Großer Beleg

zum Thema

Portierung des DROPS Device Driver Environment (DDE) für Linux 2.6 am Beispiel des IDE-Treibers

Marek Menzer
mm19@inf.tu-dresden.de
Technische Universität Dresden
Fakultät Informatik

29.9.2003

Inhaltsverzeichnis

Inhaltsverzeichnis.......... 2

Kapitel 1 – Einleitung.......... 3

Kapitel 2 – Grundlagen und Stand der Technik.......... 4

- 2.1 DROPS.......... 4
- 2.2 Linux.......... 4
- 2.3 DDE.......... 4
- 2.4 IDE-Treiber in verschiedenen Betriebssystemen.......... 6
- 2.5 IDE in Linux 2.6.......... 7
- 2.6 Generic_blk.......... 9

Kapitel 3 – Entwurf.......... 11

- 3.1 IDE-Kern.......... 12
- 3.2 Blocktreiber.......... 12
- 3.3 Abbildung von generic_blk auf Linux-Blocktreiber.......... 13
- 3.4 Request-Ablauf.......... 14

Kapitel 4 – Implementierung.......... 16

- 4.1 DDE2.6.......... 16
- 4.2 IDE-Treiber.......... 21
- 4.3 Stolpersteine.......... 22

Kapitel 5 – Leistungsbewertung.......... 23

Kapitel 6 – Zusammenfassung und Ausblicke.......... 25

Glossar.......... 26

Literaturverzeichnis.......... 27

Kapitel 1 – Einleitung

An der TU Dresden wird zur Zeit DROPS, das Dresden Real Time Operating System, entwickelt. Dieses Betriebssystem soll auch Blockgeräte ansprechen können. Für SCSI-Geräte existiert schon ein Blockgerätetreiber. Allerdings sind diese wesentlich weniger verbreitet als die preiswerteren IDE-Laufwerke. Doch ein IDE-Treiber fehlt in DROPS. Es ist den Entwicklern von DROPS und dessen Anwendungen somit nicht möglich, Programme von einer IDE-Platte zu starten oder Log-Dateien zu speichern, um sie später zu analysieren. Auch die Projekte zur Videobearbeitung, -aufzeichnung und -anzeige sind darauf angewiesen, ihre enormen Datenmengen beim Systemstart in den begrenzten Arbeitsspeicher zu laden, um damit arbeiten zu können. Es ist meine Aufgabe, einen IDE-Treiber für DROPS zu erstellen.

Es ist angedacht, für DROPS später ein komplettes Block Device Driver Framework zu besitzen, welches sämtliche Blocktreiber unter einer Schnittstelle vereint. Der IDE-Treiber sollte deshalb schon auf seine Wiederverwendbarkeit in diesem Framework hin ausgelegt werden.

Aufbau dieses Dokumentes

Nach dieser Einleitung werden, soweit es für das Verständnis dieser Arbeit nötig ist, einige Systeme vorgestellt. Danach wird auf Vorgehensweisen und Entscheidungen eingegangen, die für die Realisierung der Arbeit wichtig sind. Im Kapitel 4 folgt dann ein Überblick über die konkrete Implementierung. Besonderes Gewicht wurde hier auf Abweichungen vom in Kapitel 3 vorgesehenen Weg gelegt. Im Weiteren geht es dann um die Leistungsfähigkeit des neuen Treibers. Außerdem folgt ein Kapitel mit einer Zusammenfassung der Arbeit und Ausblicken für die Zukunft. Zum Abschluß wurden ein Glossar und eine Liste von Quellen erstellt, die bei der Arbeit hilfreich waren beziehungsweise dem geneigten Leser die Vertiefung in das ein oder andere Detail ermöglichen.

Kapitel 2 – Grundlagen und Stand der Technik

2.1 DROPS

Wie bereits erwähnt, ist DROPS [DRO] das an der TU Dresden entwickelte Betriebssystem. Diese Entwicklung ist aber nicht abgeschlossen. Eine noch fehlende Komponente ist ein IDE-Treiber.

DROPS basiert auf dem L4-Mikrokern, beziehungsweise seinem hier entwickelten Pendant FIASCO. Der Kern selbst übernimmt keinerlei Arbeit am System. Er stellt lediglich den Schutz der Adreßräume sicher, sorgt für die Kommunikation und stellt Aktivitätsträger zur Verfügung. Aktivitätsträger sind Prozesse und ihre Threads. Jeder Treiber ist in der Konzeption von DROPS, und der von Mikrokernen im Allgemeinen, ein eigener Prozeß. Die Kommunikation zwischen diesen Prozessen findet in Form von IPCs statt. Speichermanager und Scheduler sind ebenfalls eigene, im Übrigen austauschbare, Prozesse. Ein IDE-Treiber wäre auch einer.

2.2 Linux

Linux [LIN] ist ein Betriebssystem, welches Anfang der 1990er Jahre von Linus Torvalds als UNIX-Derivat programmiert wurde. Es hat in den letzten Jahren einen hohen Bekanntheitsgrad erreicht, und ist zu einem Konkurrenten zu kommerziellen Betriebssystemen geworden. Linux – das ist sind der Linux-Kernel und eine, je nach Anbieter und Zielstellung variierende, Anzahl von Zusatzprogrammen. Der Linux-Kernel ist kein Mikrokern, sondern ein monolithischer. Dabei sind sämtliche Treiber in diesen integriert, ob nachladbar oder nicht, und er stellt so einen großen Prozeß dar. Es ist damit auch nicht so einfach möglich, einem Treiber eine neue Schnittstelle für Nutzerprogramme zu geben. Er muß aus seiner bestehenden Umgebung herausgelöst und in eine neue eingesetzt werden. Dazu dient das DDE.

2.3 DDE

Das DDE (Device Driver Environment) [HEL01] ist eine Bibliothek, die es gestattet, Linux-Treiber unter DROPS laufen zu lassen. Dazu emuliert es Linux-interne Funktionen und Variablen. Zugriffe auf den PCI-Bus werden dabei über eine zentrale Instanz geleitet – den I/O-Server. Denn es muß gewährleistet sein, daß jedes PCI-Gerät zu jedem Zeitpunkt von nur einem Prozeß benutzt wird.

Im bisherigen DDE sind viele Basisfunktionen des Linux-Kernel enthalten. Das sind Funktionen zur Verwaltung von Prozessorzeit, Interrupts, Speicherbereichen, PCI und Prozessen einschließlich deren Scheduling und Synchronisationsmechanismen. Es folgt nun ein kleiner Überblick über die einzelnen Bereiche.

Prozessorzeit

Um eine einheitliche Zeitbasis zu erhalten bekommt jeder Prozeß die Kernel-Info-Page von L4 in seinen Adreßraum eingeblendet. Darin ist die Variable `jiffies` enthalten, welche – abhängig von der Architektur – in bestimmten Zeitabständen um eins erhöht wird. Sie repräsentiert somit die Echtzeit. `jiffies` ist identisch zu dem in Linux. Es ist auch möglich auf dieser Basis Timer zu generieren, die zu einem bestimmten Zeitpunkt die angegebene Funktion aufrufen. Dazu dienen die Funktionen `add_timer`, `del_timer` und `mod_timer`.

Interrupts

Ein Programm welches die Interrupt-Funktionalität von DDE nutzt, besitzt automatisch einen zusätzlichen Thread pro angeforderten Interrupt. Dieser nimmt die von DROPS gemeldeten Interruptereignisse entgegen und ruft die beim Initialisieren angegebene Funktion auf – den Interrupt-Handler. Dieser Handler führt nun weitere Aktionen aus, zum Beispiel einen Empfangspuffer zu leeren. Mit den Funktionen `request_irq` und `free_irq` kann ein Programm einen Interrupt anfordern beziehungsweise wieder freigeben. Ein generelles Sperren von Interrupts per `cli` ist in DDE nicht möglich. Unter Linux wird diese Methode oft benutzt. Das ist dort aber kein Problem, da der Linuxkern sowieso der einzige ist, der das darf. In DROPS dagegen ist jeder Treiber ein separater Prozeß. Ein `cli` eines Treibers hätte zur Folge, daß alle anderen an ihrer Ausführung gehindert werden. Dies ist eine Verletzung des Schutzes der Prozesse voreinander. DDE bildet `cli` und `sti` deshalb auf ein Interrupt-Lock ab. Jeder Treiber besitzt dabei sein eigenes Lock und kann andere somit nicht beeinflussen.

Speicher

Arbeitsspeicher wird durch die normalen Linux-Funktionen `vmalloc` und `vfree` für VMEM (virtueller Speicher) sowie `kmalloc` und `kfree` für KMEM (Kernel-Speicher) angefordert und freigegeben. Der Unterschied zwischen VMEM und KMEM besteht darin, daß letzterer nicht ausgelagert werden darf und fortlaufende logische Adressen auch fortlaufende Adressen im RAM besitzen müssen. Würde der KMEM ausgelagert, und es tritt ein Interrupt ein dessen Handler sich nicht im Speicher befindet, müßte Interrupt für den Speichermanager und dann der der Festplatte angestoßen werden. Das geht eventuell nicht, weil etwa der Speichermanager eine niedrigere Priorität besitzt. Das System wäre an dieser Stelle nicht mehr lauffähig. Selbst wenn alle Prioritäten stimmen, wäre eine inakzeptable Verzögerung die Folge. Ein weiters Problem bestünde, wenn der Speichermanager selbst nicht im Speicher ist. Das würde zu dem Paradoxon führen, daß er sich dann selbst von Platte holen muß. KMEM muß sich außerdem an fortlaufenden physischen Adressen befinden, weil sonst kein DMA möglich ist. Denn DMA arbeitet auf deren Basis.

PCI

Der Zugriff auf PCI-Geräte muß zentral gesteuert werden, da sonst Konflikte vorprogrammiert sind. Wenn beispielsweise zwei Prozesse A und B gleichzeitig auf die Festplatte schreiben wollen, und A, kurz bevor B einen Schreibbefehl gibt, das Register für den zu schreibenden Sektor ändert, tritt ein Datenverlust auf. Die zur Verhinderung einer solchen Situation nötige Koordinierungsaufgabe übernimmt der I/O-Server. Mit den Funktionen `pci_enable_device`, `pci_disable_device`, `pci_set_master` und `pci_set_powerstate` können PCI-Geräte eingerichtet werden. Mit den Funktionen `pci_read_config_{byte|word|dword}` und `pci_write_config_{byte|word|dword}` greift ein Programm auf den Configuration Space, also die eigentlichen Register, des PCI-Gerätes zu. Um ein Gerät benutzen zu können, muß es erst einmal im System gefunden werden. Dazu dienen Hilfsfunktionen wie `pci_find_device`, `pci_find_subsys`, `pci_find_slot` und `pci_find_class`. Danach muß sich ein Treiber für das Gerät an- und später wieder abmelden. Mit `pci_register_driver` und `pci_unregister_driver` kann er das tun.

