Diplomarbeit

zum Thema

Entwicklung eines Blockgeräte-Frameworks für DROPS

an der

Technischen Universität Dresden
Fakultät Informatik
Institut für Systemarchitektur
Professur Betriebssysteme

Marek Menzer

30. August 2004

Verantwortlicher Hochschullehrer:
Prof. Dr. Hermann Härtig

Betreuer:
Dipl.-Inf. Lars Reuther

Selbständigkeitserklärung:

Hiermit erkläre ich, daß ich diese Arbeit selbständig und nur mit den aufgeführten Hilfsmitteln erstellt habe.

Dürrröhrsdorf, 30. August 2004

Marek Menzer

Inhaltsverzeichnis

Kapitel 1 – Einleitung

Für DROPS, das an der Technischen Universität Dresden entwickelte Dresden Real-Time Operating System, existieren sowohl ein SCSI- als auch ein IDE-Treiber. Beide bieten ihren Nutzern zwar dieselbe Schnittstelle, sind aber dennoch sehr unterschiedlich zu bedienen. Wesentliche Unterschiede sind hier die Benennung der jeweiligen Laufwerke und die verwendeten I/O-Scheduling-Algorithmen. Für einen Dateisystem-Treiber, den wohl häufigsten Nutzer der Blockgeräte-Treiber, ist es schwierig, sich auf diese unterschiedlichen Bedingungen einzustellen. Für Blockgeräte an anderen Bussystemen als IDE und SCSI werden weitere Treiber benötigt. Diese könnten weitere Schnittstellen erforderlich machen, was zu einer hohen Unübersichtlichkeit und einem hohen Anpassungsaufwand seitens der Nutzer führen würde. Ebenso vervielfacht sich der Aufwand, wenn bei Einführung eines neuen Scheduling-Algorithmus jeder dieser Treiber angepaßt werden muß.

Die genannten Mißstände zu beseitigen ist Ziel dieser Arbeit. Dazu soll für DROPS ein Blockgeräte-Framework (kurz: BDDF für Block Device Driver Framework) entworfen und implementiert werden, welches mehrere Bussystem-Treiber über eine gemeinsame Schnittstelle unterstützt und allen Nutzern eine einheitliche Schnittstelle für Blockgeräte-Zugriffe anbietet. Dazu gehört auch, daß die Blockgeräte (Festplatten wie auch CD- und DVD-Laufwerke) der unterschiedlichen Bussysteme und deren Partitionen nach einem einheitlichen Schema benannt sind. Außerdem sollen in diesem Framework verschiedene Scheduling-Algorithmen, insbesondere auch das ebenfalls an der Technischen Universität Dresden entwickelte DAS-Scheduling, verwendbar sein.

Der bereits vorhandene IDE-Treiber ist im Hinblick auf ein solches Framework entwickelt worden. Seine Weiternutzung oder Erweiterung ist deshalb besonders zu untersuchen.

Am Schluß dieser Arbeit soll ein Blockgeräte-Framework stehen, das die genannten Eigenschaften bietet. Es soll für Nutzer transparent sowohl um weitere Bussystem-Treiber als auch neue Scheduling-Algorithmen erweiterbar sein.

1.1 Aufbau dieses Dokumentes

Im folgenden Kapitel werden einige zum Verständnis der Arbeit nötige Grundbegriffe erläutert sowie verwandte Projekte näher betrachtet. Danach werden die einzelnen Bestandteile des Frameworks entworfen. Dabei werden grundsätzliche Überlegungen vorgenommen und wichtige Einzelheiten untersucht. Die auf dieser Basis entstandene Implementation wird in Kapitel 4 anhand ausgewählter Details dargestellt. Das Augenmerk liegt dabei auf einem Überblick und den Entscheidungen, welche zu treffen waren. In Kapitel 5 wird das entwickelte Framework auf seine Leistungsfähigkeit hin untersucht. Darauf folgt eine Zusammenfassung der Arbeit sowie ein Ausblick auf zukünftige Änderungen und Erweiterungen. Zum Abschluß werden in einem Glossar einige verwendete Abkürzungen und Begriffe kurz erklärt sowie weiterführende und zugrundeliegende Literatur aufgelistet.

