.: Systematische Übertragung von einem monolithischen Betriessystem auf eine mikrokernbasierte Architektur; TU-Dresden 1996

[Stiller 93] Stiller, A.: Schattenseiten und Bremsklötze; c’t 3/93 S. 140; Verlag Heinz Heise 1994