Prozesse und Scheduling

Auf Grund der großen Verstrickung der jede Task beschreibenden Struktur `task_struct` mit anderen Funktionen und Makros, wurde genau diese Struktur aus Linux in DDE übernommen. Deshalb gibt es in diesem Bereich praktisch keine Einschränkungen. Ein Prozeß kann sich mit `[interruptible_]sleep_on[_timeout]` schlafen legen oder seine Prozessorzeit mit `schedule[_timeout]` freigeben. Außerdem können andere Prozesse mit `wake_up_process` oder `try_to_wake_up` aufgeweckt werden.

Synchronisation

Die zur Synchronisation dienenden Spinlocks und Semaphore sind einfach auf die entsprechenden DROPS-Pendants abgebildet. Die zur Synchronisation oft benutzten `cli` und `sti` sind, wie im Abschnitt Interrupts bereits erwähnt, durch ein Lock emuliert.

2.4 IDE-Treiber in verschiedenen Betriebssystemen

Windows

Windows [WIN] ist ein kommerzielles Produkt der Firma Microsoft und entsprechend nicht im Quellcode erhältlich. Der IDE-Treiber in Windows unterliegt einer hohen Aktualisierungsrate. Jede neue Hardware läuft innerhalb kürzester Zeit damit – in der Regel sind die passenden Treiber sogar mitgeliefert.

Linux

Linux ist relativ gut dokumentiert, im Quellcode erhältlich, frei von Urheberrechten und erfährt relativ häufig Anpassungen an neueste Hardware.

FreeBSD

FreeBSD [FRB] ist ähnlich Linux, nur etwas besser strukturiert. Dafür ist es aber schwer einen FreeBSD-Kernel ohne FreeBSD zu bekommen.

MacOS X

MacOS X [MAC] basiert auf FreeBSD und einem Mach-Kern. Im Zusammenhang mit IDE-Treibern gilt deshalb das für FreeBSD gesagte.

BeOS

BeOS ist auf Multimediaanwendungen ausgerichtet und deshalb auch echtzeitfähig. Die Firma Be [BEI] hat 2001 allerdings Konkurs anmelden und die Entwicklung von BeOS eingestellt. Es gibt einige Bestrebungen [OBE], BeOS wieder aufleben zu lassen. Eine solche Basis ist für einen zukunftssicheren Treiber aber nicht geeignet.

ModulOS

ModulOS [LHS98] ist ein freies Betriebssystem, welches im Wesentlichen von dem Studenten Luiz Henrique Shigunov entwickelt wird. Es besitzt zwar einen Blocktreiber, der arbeitet aber momentan nur mit Diskettenlaufwerken. Ein IDE-Treiber ist in Arbeit.

FluxOSKit

Der IDE-Treiber von FluxOSKit [FLX] basiert auf dem von Linux und ist damit im Sinne dieser Arbeit keine neue Variante.

2.5 IDE in Linux 2.6

Wenn bisher vom IDE-Treiber in Linux gesprochen wurde, waren immer zwei Teile gemeint. Der eigentliche IDE-Kern und der darüberliegenden Blocktreiber. Diese beiden sind sehr miteinander verstrickt. Die Schnittstelle zwischen ihnen ist viel komplexer, als die zwischen dem Blocktreiber und einem Client. Trotzdem ist es für das Verständnis der Arbeit wichtig, beide getrennt zu betrachten.

IDE-Kern

Der IDE-Kern enthält unter anderem sämtlichen Quellcode der nötig ist, um die verschiedenen Chipsätze anzusprechen. Im Unterverzeichnis `/pci` befindet sich zu diesem Zweck für jeden eine Datei. In dieser Datei sind dann Strukturen definiert, die seine Eigenschaften beschreiben – Name, Kennung auf dem PCI-Bus und verschiedene Funktionen. Zu diesen Funktionen gehören solche, die ihn initialisieren, nachschauen ob er überhaupt verfügbar ist, Einstellungen wie DMA-Modus vornehmen und solche die Informationen über Version und Zustand ausgeben. Zu diesen Informationen gehört auch, welches Laufwerk in welchem Modus betrieben wird.

Der eigentliche Daten- und Kommandotransfer findet in zwei Dateien statt, die zum einen einen DMA-Transfer aufbauen und durchführen und zum anderen die Kommandos an das entsprechende Gerät schicken. Genau genommen sind es genau diese beiden, die das IDE-Protokoll implementieren. Sie sind gewissermaßen der Kern des IDE-Kern und setzen direkt auf den Chipsatz-spezifischen Funktionen auf.

Im IDE-Kern wird nach Diskettenlaufwerk, Festplatte, CD/DVD-Laufwerk und Bandlaufwerk unterschieden, weil diese jeweils etwas unterschiedlich behandelt werden müssen. Im Fall CD/DVD-Laufwerk sogar sehr unterschiedlich, weil hier das Packet Command Set verwendet wird, welches eine Abwandlung des SCSI-Protokolles ist. Das Augenmerk liegt bei dieser Arbeit aber auf der Behandlung von Festplatten. Für jede dieser Gerätearten gibt es eine eigene Datei. Hierauf wiederum setzt eine Datei auf, die nach Laufwerken sucht und diese im System einrichtet.

Es existieren noch weitere Dateien, die sich beispielsweise um Plug'n'Play, TCQ oder das ProcFS kümmern. Diese Funktionen sind für diese Arbeit aber unwichtig. Außerdem gibt es noch solche Dateien, die nur eine Bibliotheksfunktion erfüllen oder das gesamte IDE-System initialisieren.

Letztlich ist der IDE-Kern nur eine ausführende Schicht. Er besitzt nach außen keinerlei Verwaltungsfunktion. Er ist nur ein Hilfsmittel des Blocktreibers und stellt für ihn den Bezug zwischen den abstakten Datenanfragen und der Hardware dar.

Blocktreiber

Der Blocktreiber stellt die Verbindung zwischen Dateisystem und dem IDE-Kern dar. Ihm sind jedoch auch noch andere Kerne – wie der SCSI-Kern – unterstellt. Eigentlich agiert dazwischen noch der Blockcache, welcher für das Verständnis und die grundlegende Funktion aber unwichtig ist. Die folgende Grafik soll die Zusammenhänge der verschiedenen Teilsysteme verdeutlichen.

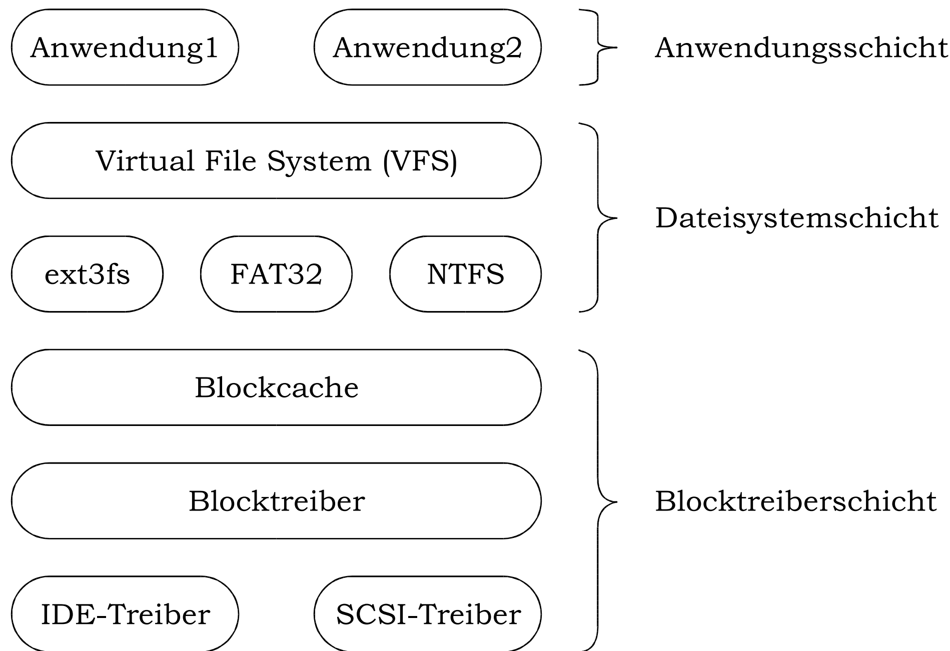

Abbildung 1 - Stellung des Blocktreibers in Linux

Das zentrale Element des Datentransports ist die BIO (block io operation). In ihr ist angegeben, an welches Gerät die Anfrage gerichtet ist, welche Sektoren betroffen sind, welche Funktion bei Beendigung aufgerufen werden soll und wo die zugehörigen Daten im Arbeitsspeicher zu finden sind beziehungsweise wohin sie sollen. Für letztgenannte Information enthält die BIO eine Liste von BIO_VECs. Das sind Vektoren, die auf eine bestimmte Page zeigen und außerdem Angaben zum Anfangspunkt des Puffers innerhalb dieser Page und zur Länge des Puffers enthalten. Ein BIO_VEC kann auch über die Grenze einer Page hinauszeigen, solange die nachfolgende auch zu diesem gehört. Die BIO zieht sich durch alle Schichten. Sie wird von der Anwendung erstellt und vom Blocktreiber nach einigen Verarbeitungsschritten zum IDE-Kern durchgereicht.

Dennoch ist eine BIO noch in eine größere Struktur eingehüllt – den Request. Das ist nötig, weil die BIO während ihrer Bearbeitung eigentlich nicht verändert werden darf. Trotzdem sind zur Bearbeitung Notizen nötig. Zum Beispiel muß festgehalten werden, wo innerhalb des gesamten Puffers sich der Treiber gerade befindet, das heißt welche Teile schon abgearbeitet sind. Ein Request kann mehrere BIOs enthalten, die miteinander verkettet sind. Es ist somit möglich, daß der Blocktreiber BIOs über eine gewisse Zeit sammelt, um dann alle gemeinsam abzusenden. Ein Richtwert für diese Vorhaltezeit ist drei Millisekunden, in manchen Systemen zehn. Stehen innerhalb dieser Zeit aber mehr als vier BIOs bereit, wird der Request sofort abgesendet. Somit gibt es in Hochlastsituationen keine künstliche Verzögerung.