1.2 Danksagung

Ich bedanke mich bei allen, die mir das Studium und diese Diplomarbeit ermöglicht oder die Zeit währenddessen erleichtert haben. Besonders sind das die Mitarbeiter der Proffesur Betriebssysteme, mein Betreuer Lars, meine Eltern Gerlinde und Bernd, meine Freundin Tanja, Katja, Carsten, Peter, Marko und alle Square Dancer, die ich in dieser Zeit getroffen habe.

Kapitel 2 – Grundlagen und Stand der Technik

In diesem Kapitel sind Begriffe und Verfahren erläutert, deren Kenntnis in den nachfolgenden Kapiteln vorausgesetzt wird. Es bildet damit die Grundlage für das Verständnis der weiteren Arbeit. Dabei werden zuerst die Begriffe *Echtzeit* und *Scheduling* sowie die damit verbundenen Modelle und einige Verfahren verdeutlicht. Anschließend wird auf *DROPS*, das Betriebssystem für welches das Block Device Driver Framework entwickelt werden soll, und seine Besonderheiten eingegangen. Danach wird *Generic_blk* erläutert, welches eine Schnittstelle für Blockgeräte unter DROPS ist. Zum Abschluß werden *L4IDE* – ein bereits unter *DROPS* existierender Blockgeräte-Treiber –, seine Funktionsweise und seine Schnittstellen beschrieben.

2.1 Echtzeit und Scheduling

In einem Computersystem existiert eine Reihe von Ressourcen, wie Speicher und Prozessorzeit. Die Prozesse des Systems stehen in ständiger Konkurrenz um diese Ressourcen, denn von jeder gibt es nur eine begrenzte Menge. Die Verteilung der Ressourcen an die verschiedenen Prozesse wird Scheduling genannt – ein Prozeß, der dieses für eine bestimmte Ressource übernimmt, heißt Scheduler. Er entscheidet bei der Verteilung nach bestimmten Kriterien, wie der Bevorzugung einiger Prozesse, Erzielung einer maximalen Bandbreite oder einer minimalen Verzögerungszeit.

Die meisten Ressourcen lassen sich einem Prozeß wieder entziehen und anderen Prozessen zur Verfügung stellen, was aber einen gewissen Rechenaufwand erfordert und daher nicht immer sinnvoll ist. Zu den nicht wieder entziehbaren Ressourcen gehören Festplattenaufträge. Ist ein solcher einmal abgesendet, wird er vollständig abgearbeitet und kann dabei nicht unterbrochen werden. Ein Scheduler für diese Aufträge heißt I/O-Scheduler. Manche Prozesse erwarten die Bearbeitung eines Auftrages bis zu einem bestimmten, vorher bekannten Zeitpunkt – der Zeitschranke oder Deadline. Systeme, die in der Lage sind, solche Forderungen zu erfüllen, nennt man Echtzeit-Systeme.

Da in dieser Arbeit nur Blockgeräte behandelt werden, ist die hier interessierende Ressource der Festplattenauftrag. Bei einigen Prozessen – meist die von Multimedia-Anwendungen – unterliegen diese Aufträge einer gewissen Regelmäßigkeit. Man spricht dann von einem