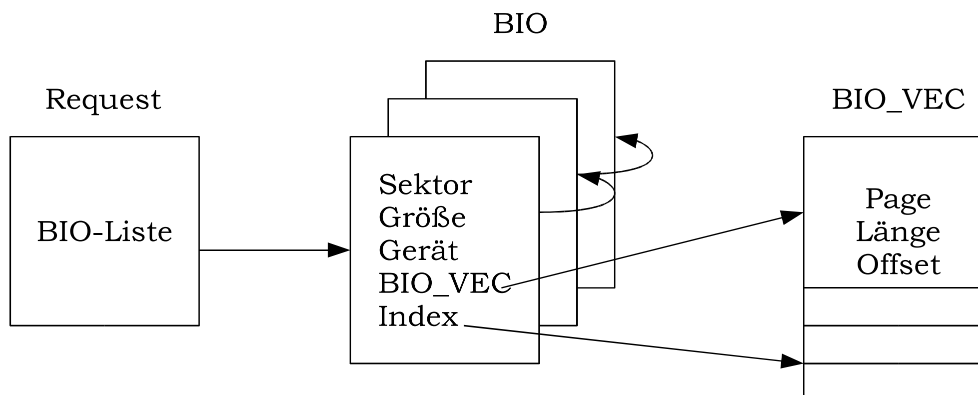

Abbildung 2 - Zusammenhang zwischen Request, BIO und BIO_VEC

Sämtliche Requests werden in eine Request Queue eingestellt. Für jedes Gerät existiert eine solche Warteschlange. Das Einstellen und Ausgeben übernimmt ein sogenannter I/O-Scheduler. Dieser kann entscheiden, einen bestimmten Request einem anderen vorzuziehen. Das ist hauptsächlich für Echtzeitanwendungen interessant, bei denen dann beispielsweise versucht wird, die Deadlines aller Requests einzuhalten. Im einfachsten Fall findet aber der NOOP-Scheduler Verwendung, der einfach alle Requests in ihrer Ankunftsreihenfolge abarbeitet. Eine besondere Aufgabe der I/O-Scheduler ist allerdings, daß sie versuchen, mehrere kleine BIOs zu einer großen zusammenzufassen. Wenn eine Anwendung beispielsweise die Sektoren 511-515 anfordert und Anwendung B die Sektoren 500-510, dann können diese beiden Anfragen zusammengefaßt werden – dem IDE-Kern ist es schließlich egal wer die Daten letztlich bekommt. Solche Fälle sind nicht unüblich, und es ergibt sich somit ein großes Einsparungspotential. Denn ein großer Transfer ist wesentlich schneller abgearbeitet als zwei kleine, weil zum einen der Kommando-Overhead eines Transfers gespart wird. Zum anderen muß so nicht erst gewartet werden, bis die Folgesektoren erneut unter dem Lesekopf der Platte erscheinen, was selbst bei schnellen Platten acht Millisekunden dauert. Für jede Warteschlange kann ein anderer I/O-Scheduler gewählt werden, was auch im laufenden Betrieb kein Problem ist.

2.6 Generic_blk

Generic_blk ist eine definierte Schnittstelle, um Blockgerätetreiber in Client-Server-Manier anzusprechen. Sie wurde als Standartschnittstelle für DROPS entwickelt. Zum Transfer der Kommandos und Daten werden intern IPCs – das zentrale Kommunikationsprimitiv in L4 – verwendet. Die Anzahl der zur Verfügung stehenden Funktionen wurde bewußt klein gehalten, um dem Nutzer ein möglichst einfaches und überschaubares, andererseits aber auch flexibles Instrument an die Hand zu geben. Im einzelnen sind das (aus Sicht des Client):

`open_driver`	der Treiber (Server) mit dem angegebenen Namen wird gesucht und für den Client geöffnet
`close_driver`	der angegebene Treiber wird für den Client wieder geschlossen
`put_request`	dem Treiber wird ein Request zugesandt
`do_request`	dem Treiber wird ein Request zugesandt und auf dessen Bearbeitung gewartet

Außerdem gibt es weitere Funktionen beispielsweise zur Fehlerabfrage oder Echtzeitbehandlung. Diese sind aber für das allgemeine Verständnis dieser Arbeit unnötig oder werden überhaupt nicht benutzt. Dem interessierten Leser sei die entsprechende Dokumentation [REU01] empfohlen.

Der eigentliche Zugriff auf Daten findet in Form von Requests statt. Ein Request enthält im Wesentlichen folgende Informationen:

driver	der Treiber, der diesen Request erhalten soll (dieser wurde vorher mit open_driver geöffnet)
cmd	ob es ein Lese- oder Schreibrequest ist
device	für welches Gerät der Request bestimmt ist
block	ab welchem Block das Gerät gelesen oder geschrieben werden soll
count	wie viele Blocks gelesen oder geschrieben werden sollen
sg_list	Scatter/Gather-Liste, die den zu benutzenden Puffer angibt
sg_num	Anzahl der Elemente der Scatter/Gather-Liste
sg_type	Typ der Scatter/Gather-Liste
done	eine Funktion, die nach Beendigung des Requests aufgerufen werden soll

Auf die Scather/Gather-Listen wird in Abschnitt 3.3 noch näher eingegangen.

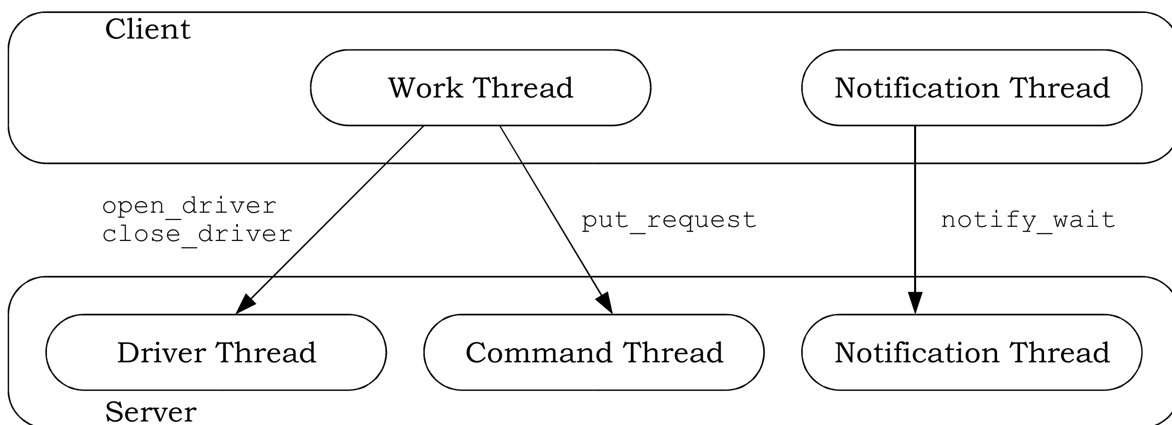

Abbildung 3 - Aufbau des generic_blk-Interfaces

Da DICE [DIC] – das hier verwendete Werkzeug zur automatischen Generierung von Schnittstellen – keine asynchronen Interfaces direkt unterstützt, verwendet generic_blk zur Kommunikation verschiedene Threads. Ein Server enthält zumindest einen Driver-Thread. Dieser ist derjenige, der beim Namensdienst gemeldet ist. Über den Namensdienst erfährt der Client mittels open_driver die ID dieses Threads, die des Command-Threads und die des Notification-Threads. Der Command-Thread ist derjenige, der Kommandos wie bei put_request erhält. Beim Notification-Thread können sich Clients Benachrichtigungen, beispielsweise über bearbeitete Requests, abholen. Es ist nicht zwingend, daß jeder Thread ein eigenständiger ist. Es ist beispielsweise auch möglich, daß Command- und Driver-Thread identisch sind. Auf jeden Fall muß jeder Client seinen eigenen Notification-Thread bekommen.

Kapitel 3 – Entwurf

In DROPS soll es in Zukunft ein Block Device Driver Framework geben. Dieses wird alle Blockgerätetreiber verwalten und über eine gut definierte Schnittstelle den Anwendungen – in erster Linie aber den Dateisystemen – anbieten. Außerdem wird es Möglichkeiten geben, in die Bearbeitung der Requests einzugreifen. Das ist nötig und wichtig, weil DROPS mit Echtzeitfähigkeiten aufwarten kann, die von den Blocktreibern natürlich ebenso unterstützt werden müssen. Es existieren schon einige Projekte – wie generic_blk, SCSI-Treiber [MEH98], DSI [LHR01] und Echtzeitschedulingverfahren [RP03] – die alle Teil dieses Frameworks sein können. Teilweise werden sie dazu aber in Details geändert werden müssen. So weist generic_blk in der Behandlung von Partitionen noch Lücken auf, und der SCSI-Treiber enthält noch seine eigene Blocktreiberschicht. Der IDE-Treiber soll von Anfang an auf dieses Framework hin ausgerichtet sein.

Für die Implementation eines IDE-Treibers gibt es mehrere Ansätze. Es kann:

(a) ein komplett neuer geschrieben werden,
(b) Teile aus einem bestehenden genommen und daraus ein neuer geschrieben werden, oder
(c) ein bestehender komplett genommen und eine Emulation seiner Originalumgebung programmiert werden.

(a) kommt für diese Arbeit nicht in Frage, da die Dokumentationen der einzelnen Chipsätze sehr schwer erhältlich sind. (b) setzt voraus, daß dieser im Quellcode erhältlich und frei von Urheberrechten ist. (c) setzt wiederum voraus, daß zumindest seine Umgebung gut dokumentiert ist. Dies ist bei kommerziellen Treibern aber praktisch nie der Fall. Windows scheidet damit aus, und es muß nach einer freien Variante Ausschau gehalten werden.

Nach näherer Betrachtung bleibt nur die Wahl zwischen Linux und FreeBSD. Vor- und Nachteile halten sich, im Kontext dieser Arbeit, in etwa die Waage. Ich habe mich für den IDE-Treiber aus Linux entschieden, weil es für Linux schon eine Emulationsumgebung unter DROPS gibt – das DDE. Dieses ist für den Linux-Kernel 2.4 ausgelegt. Die Version 2.6 ist aber gerade in der letzten Testphase, und deshalb ist es zu überlegen, eventuell gleich diese neue Version zu verwenden. Dafür wäre dann eine Anpassung des DDE auf die Version 2.6 des Linux-Kernel notwendig.

Der IDE-Treiber von Linux 2.6 hat gegenüber dem in Linux 2.4 sehr große Änderungen erfahren. Es wurde ein neues Objekt eingeführt, welches als Basis jedes Datentransfers dient – die BIO. Dieses Objekt ist eine große Erleichterung besonders im Umgang mit DMA, da es direkt mit Pages statt mit virtuellen Adressen arbeitet. Weiterhin wurden bisher lediglich 32 Bit für die Repräsentation der Sektornummern benutzt. Bei der üblichen Sektorgröße von 512 Byte lassen sich damit heute schon fast erreichte zwei Terabyte ansprechen. Um für sehr viel größere Geräte gerüstet zu sein, ist diese Repräsentation auf 64 Bit erweitert worden. Außerdem wurde viel Verwaltungsfunktionalität vom IDE-Kern in den Blocktreiber verlagert. Der neue Kern kennt nur noch physische Platten, keine Partitionen mehr. Das hat für diese Arbeit den Vorteil, daß die Verwaltung von Warteschlangen neu programmiert werden kann, ohne den IDE-Kern zu verändern. Dieser kann also einfach übernommen werden. Das ist enorm wichtig, wenn ein neuerer eingesetzt werden soll, weil der beispielsweise ein

wichtiges Bugfix oder die Unterstützung eines neuen Chipsatzes enthält. Es ist auch zu erwarten, daß die Weiterentwicklung von Linux 2.4 in absehbarer Zeit eingestellt wird – die von Linux 2.6 hat gerade erst begonnen. Das bedeutet, daß es für einen in einem Jahr erscheinenden Chipsatz vielleicht keine Unterstützung in Linux 2.4 mehr geben wird. Diese Last sollte ein neues Projekt nicht auf sich nehmen. Nicht zuletzt ist der IDE-Treiber in Linux 2.6 auch sehr viel aufgeräumter, übersichtlicher und besser dokumentiert. Die damit verbundene erleichterte Arbeit ist ein unschätzbarer Vorteil.

Wegen all den genannten Vorteilen entscheide ich mich also dazu, den IDE-Treiber aus Linux 2.6 nach DROPS zu portieren. Das hat aber auch einige Konsequenzen. Er ist nämlich auf die Arbeit in einer 2.6er Kernelumgebung ausgelegt, die sich in einigen Punkten von der 2.4er unterscheidet. Das bisherige DDE kommt somit nicht in Frage, denn der Anpassungsaufwand des Treibers wäre enorm. Dies sollte ja aber verhindert werden. Es bleibt also, gerade auch für andere zukünftige Projekte, nichts weiter übrig, als ein neues DDE für Linux 2.6 zu entwickeln.

3.1 IDE-Kern

Der IDE-Kern sollte idealerweise unangetastet bleiben. Dann bestünde die Möglichkeit, bei Neuerungen und Fehlerbereinigungen einfach den jeweils aktuellen Kern aus dem Linux-Kernel zu nehmen, und ohne weitere Anpassungen einzusetzen. Da er zum einen auf der Linux-Kernel-Umgebung aufsetzt, welche ja durch das DDE emuliert wird, und zum anderen nur den Blocktreiber als Schnittstelle „nach oben" benötigt, welcher ja sowieso noch zu entwerfen ist, bereitet diese Unangetastetheit auch keine Probleme. Falls es Abhängigkeiten zu nicht implementierten, ansonsten für den IDE-Kern aber unnötigen Kernel-Funktionen gibt – wie es bei sysfs der Fall ist – können diese durch leere Funktionsrümpfe emuliert werden.

Wenn der IDE-Kern unangetastet bleibt, bleibt logischerweise auch die Schnittstelle zwischen ihm und dem Blocktreiber unangetastet. Dies hat den nicht unwichtigen Vorteil, daß auch andere Kerne – wie der SCSI-Kern – einfach aus Linux genommen und mit diesem eingesetzt werden können. Insbesondere im Hinblick auf das kommende Block Device Driver Framework ist das von Vorteil.

3.2 Blocktreiber

Es gibt hier zwei Möglichkeiten. Entweder einen generic_blk-kompatiblen Blocktreiber neu zu implementieren oder den aus Linux an generic_blk anzupassen. Die Neuimplementation hat auf Grund der Komplexität des Linux-Blocktreibers sicher den Vorteil, daß dann genau bekannt ist, was im Treiber später vorgeht. Schließlich finden hier Scheduling-Entscheidungen und Warteschlangen-Verwaltung statt, die ein System schnell zum Blockieren bringen können.

Den bestehenden Blocktreiber anzupassen hat zwei wesentliche Vorteile. Zum einen kann von einer fehlerfreien Implementation der Warteschlangen ausgegangen werden, und zum anderen bietet er – wie in Kapitel 2 schon gezeigt – eine Fülle von Optimierungsmechanismen, die auch für DROPS interessant sind und deren Neu- beziehungsweise Nachimplementation einen erheblichen Aufwand bedeutet. Der hier schon vorhandene Quellcode kann sehr gut wiederverwendet werden.

Die Schnittstelle zwischen IDE-Kern und Blocktreiber erzwingt, genau eine Request-Warteschlange pro physischem Gerät (Festplatte) zu benutzen. Andere Ansätze wären eine Warteschlange pro logischem Gerät (Partition) oder eine Warteschlange pro Client, wie es von generic_blk propagiert wird. Theoretisch kann auch eine Warteschlange pro

Client benutzt, aus diesen dann jeweils Requests entnommen und in die entsprechende Warteschlange des zugehörigen physischen Gerätes eingereiht werden. Dies hätte eine erheblich kompliziertere Implementation zur Folge und würde lediglich den Vorteil bieten, daß dann eine Bandbreitenbegrenzung pro Client möglich wäre. Dieses ist mit dem bestehenden Blocktreiber und seinen I/O-Schedulern aber auch bisher schon möglich. Es läßt sich schließlich auch ein solcher konstruieren, der die Requests eines Clients solange verzögert, bis ein anderer seine zugesicherte Bandbreite erreicht.

Auf Grund der hohen Flexibilität des bestehenden Blocktreibers werde ich diesen verwenden und an generic_blk anpassen.

3.3 Abbildung von generic_blk auf Linux-Blocktreiber

Für den IDE-Treiber wäre es vorteilhaft, für jede Request-Queue – also für jedes physische Laufwerk – einen Command-Thread zu haben, anstatt insgesamt nur einen. Denn dieser muß unter Umständen warten, weil das zugehörige Laufwerk so langsam ist, daß er die Queue vollständig füllt. Ein anderer Client, der auf ein anderes Laufwerk zugreifen will, müßte so unnötigerweise warten. Es ist mit der bestehenden Implementation von generic_blk aber nicht möglich, dieses umzusetzen. Denn ein Client erhält die ID des Command-Threads schon bei Ausführung von `open_driver`. An dieser Stelle wurde aber noch nicht mitgeteilt, auf welches Laufwerk zugegriffen werden soll. Das wird ja erst beim Zugriff selbst getan. Der Ausweg ist, für jeden Client einen separaten Command-Thread zu erzeugen, wenn dieser `open_driver` aufruft.

Weiterhin ist es nötig, die Requests, die der Treiber von generic_blk bekommt, in sein Linux-Pendant umzuwandeln. Obwohl es in Linux auch Requests gibt, ist das direkte Gegenstück die BIO. Ein kurzer Überblick über beide Strukturen zeigt schnell, daß sie sich sehr ähnlich sind.

```
typedef struct l4blk_blk_request
{
  l4_uint32_t         req_handle;
  l4_uint32_t         cmd;
  l4_uint32_t         device;
  l4_uint32_t         block;
  l4_uint32_t         count;
  l4blk_stream_t      stream;
  l4_uint32_t         req_no;
  l4_uint32_t         flags;
} l4blk_blk_request_t;
```

```
struct bio {
      sector_t               bi_sector;
      struct bio             *bi_next;
      struct block_device    *bi_bdev;
      unsigned long          bi_flags;
      unsigned long          bi_rw;
      unsigned short         bi_vcnt;
      unsigned short         bi_idx;
      unsigned short         bi_phys_segments;
      unsigned short         bi_hw_segments;
      unsigned int           bi_size;
      unsigned int           bi_max_vecs;
      struct bio_vec         *bi_io_vec;
      bio_end_io_t           *bi_end_io;
      atomic_t               bi_cnt;
      void                   *bi_private;
      bio_destructor_t       *bi_destructor;
};
```

Die folgenden (wichtigsten) Elemente lassen sich direkt aufeinander abbilden:

generic_blk	BIO	Bedeutung
`device`	`bi_bdev`	das Gerät, für welches die Aktion gedacht ist
`cmd`	`bi_rw`	die auszuführende Aktion (Lesen oder Schreiben)
`block`	`bi_sector`	der Sektor/Block, bei dem die Aktion begonnen werden soll
`count`	`bi_size`	auf wie viele Sektoren/Blocks sich die Aktion bezieht

Diese Ähnlichkeit erleichtert die Implementation des Blocktreibers erheblich. Die restlichen Felder eines `l4blk_blk_request` sind gar nicht oder nur zur Verwaltung nötig.

Die restlichen Felder einer BIO sind entweder einfach mit 0 (bi_idx, bi_phys_segments, bi_hw_segments) beziehungsweise NULL (bi_next, bi_private) oder durch andere Standartwerte initialisierbar. Eine Ausnahme bilden die Felder bi_vcnt und bi_io_vec, deren Bedeutung im Folgenden erklärt wird.

Bisher sind in keiner der beiden Strukturen die eigentlichen Datenpuffer aufgetaucht. Bei einer BIO gibt es dafür die BIO_VECs, die, wie schon erwähnt, die die einzelnen Teile der Puffer darstellen.

```
struct bio_vec {
      struct page    *bv_page;
      unsigned int   bv_len;
      unsigned int   bv_offset;
};
```

Das Feld bi_io_vec zeigt auf den Beginn der Liste von BIO_VECs, die den zur BIO gehörenden Puffer bilden. bi_cnt gibt an, wie viele BIO_VECs diese Liste enthält. Um diese Liste zu bekommen, ist aber ein wenig Arbeit nötig, denn sie wird von generic_blk nicht direkt geliefert. Generic_blk verwendet eine etwas andere, aber verwandte Art diese Puffer darzustellen – die Scatter/Gather-List. Dabei gibt es zwei verschiedene Typen. Einen mit physischen Adressen und einen mit Dataspacezeigern. Letztere sind eine einfache Möglichkeit Puffer in DROPS in anderen Adreßräumen einzublenden, was aus Schutzgründen nicht direkt möglich ist. Es ist nicht Pflicht beide Typen einzusetzen. Im Folgenden wird nur auf Scatter/Gather-Lists mit physischen Adressen näher eingegangen, weil diese zur Anschauung besser geeignet sind. Sie lassen sich gut in BIO_VECs umrechnen. Bei Verwendung von Dataspacezeigern sind dazu einige Operationen mehr nötig.

```
typedef struct l4blk_sg_phys_elem
{
  l4_addr_t addr;
  l4_size_t size;
} l4blk_sg_phys_elem_t;
```

Das Feld size ist identisch mit dem Feld bv_len einer BIO_VEC. In DDE gibt es keine Page-Strukturen wie in Linux. Das Feld bv_page ist allerdings nur bei der Generierung der Scatter/Gather-Liste für die DMA-Operationen nötig, welche per blk_rq_map_sg geschieht. Diese Funktion ist Bestandteil des Blocktreibers und deshalb beliebig änderbar. Dabei wird ursprünglich die Angabe der Page in eine physische Adresse umgewandelt. Diesen Aufwand ist im neuen Blocktreiber nicht mehr nötig, weil sowieso schon eine physische Adresse vorliegt. Die Adresse kann also direkt in das Feld bv_page eingetragen werden. Das Feld bv_offset ist dann überflüssig und kann unangetastet bleiben.

Die BIO ist damit vollständig, aber noch kein fertiger Request. Dieser kann aber mit Hilfe der Funktion generic_make_request aus der BIO erstellt werden. Die Funktion sorgt auch gleich dafür, daß der eben erstellte Request in die entsprechende Request-Queue eingestellt wird. Die weitere Bearbeitung findet ab hier automatisch statt.

3.4 Request-Ablauf

Zusammenfassend sei an dieser Stelle der Weg eines Requests vom Client über den Server und wieder zurück gezeigt – Abbildung 4 soll dabei helfen.