Datenstrom. Dieser kann im einfachsten Fall durch die Parameter *Größe eines Auftrages* und *Häufigkeit der Aufträge* charakterisiert werden. Letzteres wird meist in Form einer Periodendauer angegeben, innerhalb welcher die Aufträge im Mittel eintreffen. Fordert der Prozeß, daß jeder Auftrag innerhalb einer bestimmten Periode bearbeitet wird, handelt es sich um einen Echtzeit-Strom. Der Anfang einer neuen Periode markiert dabei die Deadline eines Auftrages. Für eine sichere Planung muß der I/O-Scheduler dabei von der schlechtesten Verarbeitungszeit für diesen Auftrag ausgehen – der Worst-Case-Ausführungszeit. Da diese aber in den seltensten Fällen zur Bearbeitung wirklich erforderlich ist, tritt eine Überreservierung ein, wodurch die mögliche Bandbreite weit verfehlt wird. Da gerade bei Multimedia-Anwendungen ein gewisser Anteil an Daten ausbleiben kann, ohne daß ein Nutzer große Qualitätseinbußen befürchten muß, kann für diese Klasse der optionalen Echtzeit-Aufträge *o* ein zusätzlicher Qualitätsparameter *q* (QoS, Quality of Service) angegeben werden. Dieser gibt dem I/O-Scheduler bekannt, wieviel Prozent der Aufträge im Schnitt nur bearbeitet werden müssen. Für den Scheduler ergibt sich dadurch deutlich mehr Planungsfreiheit, und die erzielbare Bandbreite steigt. Trotzdem gibt es in einem Multimedia-Strom auch Aufträge, deren Bearbeitung unbedingt erforderlich ist. So ist es meist nötig, den Kopf eines neuen Datenblocks zu lesen, auch wenn die Daten selbst fehlen dürfen. Zu dieser Klasse unabdingbarer Echtzeit-Aufträge *m* gehören auch die sogenannten Metadaten-Aufträge. Als Metadaten gelten die Informationen, die zur Verwaltung des Dateisystems notwendig sind. Sie besitzen die höchste Priorität, denn eine große Verzögerung oder gar ein Verlust können sich fatal auf die Datensicherheit auswirken. Die Nicht-Echtzeit-Aufträge werden in der Klasse *nrt* zusammengefaßt.

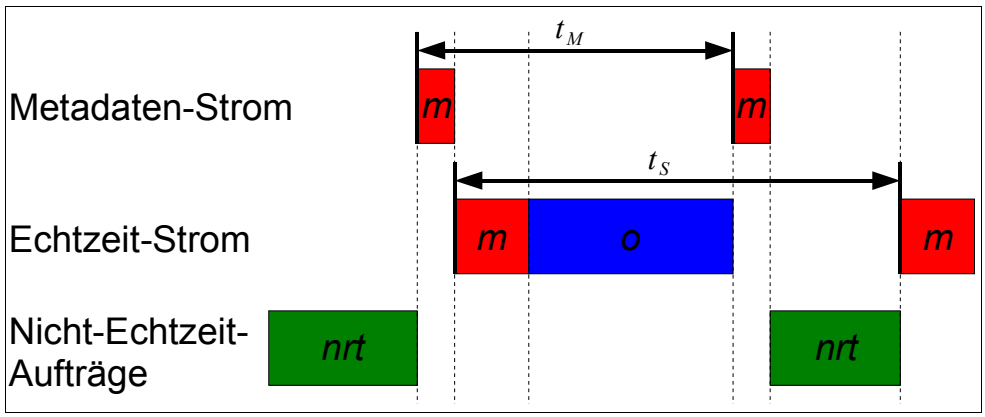

Abbildung 1: *Die verschiedenen Auftragsklassen*

In Abbildung 1 sind die verschiedenen Klassen von Aufträgen gezeigt. Metadaten enthalten nur unabdingbare Aufträge *m*, die mit der Periode t_M eintreffen. Ein Echtzeit-Strom hingegen besitzt auch optionale Aufträge *o* und die Periode t_S. Nicht-Echtzeit-Aufträge *nrt* können immer dann bearbeitet werden, wenn ihre Worst-Case-Ausführungszeit kleiner ist als die Zeit, die bei der Bearbeitung aller Metadaten- und Echtzeit-Ströme übrig bleibt. Die Entscheidung darüber obliegt aber dem konkreten Scheduler.