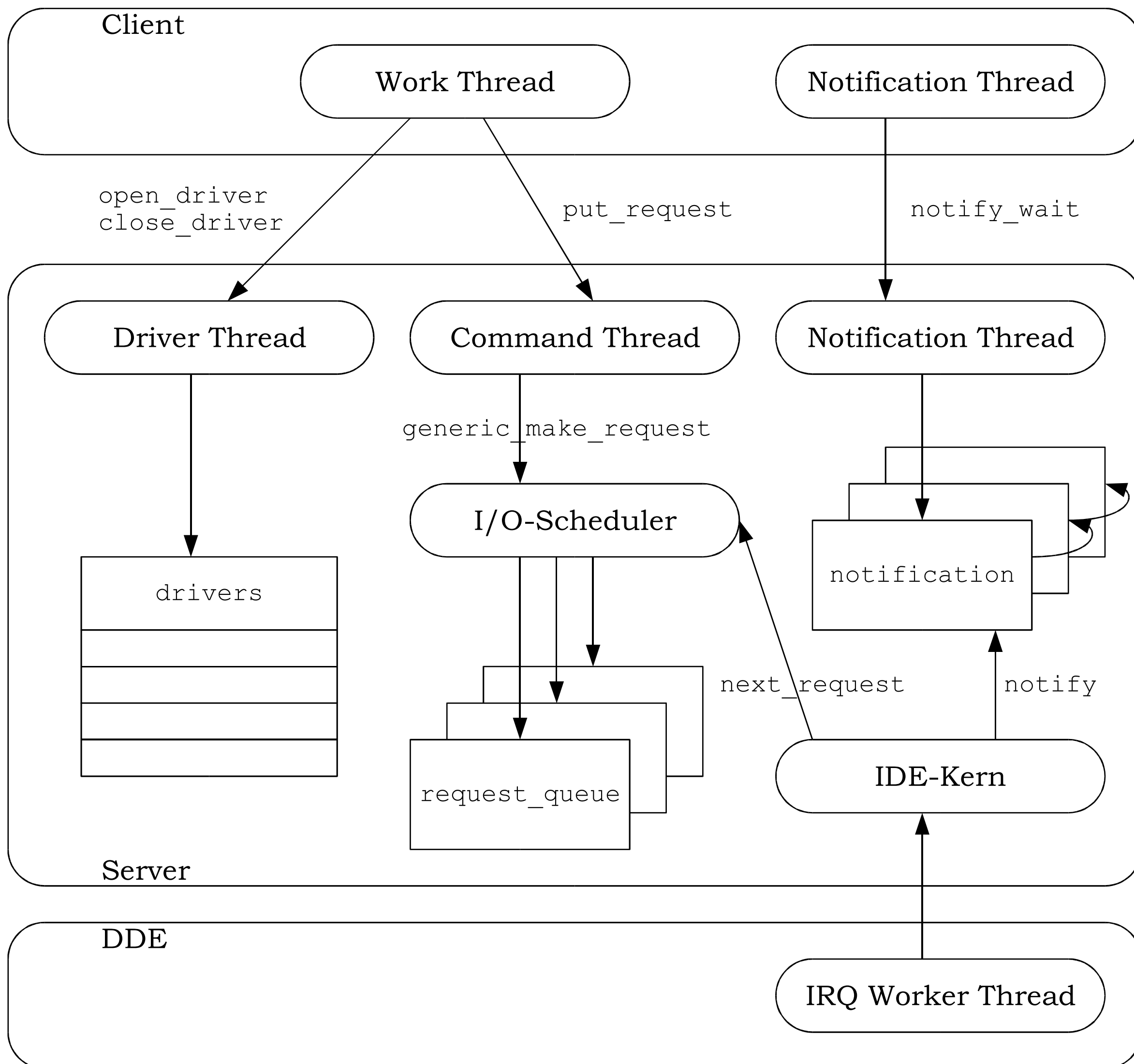

Abbildung 4 - IDE-Treiber mit DDE und Client

Der Client generiert einen Request in dem er einträgt, welche Blöcke welchen Gerätes er schreiben oder lesen möchte. Außerdem gibt er einen Puffer in Form einer Scatter/Gather-List an, in dem die zugehörigen Daten abgelegt beziehungsweise abzulegen sind. Er schickt diesen Request mit `put_request` oder `do_request` an den Server. Der Server – sprich der Blocktreiber – bekommt diesen Request und wandelt ihn in eine BIO. Dabei wird die empfangene Scatter/Gather-List in BIO_VECs transformiert. Mittels `generic_make_request` erzeugt der Blocktreiber aus der BIO einen Request, der gleich in die entsprechende Warteschlange gestellt wird. Dieses Einstellen übernimmt der für diese Warteschlange gewählte I/O-Scheduler. Er faßt diesen Request mit anderen zusammen, wenn das möglich ist. Der IDE-Kern nimmt einen Request nach dem anderen aus der Warteschlange – und zwar wieder mit Hilfe des I/O-Schedulers. Falls diese vorher leer war und der IDE-Kern somit geschlafen hat, wird er vom Blocktreiber geweckt. Der Kern bearbeitet den Request – möglicherweise auch teilweise – und informiert den Scheduler über jeden Teilschritt. Dieser kann in dem Moment entscheiden, auf die komplette Fertigstellung zu warten oder den Client schon über die Beendigung zu informieren. Letzteres ist der Fall, wenn der Request vorher aus mehreren BIOs zusammengesetzt wurde. Die Benachrichtigung des Clients erfolgt mit Hilfe des Notification-Threads von generic_blk. Je nachdem, ob der Client `put_request` oder `do_request` verwendet hat, wird sein Programmablauf jetzt fortgesetzt oder die von ihm angegebene Funktion aufgerufen. Die Abarbeitung des Requests ist hiermit beendet.

Kapitel 4 – Implementierung

4.1 DDE2.6

In Linux 2.6 hat sich auf den ersten Blick sehr viel geändert. Auf den zweiten Blick jedoch stellt sich heraus, daß viele Änderungen nur den Namen einer Funktion nicht aber ihre Aufgabe betreffen. Oftmals sind auch Teile einer Header-Datei in eine andere verlegt worden. Es kommt auch sehr häufig vor, daß sich Änderungen nur auf Makros beziehen, die jetzt anders funktionieren oder nicht mehr als Makro, sondern als Funktion existieren. In all diesen Fällen ist lediglich Recherche- und Umbenennungsarbeit nötig. Das kostet Zeit, ist aber für die Implementation nicht von großer Bedeutung. Anders sieht es aus, wenn Funktionen neu hinzukommen. Hier mußte entschieden werden, ob die neue Funktion wichtig für das Funktionieren verschiedenster Treiber ist. In solch einem Falle mußte sie implementiert werden. Noch schwieriger als das Auftauchen neuer Funktionen ist eine Änderung der benutzten Variablentypen, was ebenfalls oft auftrat. Im Folgenden sind die wesentlichen Änderungen in den einzelnen Bereichen des DDE beschrieben.

PCI

Viele Änderungen hat es bei der Behandlung von PCI-Geräten gegeben. Es war nötig folgende neue Funktionen zu implementieren.

`pci_find_device_reverse`	Diese Funktion entspricht der Funktion `pci_find_device`, welche den PCI-Bus nach Geräten mit bestimmten Eigenschaften – etwa einer Kennung – durchsucht. Der Unterschied ist, daß bei der Suche rückwärts vorgegangen wird.
`pci_bus_{read|write}_config_{byte|word|dword}`	Diese sechs Funktionen entsprechen denen, die bisher schon ohne den Namensteil `bus` existierten. Jetzt kann aber auch der Bus gewählt werden, auf dem gelesen oder geschrieben werden soll. Da im DDE alle Geräte auf einem Bus zusammengefaßt sind, wird das Argument `bus` einfach ignoriert.

Die folgenden Funktionen waren in Linux 2.4 schon vorhanden, aber im DDE2.4 noch nicht implementiert, was nachgeholt wurde.

`pci_request_region`	Diese Funktion fordert eine bestimmte Ressource des angegebenen Gerätes an.
`pci_request_regions`	Diese Funktion führt `pci_request_region` einfach für alle Ressourcen aus.
`pci_release_region`	Einmal angeforderte Ressourcen müssen natürlich auch wieder freigegeben werden. Das geschieht mit dieser Funktion.
`pci_release_regions`	Führt `pci_release_region` für alle Ressourcen aus.

IRQ

Dem IDE-Kern muß bekannt sein, welcher Controller welchen Interrupt verwendet. Sehr oft muß er dazu testen, welcher Interrupt ausgelöst wird, wenn er ein bestimmtes Gerät dazu bringt, den ihm zugeordneten auszulösen. Linux stellt dazu die Funktionen `irq_probe_on` und `irq_probe_off` zur Verfügung. Mit ersterer werden alle Interrupts angefordert und es wird einige Zeit gewartet. Die nach dieser Zeit bereits ausgelösten Interrupts sind höchstwahrscheinlich nicht die gesuchten, denn diese sollten ja eigentlich noch inaktiv sein – sie werden wieder freigegeben. Der Treiber führt jetzt eine Aktion aus, welche mit Sicherheit den gesuchten Interrupt auslöst und ruft danach die Funktion `irq_probe_off` auf. Hat seit `irq_probe_on` genau ein Interrupt ausgelöst, ist dieser der gesuchte und wird dem Treiber mitgeteilt. In allen anderen Fällen ist das Ergebnis nicht eindeutig und der Test schlägt fehl. In jedem Falle werden bei `irq_probe_off` alle Interrupts wieder freigegeben.

Die genannte Funktionalität existierte zwar schon in Linux 2.4, wurde in DDE2.4 aber noch nicht implementiert. Da diese aber so dringend benötigt wurde und wird, wurde das nachgeholt. Dazu sind auch einige weitere Änderungen nötig gewesen, sodaß es das Einfachste war, an dieser Stelle die interne Struktur von Linux zu übernehmen.

Diese hatte auch zur Folge, daß `enable_irq` und `disable_irq` implementiert werden konnten. Außerdem ist damit der Weg für IRQ Sharing geebnet. Dieses wird momentan lediglich abgewiesen, weil es aus Mangel eines passenden Systemes noch nicht getestet werden konnte. Aus Treibersicht steht dem jedoch nichts im Wege.

Mempools

Neu in Linux 2.6 sind Mempools. Diese sind ähnlich den dm_mem-Pools in DROPS. Bei beiden wird Speicher einer bestimmten Größe vorallokiert und steht auch bei Hochlastsituationen verzögerungs- und deadlockfrei zur Verfügung. Der Unterschied besteht darin, daß in DROPS byteweise allokiert wird und in Linux elementweise. Es wird immer versucht, angeforderte Elemente mit Hilfe der angegebenen Allokationsfunktion zu erzeugen. Nur wenn das fehlschlägt, also in Hochlastsituationen, wird der Speicher aus dem Pool genommen.

Folgende Funktionen sind dazu vorhanden:

`mempool_create`	erstellt einen Mempool unter Angabe der Anzahl gesicherter Elemente und der Allokations- und Freigabefunktionen
`mempool_destoy`	zerstört diesen Mempool wieder
`mempool_alloc`	gibt einen Zeiger auf ein neues Element zurück
`mempool_free`	gibt dieses Element wieder frei

KObjects

Bei Gerätetreibern ist es meist der Fall, daß viele hierarchisch geordnete Objekte verwalten müssen. Zum Beispiel alle ihnen angeschlossene Geräte, von denen jedes mehrere Puffer besitzt. Diese Hierarchien werden in der Regel durch verkettete Listen dargestellt. Wenn ein Objekt nicht mehr benötigt wird, muß der Speicher den es belegt freigegeben werden. All diese Funktionen treten immer wieder auf – insbesondere im Blocktreiber mit all seinen Laufwerken, Warteschlangen, Requests und BIOs. Die Entwickler des Linux-Kernel 2.6 haben deshalb die KObjects entwickelt.

Jedes Objekt soll neben seinen eigenen Parametern auch ein KObject enthalten. Dieses KObject enthält nun die Standartelemente wie einen Zugriffszähler und Zeiger zu anderen KObjects. Außerdem enthält es einen Type Descriptor, mit dessen Hilfe sein Typ festgestellt werden kann und in dem sein Destruktor festgelegt ist. Durch Aufruf des Makros `container_of` kann man jederzeit das Objekt erhalten, in dem das angegebene KObject enthalten ist. Die Objekte können so miteinander verkettet werden.

Für eine Hierarchie fehlen aber noch die übergeordneten Objekte. Das sind die KSets. Mehrere KObjects des selben Typs können einem gemeinsamen KSet angehören und enthalten dafür einen Zeiger auf selbiges. Dieses enthält zusätzlich aber ebenfalls ein KObject eines bestimmten Typs und kann somit selbst einem KSet angehören. Damit sind bereits ausladende Baumstrukturen möglich.

Was jetzt fehlt ist eigentlich nur noch ein Wurzelknoten. Dieser wird durch ein Subsystem dargestellt. Ein solches besitzt lediglich ein eingebettetes KSet – wohl aber können mehrere KSets ein und dasselbe Subsystem als das ihrige ausweisen. Die folgende Abbildung verdeutlicht die Zusammenhänge etwas.

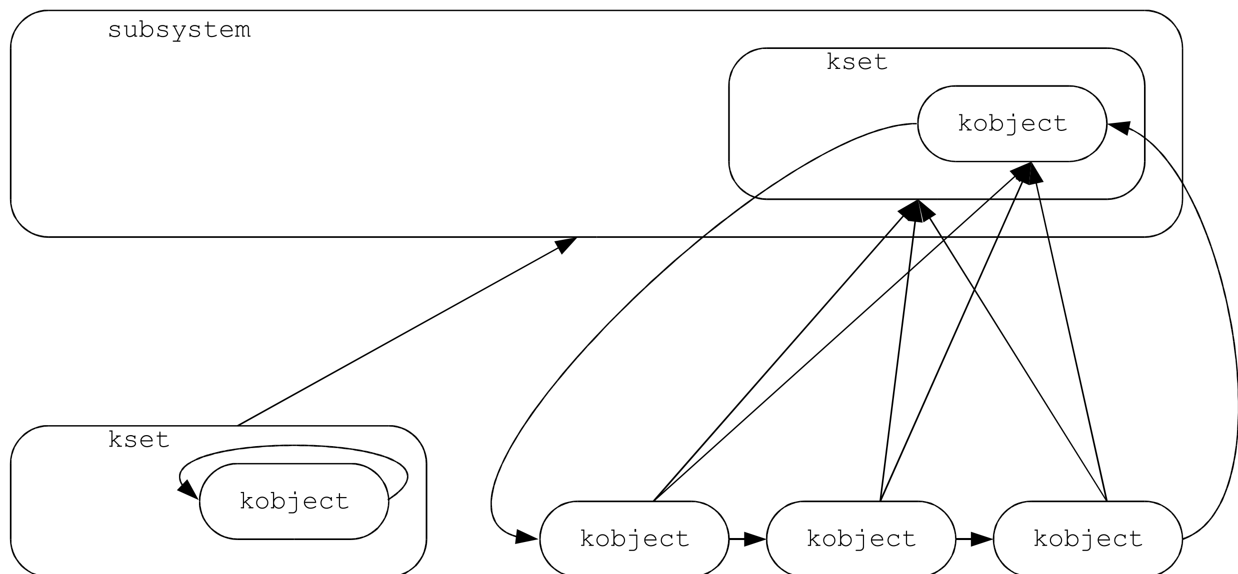

Abbildung 5 - Zusammenhang zwischen KObject, KSet und Subsystem

Die KObjects, KSets und Subsysteme sind zwar ganz nett, stellen aber für sich genommen noch keinen Vorteil gegenüber der herkömmlichen Methode dar. Dieser ergibt sich erst durch die folgenden Funktionen:

`subsystem_register`	erstellt ein Subsystem
`subsystem_unregister`	entfernt sein eigenes KSet
`kset_init`	erstellt ein KSet
`kset_add`	fügt ein KSet seinem übergeordneten KSet oder Subsystem hinzu
`kset_register`	führt `kset_init` und `kset_add` aus
`kset_unregister`	entfernt das KSet aus seinem übergeordneten KSet oder Subsystem
`kset_find_obj`	durchsucht das KSet nach einem KObject mit dem angegebenen Namen und gibt dieses zurück
`kobject_init`	initialisiert ein KObject
`kobject_add`	fügt das KObject dem übergeordneten KSet hinzu
`kobject_register`	führt `kobject_init` und `kobject_add` aus
`kobject_del`	entfernt das KObject aus dem übergeordneten KSet
`kobject_unregister`	führt `kobject_del` und `kobject_put` aus
`kobject_get`	erhöht den Zugriffszähler des KObjects um eins
`kobject_put`	verringert den Zugriffszähler des KObjects um eins
`kobject_cleanup`	gibt alle Ressourcen des KObjects frei
`kobject_set_name`	benennt das KObject

Wird ein KObject registriert, wird es im Linux-Kernel angemeldet. Es ist jetzt über sysfs sicht- und ansprechbar. Außerdem wacht der Kernel über den Zugriffszähler und ruft den Destruktor auf, wenn dieser null erreicht, das Objekt also von niemandem mehr benutzt wird. Der Programmierer muß sich also nicht mehr selbst darum kümmern, was insbesondere durch exzessive Zeigerarithmetik ein hohes Fehlerpotential birgt. Mit der Zeit sollen alle Treiber auf KObjects umgebaut sein. Dann ist es möglich, die komplette Treiberstruktur mit all ihren Objekten per sysfs zu durchsuchen – ähnlich wie es derzeit mit procFS möglich ist.

KObj_Map

Um dem die Arbeit mit neuen Blockgerätetreiber in Linux 2.6 etwas zu erleichtern, wurden für KObjects Mapping-Funktionen erstellt. Diese finden aber mittlerweile auch in anderen Treibern Verwendung, weshalb sie ins DDE2.6 übernommen wurden.

Linux verwendet sogenannte major numbers, welche den Treibern zugeordnet werden, um diese ohne Kenntnis ihres Namens ansprechen zu können. Ein Beispiel wäre ein Treiber für serielle Schnittstellen mit der major number 4. Da es mehrere serielle Schnittstellen in einem System geben kann, sind diese noch in sogenannte minor numbers unterteilt. Ein Treiber weiß üblicherweise nicht im Voraus, mit wie vielen physischen Geräten er zu tun haben wird. Deshalb registriert er vorerst eine bestimmte Anzahl minor numbers für sich. Später muß er dann testen, hinter welchen dieser minors sich tatsächlich Geräte befinden. Dazu benutzt er eine sogenannte Probe Function.

Der Ablauf Registrieren, Probieren und wieder Abmelden ist immer der gleiche – bei allen Treibern die mehrere Geräte unterstützen. Die folgenden neuen Funktionen helfen nun dabei:

kobj_map_init	erstellt eine neue Map und gibt einen Zeiger darauf zurück
kobj_map	fügt der angegebenen Map für den angegebenen Bereich minor numbers und der major number des angegebenen Gerätes eine Lock und eine Probe Function hinzu
kobj_unmap	macht ein Mapping für den angegebenen Bereich rückgängig
kobj_lookup	gibt das KObject zurück, welches von der Probe Function für die angegebene major/minor-Kombination gefunden wurde

Es können innerhalb eines bereits registrierten Bereiches weitere registriert werden, die dann natürliche auch andere Probe Functionen besitzen können. Der kleinste Bereich hat bei einem Aufruf von `kobj_lookup` dann immer die höchste Priorität.

Weitere Änderungen und neue Funktionen

Es gibt folgende weitere Funktionen, die teilweise der Vollständigkeit halber oder wegen Anhängigkeiten im IDE-Treiber zu implementieren waren.

del_timer_sync	Diese Funktion unterscheidet sich in der Implementierungen bisher nicht von `del_timer`. Aus diesem Grunde habe ich sie als Makro auf `del_timer` definiert.
default_wake_function	Diese Funktion ist die Standart-Aufwachfunktion und wird in einigen Strukturen verwendet.
wait_for_completion	Diese Funktion gibt solange CPU-Zeit ab, bis das angegebene Beendigungssignal eintrifft. Dies ist eine Prozeßsynchronisationseinrichtung.
nr_free_pages	Diese Funktion gibt die Anzahl der noch allokierbaren Pages zurück.
ndelay	Diese Funktion verzögert die Bearbeitung um die angegebene Anzahl Nanosekunden.

Die Funktion `nr_free_pages` ist im Moment noch nicht optimal implementiert, denn die Berechnung beruht auf dem noch zur Verfügung stehendem Speicher. Dies ist aber nur eine Näherung an den tatsächlichen Wert der allokierbaren Pages. In einem System ohne vollständige Speicherbelegung bereitet das aber keine Schwierigkeiten.

Ebenfalls problematisch war die Implementation der Funktion `ndelay`. Auf den meisten Architekturen gibt es keine solche Funktionalität und `ndelay` ist dort nur als Marko definiert, welches auf `udelay(1)` abbildet. Dies entspricht einer Verzögerung um eine Mikrosekunde. In DDE2.6 wurde ebenfalls dieses Makro definiert. Es ist auch schwierig, innerhalb einiger Nanosekunden zu einem anderen Prozeß umzuschalten, und dann wieder zurück. Die Alternative wäre eine Schleife, die diese Zeit wartet. Hier ist allerdings eine genaue Kenntnis der Bearbeitungszeit dieser Schleife nötig, was auf den meisten Architekturen nicht gegeben ist. Deshalb ist diese einfache Methode ausreichend.

In Linux 2.6 ist die Initialisierung von Tasks etwas umfangreicher geworden. Deshalb mußte eine Datei `init_task.c` erstellt werden, die es in Linux so nicht gibt. Neue Funktionen sind dadurch aber nicht entstanden.

4.2 IDE-Treiber

Im IDE-Kern waren einige kleine Änderungen nötig. So unterstützt die C-Bibliothek in DROPS noch nicht die Ausgabe von 64 Bit-Werten. Damit aber wichtige Informationen – wie die Größe der Festplatte – überhaupt angezeigt werden können, mußte an den betreffenden Stellen auf 32 Bit-Werte zurückgestellt werden. Das reicht vorläufig noch, sollte in naher Zukunft aber geändert werden. Bei der Erstellung der Scatter-/Gather-List in der Datei `ide-dma.c` wurde der Aufruf der Funktion `pci_map_sg` umgangen, weil er an dieser Stelle nicht mehr nötig ist. Diese Änderungen sind zwar nur minimal, sind hier aber erwähnt, weil sich der IDE-Kern an diesen Stellen eben doch noch vom Linux-Original unterscheidet.

Der Blocktreiber ist in vielen Kernelementen gleich geblieben. So sind die Request-Warteschlangen und deren Verwaltung – inklusive I/O-Scheduler – unverändert. Tagged Command Queueing ist noch nicht implementiert, weil es im Linux 2.6 noch als experimentell deklariert und momentan sogar abgeschaltet ist. Ebenso ist der PIO-Datentransfer für Clients noch nicht möglich – der Treiber selbst benutzt es aber intern. Dazu muß er nämlich Zugriff auf den Puffer des Clients besitzen. Dies ist aber nur mit Dataspace-Zeigern möglich, nicht mit den physischen Adressen die dieser ihm eventuell liefert. Es ist noch ungeklärt, wie reagiert werden soll, wenn ein Client nur eine Scatter-/Gather-List des Typs `L4BLK_SG_PHYS` sendet. Er könnte abgewiesen werden. Das wäre aber meist unnötig, weil heutige Laufwerke fast ausnahmslos DMA benutzen – und dazu reicht die physische Adresse. PIO kommt nur ins Spiel, wenn der Datentransfer aus irgendwelchen Gründen fehlschlägt. Es ist aber nicht sinnvoll möglich, in solch einem Falle den Request als fehlgeschlagen zurückzuweisen. Es müssen also entweder PIO-Transfers generell verboten, oder Requests mit Verwendung des Typs `L4BLK_SG_PHYS` abgewiesen werden.

Der Blocktreiber weist eine große Verstrickung mit Elementen des VFS auf. An erster Stelle seien hier die Inodes genannt. Diese sind Falle des IDE-Treibers aber unnötig und bedeuten einen erheblichen Emulationsaufwand. Sie wurden deshalb entfernt. Außerdem werden im Blocktreiber eigene Versionen der DDE-Funktionen zum Registrieren von Bussen und Geräten benutzt. Eigentlich basieren diese nämlich auf KObjects. Leider war hier ebenfalls keine saubere Trennung zwischen Blocktreiber und restlichem Linux-Kernel möglich, sodaß es wegen fehlender Objekte des Kernels immer wieder zu Problemen kam. Der Blocktreiber ist aber nicht auf diese Funktionalität angewiesen, sondern führt dies eher wegen der dann vorhandenen Sichtbarkeit seiner Geräte im VFS durch. Deshalb war es leicht möglich darauf zu verzichten.

Das Einstellen eines Requests in die Warteschlange eines Gerätes kann unter Umständen zu einem Blockieren des Command-Thread führen. Ein Client sollte beim Aufruf von `put_request` aber über die Annahme oder Verweigerung seines Requests informiert werden. Ein „probier es später noch mal“ ist in generic_blk nicht vorgesehen. Aus diesem Grunde muß der Client also auch blockieren – er kann in der Zeit natürlich auch keine neuen Requests senden. Für andere Clients gilt das allerdings nicht. Sie sollten weiterhin dazu in der Lage sein. Es muß also für jeden Client einen Command-Thread geben. Greift ein zweiter Client also auf eine andere Festplatte zu, so ist dies trotz blockiertem ersten Client möglich. Greift er aber auf genau dieselbe Platte zu wie der erste Client, wird natürlich auch er blockieren.

In generic_blk ist festgelegt, daß für die Bezeichnung von Festplatten die in Linux übliche Methode der major und minor numbers zu verwenden ist. Der Vorteil ist, daß Nutzer nicht von dieser etablierten Variante abweichen müssen. Außerdem benutzt der Bocktreiber diese intern sowieso. Die erste logische Partition der primären Masterplatte – in Linux als „hda5“ bezeichnet – besitzt also auch hier die major/minor-Kombination 3/5 als Kennung. Allerdings ist noch nicht geklärt, wie ein Client herausfindet, welche Laufwerke und Partitionen existieren. Es gibt bisher nur die Möglichkeit, die Anzahl

physischer Platten abzufragen. Wie diese heißen und welche Partitionen sie besitzen, bleibt dem Client verborgen. Er muß es vom Nutzer mitgeteilt bekommen. Hier gibt es also noch gewaltigen Nachholbedarf an generic_blk. Im zukünftigen Block Device Driver Environment sollte das von Anfang an berücksichtigt werden.

4.3 Stolpersteine

Beim Testen stellte sich heraus, daß der IDE-Treiber an eine bestimmte Adresse geladen werden wollte. Der Parameter `reloc` schien ohne Wirkung zu sein. Nach etlichen Stunden Nachforschung war das Problem gefunden. Das Makro `list_for_each` ist in Linux 2.6 optimiert worden. Es führt bei jedem Durchlauf ein Makro `prefetch` aus. Dieses ist zur Optimierung gedacht und architekturabhängig. Der Linux-Kernel ist im Testsystem auf 586 eingestellt. Da aber tatsächlich eine L4-Architektur vorliegt, kam es zum Konflikt. Der Treiber wollte einen bestimmten Speicherbereich eingeblendet haben, der bei L4 aber als normaler Speicher verwendet wird. Das war genau der Bereich, in dem der L4-Kernel liegt – und der ist immer belegt. Die Lösung war dann das Verwenden des Makros `__list_for_each`, welches der alten unoptimierten Version entspricht.

Kopfzerbrechen bereitete auch die Verwendung des Makros `module_init` in vielen Dateien des IDE-Kerns. In DROPS führt dieses dazu, daß die dabei angegebenen Funktionen noch vor dem eigentlichen Anfangspunkt – nämlich der Funktion `main` – ausgeführt werden. Dieses ist aber zum Scheitern verurteilt, weil in dem Moment das DDE noch gar nicht initialisiert ist. Auch die Initialisierung ebenfalls mit `module_init` vornehmen zu lassen war kein Ausweg, weil so nicht bestimmt werden kann, daß genau diese Funktion die erste auszuführende ist. In der entsprechenden dahinterliegenden Funktion `ctor` ist zwar ein Prioritätsparameter vorgesehen, funktioniert aber in der aktuellen Implementation noch nicht. Als Lösung wurde ein eigenes Makro `module_init` definiert, welches die eigentlich nur lokal verfügbaren Initialisierungsfunktionen global macht, indem sie in einer anderen globalen Funktion mit der Vorsilbe `dde_ide_init_` ausgeführt wird. Diese neue Funktion kann nun in der Funktion `main` ausgeführt werden. Und zwar in einer festgelegten Reihenfolge.

Kapitel 5 – Leistungsbewertung

Um die Leistung des mit dieser Arbeit entstandenen IDE-Treibers beurteilen zu können, ist es nötig, die Zeit die dieser für die Bearbeitung eines Requests benötigt mit der zu vergleichen, die der originale Linux-Treiber dazu benötigt. Absolute Werte sind unnütz, weil sie vom konkreten System abhängig, und somit nicht übertragbar sind. Da auch innerhalb eines Systemes relativ große Schwankungen dieser Bearbeitungszeiten auftreten können, ist es weiterhin nötig, sämtliche Unwägbarkeiten – wie Verzögerungszeiten durch das Positionieren des Schreib-/Lesekopfes oder Einflüsse eines Caches – soweit möglich zu eliminieren. Es ist auch von besonderer Bedeutung, daß die Messungen unter möglichst gleichen Bedingungen stattfinden. Deshalb wurde auf das Messen unter Linux im eigentlichen Sinne verzichtet. Hier gibt es Dateisystem (VFS) und Blockcache, die die Werte stark verfälschen würden. Statt dessen wurde derselbe Benchmark der unter L4 lief, direkt in den Linux-Kernel eingebaut. Wichtig war auch, daß die Konfiguration genau der unter L4 entspricht.

Als Grundlage diente der Benchmark aus dem SCSI-Treiber Package. Die nötige Anpassung lag darin, daß zum Start des nächsten Requests eine Semaphore zur Synchronisation benutzt wird. Letzteres war nötig, weil die Request-Beendigungsfunktion im IDE-Treiber im IRQ-Kontext läuft, und der neue Request somit ebenfalls in diesem gestartet würde. Die Warteschlangenverwaltung startet den Request dann sofort. Bei Verwendung von generic_blk ist dies aber nicht der Fall und die Ergebnisse wären nicht vergleichbar. Um aber dennoch ein sofortiges und gleichartiges Starten in allen Situationen zu erzwingen, mußte im IDE-Treiber der Verzögerungsmechanismus unterdrückt werden, der sonst zu große Schwankungen erzeugt hätte.

Zum Vergleichen von Zeitabschnitten ist es nötig, immer dieselbe, möglichst genaue Zeitbasis zu benutzen. DROPS und Linux verwenden aber verschiedene Maße. Von Vorteil erweist sich hier die Möglichkeit, in DROPS den Time Stamp Counter (TSC) des Prozessors auslesen zu können. Unter Linux ist dieses zwar nicht gegeben, aber mit `sched_clock` kann ein in Nanosekunden umgerechneter Wert auf Basis des TSC ausgelesen werden. Diese Umrechnung wurde für die Messung auch auf die von DROPS ermittelten TSC-Werte angewendet und somit eine wirklich einheitliche Zeitbasis geschaffen.

Um Positionierzeiten zu eliminieren, wurde immer wieder derselbe Request durchgeführt. Dessen Größe lag bei 32kB, womit er bei beiden gemessenen Platten komplett im Cache vorlag. Außerdem wurde der jeweils erste Request ignoriert. In jeder Messung wurde 10000 mal gemessen, wie lange es dauert, einen Request zu erzeugen, abzusenden und durchzuführen. Sonstige Verwaltungsaufgaben des Benchmarks gingen nicht mit ein. Es wurden auf zwei Systemen jeweils sechs Messungen durchgeführt. Und diese wiederum jeweils für die Linux-Version, den IDE-Treiber ohne generic_blk und den IDE-Treiber mit generic_blk. Aus jeder Meßreihe wurde dann der Mittelwert errechnet.

Die beiden Testsysteme waren:

	System A	System B
Prozessortyp	Pentium (P54C)	AthlonXP (Thoroughbred)
Taktrate	166 MHz	1533 MHz
Mainboard	Gigabyte GA-586HX	Gigabyte GA-7VTXE
Chipsatz	Intel 430HX (Triton II mit PIIX3)	VIA KT266A
Festplatte	Quantum Bigfoot TX6.0AT (6 GB)	IBM IC35L060AVV207-0 (60 GB)

Und folgende Werte wurden auf ihnen ermittelt:

Linux		IDE ohne generic_blk		IDE mit generic_blk	
System A	System B	System A	System B	System A	System B
24,36821	3,54148	27,57956	3,80816	29,98406	3,96726
24,37538	3,54968	27,52813	3,80971	29,93278	3,97365
24,38851	3,54938	27,52596	3,80232	30,09570	3,96654
24,37713	3,54155	27,52555	3,80233	30,04567	3,96745
24,37347	3,54960	27,53350	3,80815	29,93304	3,97294
24,38327	3,54157	27,54138	3,80267	29,85947	3,96570
24,37766	3,54554	27,53901	3,80556	29,97512	3,96893

Alle Angaben sind in Sekunden per 10000 Requests. Die letzte Zeile ist das jeweilige arithmetische Mittel.

Beim Vergleich der ermittelten Werte von Linux und dem kompletten IDE-Treiber – also mit generic_blk – ergibt sich eine Verzögerung von etwa 5,6 Sekunden in System A und 0,42 Sekunden in System B. Pro Request bedeutet das 560 (System A) und 42 (System B) Mikrosekunden. Im Verhältnis zur Gesamtzeit wären das dramatische 23 (System A) beziehungsweise 12 Prozent (System B). Doch diese Werte sind rein theoretischer Natur und gelten für einen einzelnen Request, dessen Daten sich noch dazu im Plattencache befinden. Dieser Fall ist in der Praxis nicht nur selten, sondern es treten noch ganz andere Effekte in Erscheinung. So verzögert der Blocktreiber einen Request um einige Millisekunden, um ihn mit eventuell folgenden zusammenfassen zu können. Angesichts dieser Zeit sind die 560 Mikrosekunden von System A verschwindend gering – von heutigen schnelleren Systemen ganz zu schweigen. Doch auch in Hochlastsituationen bleibt von den Verzögerungen kaum etwas übrig. Denn während die Daten eines Requests übertragen werden, wird der nächste schon vorbereitet. Insbesondere gilt das für den nicht seltenen Fall, daß der Schreib-/Lesekopf neu positioniert werden muß.

Doch woher kommt die Verzögerung überhaupt? Und wieso ist System B so viel schneller als System A? Die Verzögerung des generic_blk-Interfaces rührt daher, daß hier eine zusätzliche Verarbeitung stattfindet, die außerdem noch über IPCs statt über direkte Funktionsaufrufe laufen muß. Das kostet natürlich einige Zeit. Der Unterschied zwischen Linux und dem IDE-Treiber ohne generic_blk ist dagegen nicht ganz offensichtlich, da hier ja weitestgehend derselbe Quelltext mit denselben Optimierungen benutzt wurde. Aber die Kernelumgebung – das DDE – läuft auf L4. Hier sind für jede Kommunikation IPCs nötig. Außerdem werden Interrupts ebenfalls per IPC zugestellt und es gibt sogar für jeden einen eigenen Thread. Es sind hier also einige Context Switches mehr nötig, als unter Linux. Und das kostet ebenfalls viel Zeit. In [AIP97] wurde schon gezeigt, daß IPCs sehr schnell sind, wenn sich L4 im L1-Cache des Prozessors befindet. Dieser ist beim Pentiums mit 8+8 kB recht mager (der AthlonXP besitzt 64+64+24 kB). Für System A sind die IPCs also „teurer“. Das erklärt, warum es gegenüber System B so schlecht abschneidet.

Letztlich sind die Verzögerungen also prinzipbedingt und haben der Praxis kaum Auswirkungen.

Kapitel 6 – Zusammenfassung und Ausblicke

Aufgabe war es, einen IDE-Treiber für DROPS zu entwickeln. Ich hatte mich auf Grund einer einfacheren Wartbarkeit und besseren Versorgung für die Portierung des IDE-Treibers aus Linux 2.6 entschieden. Dafür mußte das DDE an die neue Linux-Version angepaßt werden. Die Portierung des Treibers aus Linux 2.4 und damit eine Aufwandsminimierung kam nicht in Frage, weil das neue DDE auch für weitere Projekte nutzbar ist und früher oder später sowieso hätte angepaßt werden müssen. Außerdem ist mittelfristig das Ende des 2.4er Kernels absehbar und alle Neuerungen finden demnächst nur noch im 2.6er Kernel Verwendung. Die Neuerungen im IDE-Treiber der 2.6er Version – vor allem die I/O-Scheduler – sind auch für die weitere Entwicklung und Benutzung des Treibers innerhalb von DROPS, besonders innerhalb des kommenden Block Device Driver Framework, nützlich. Für die Anpassung des DDE mußten für die schon bestehende Funktionalität nur wenige Änderungen vorgenommen werden. Allerdings war es nötig, das DDE um neue, mit Linux 2.6 erschienene Funktionalitäten zu erweitern. Diese sind KObjects, KObj_Map und Mempools. Der IDE-Treiber konnte dank des neuen DDE ohne größere Änderungen in DROPS verwendet werden. Er mußte lediglich um die Schnittstelle generic_blk ergänzt werden. Es ist nun möglich, unter DROPS IDE-Laufwerke anzusprechen.

Für die Zukunft ist noch denkbar, das DDE auf die komplette Linux-Kernumgebung zu erweitern. Dann sollte es möglich sein, sämtliche Treiber ohne Änderung in DROPS laufen zu lassen. An einigen Stellen ist dann allerdings ein eingehende Aufwandsabschätzung nötig. So sind Dateisystemtreiber in der nativen DROPS-Umgebung besser aufgehoben als in einer Emulationsumgebung. Es ist dann wenig sinnvoll die Basis für diese Treiber zu emulieren.

Der IDE-Treiber kann zukünftig durch einfaches Kopieren und wenige Handgriffe immer auf dem aktuellsten Stand gehalten werden. Zumindest solange er keine dann neuen Funktionen des Linux-Kernels benutzt. Außerdem ist er durch die gute Trennung von IDE-Kern und Blocktreiber für die Verwendung im Block Device Driver Framework gerüstet.

Insgesamt können DDE und IDE-Treiber noch in Details verbessert werden. Beispielsweise könnte die Unterstützung von Tagged Command Queueing implementiert oder die Verwendung von Shared Interrupts getest und zugeschaltet werden.

Glossar

Deadline	(engl. Todeslinie) der Zeitpunkt, zu dem eine bestimmte Bearbeitung abgeschlossen sein muß
DMA	(Direct Memory Access) bezeichnet den vom Prozessor unabhängigen Datentransfer; dadurch sehr schnell; siehe auch PIO
ID	eindeutige Kennung
IDE	ein Bussystem um Festplatten, optische Laufwerke sowie Disketten- und Bandlaufwerke anzuschließen
IPC	(Inter Process Communication) bezeichnet sowohl die Kommunikation zwischen Prozessen im Allgemeinen als auch das dazu nötige Dienstprimitiv
IRQ	(Interrupt Request) Aufforderung, die aktuelle Bearbeitung zu unterbrechen, um auf ein bestimmtes Ereignis zu reagieren
IRQ-Sharing	bezeichnet die Fähigkeit eines Systems, mehreren Komponenten einen gemeinsamen IRQ zuzuweisen
Makro	ein Platzhalter, der beim Kompilieren durch Funktionen, Werte oder weitere Makros ersetzt wird
Page	(engl. Seite) bezeichnet die kleinste Verwaltungseinheit von Arbeitsspeicher
PCI	(Peripheral Component Interconnect) Standardbussystem für Erweiterungskarten im Rechner
PIO	(Programmed Input Output) der „programmierte" Datentransfer; der Prozessor muß jedes Quantum selbst übertragen; dadurch recht langsam; siehe auch DMA
Plug'n'Play	(engl. Einstecken und Losspielen) bezeichnet die Fähigkeit eines Systems oder einer Komponente dessen, direkt nach Einstecken einsatzbereit zu sein
ProcFS	bezeichnet ein Dateisystem, in dem jeder Prozeß eine Repräsentation enthält; kann zum Auslesen verschiedenster Parameter benutzt werden
Prozeß	eine Verarbeitungseinheit (Programm) mit eigenem Adreßraum
Scheduler	ein Prozeß, der das Scheduling durchführt
Scheduling	Vorgehensweise zur Einplanung von Aufträgen, die durch ein aktives Betriebsmittel zu bearbeiten sind (im Kontext dieser Arbeit: Auftrag/Betriebsmittel = Prozeß/Prozessor und Request/Festplatte)
SCSI	(Small Computer Systems Interface) ein Bussystem um Laufwerke aller Art anzuschließen; im Heimbereich ist eher IDE anzutreffen
sysfs	bezeichnet ein Dateisystem, bei dem jedes Objekt eine Repräsentation enthält; ist ähnlich ProcFS und soll dieses später ersetzen
TCQ	(Tagged Command Queueing) bietet die Möglichkeit, einen Request an ein Laufwerk zu senden, obwohl dieses noch mit der Bearbeitung früherer Requests beschäftigt ist; das Laufwerk kann so eine Optimierung durch Änderung der Request-Reihenfolge vornehmen
Thread	einer von möglicherweise mehreren Aktivitätsträgern innerhalb eines Prozesses
VFS	(Virtual File System) ein virtuelles Dateisystem, welches andere Dateisysteme in sich vereinigt; auch Geräte haben in VFS eine Repräsentation, können also wie Dateien angesprochen werden

Literaturverzeichnis

[DRO] Dresden Realtime Operating System. http://os.inf.tu-dresden.de/drops

[LIN] Linux. http://www.linux.org

[HEL01] Christian Helmuth. Generische Portierung von Linux-Gerätetreibern auf die DROPS-Architektur. Technische Universität Dresden, 2001

[WIN] Windows. http://www.microsoft.com/windows

[FRB] FreeBSD. http://www.freebsd.org

[MAC] MacOS X. http://www.apple.com/macosx

[BEI] Be Incorporated. http://www.beincorporated.com

[OBE] OpenBeOS. http://www.openbeos.de

[LHS98] Modular Operating System. http://modulos.sourceforge.net

[FLX] Flux OSKit. http://www.cs.utah.edu/flux/oskit

[REU01] Lars Reuther. DROPS Block Device Driver Interface Specification. Technische Universität Dresden, 2001

[DIC] DROPS IDL Compiler. http://os.inf.tu-dresden.de/dice

[MEH98] Frank Mehnert. Ein zusagenfähiges SCSI-Subsystem für DROPS. Technische Universität Dresden, 1998

[LHR01] Jork Löser, Hermann Härtig, Lars Reuther. A Streaming Interface for Real-Time Interprocess Communication. Technische Universität Dresden, 2001

[RP03] Lars Reuther, Martin Pohlack. Rotational-Position-Aware Real-Time Disk Scheduling Using a Dynamic Active Subset (DAS). Technische Universität Dresden, 2003

[AIP97] J. Liedtke, K. Elphinstone, S. Schönberg, H. Härtig, G. Heiser, N. Islam, T. Jaeger. Achieved IPC Performance (Still the Foundation of Extensibility). In: 6th Workshop on Hot Topics in Operating Systems (HotOS), 1997