2.1.1 Quality-Assuring-Scheduling

Mit dem für Quality-Assuring-Scheduling (im weiteren kurz QAS) entwickelten Modell können Echtzeit-Ströme beschrieben werden. Zu denen gehören die Metadaten-Ströme ebenfalls. Weil es ein allgemeines, nicht nur für Festplattenaufträge geeignetes Modell ist, wird ein Strom hier Task genannt. Dabei wird davon ausgegangen, daß eine Task pro Periode t maximal einen unabdingbaren Auftrag m besitzt und dessen Worst-Case-Ausführungszeit w bekannt ist. Optionale Aufträge o darf sie beliebig viele enthalten, ihre Anzahl c muß aber über ihren Existenz-Zeitraum konstant sein. Um dem Modell einen Anhaltspunkt zur Berechnung des Zulassungskriteriums – der Reservierungszeit r – zu geben, ist auch die Angabe der Verteilung der Bearbeitungszeiten erforderlich. Es ergibt sich dadurch für eine Task T_i folgendes Tupel:

$$T_i = \left(X_i, Y_i, c_i, w_i, q_i, t_i\right), \quad i \in \mathbb{N}$$

mit

- X_i Verteilung der Bearbeitungszeiten der unabdingbaren Aufträge,
- Y_i Verteilung der Bearbeitungszeiten der optionalen Aufträge,
- c_i Anzahl der optionalen Aufträge pro Periode,
- w_i Worst-Case-Ausführungszeit des unabdingbaren Auftrages,
- q_i Qualitätsparameter für die optionalen Aufträge,
- t_i Periodendauer.

Da Metadaten-Ströme nur unabdingbare Aufträge enthalten, stellen sie einen Spezialfall dar. Bei QAS sind sie durch $T_M = \left(X_M, 0, 0, w_M, 0, t_M\right)$ beschreibbar.

Die Darstellung der weiteren Berechnungen erfolgt, abweichend vom allgemeinen Modell, mit der Einschränkung, daß die Perioden aller Tasks gleich lang sind. Außerdem beginnen alle Perioden zum gleichen Zeitpunkt. Der Ansatz zur Berechnung des Zulassungskriteriums für den allgemeinen Fall und weitere Details sind in [HLR+01] nachzulesen.

Während des Scheduling-Prozesses werden bei QAS zunächst die unabdingbaren Aufträge aller Tasks zur Ausführung zugelassen. Diese werden zwar mit ihrer Worst-Case-Ausführungszeit eingeplant, die tatsächlichen Ausführungszeiten sind allerdings meist geringer und durch X_i dargestellt. Das heißt, die Ausführungszeit aller unabdingbaren Aufträge zusammen beträgt pro Periode

$$X = \sum_{i=1}^{n} X_i \text{ mit } n = \text{Anzahl aller Tasks im System.}$$

In der verbleibenden Zeit können optionale Aufträge ausgeführt werden. Dazu wird für jede Task T_i die Reservierungszeit r_i so berechnet, daß die gewünschte Qualität q_i erreicht wird. Die Reservierungszeiten aller Tasks ergeben sich aus:

$$\exists r_1, \ldots, r_n \forall i = 1, \ldots, n : r_i = \min(r \mid \boldsymbol{E} A_i \geq q_i c_i)$$

mit

$n =$ Anzahl aller Tasks im System,

$$\boldsymbol{E} A_i = \sum_{k=1}^{c_i} \boldsymbol{P}(A_i \geq k).$$

Für jede Task ist dabei die Verteilung der Anzahl A_i innerhalb einer Periode ausführbarer optionaler Aufträge zu berechnen. In diese Berechnung fließen die möglichen Werte für r_i mit ein.

2.1.2 DAS-Scheduling

DAS steht für Dynamic Active Subset, was schon die Grundidee hinter diesem I/O-Scheduling-Verfahren andeutet – nämlich dynamisch die Teilmenge der Aufträge zu bestimmen, welche als nächste dem Festplattenlaufwerk übermittelt werden können. Ein Festplatten-Scheduler kann sich nun aus dieser Teilmenge den Auftrag auswählen und an das Laufwerk senden, der am besten zu seiner Strategie paßt.

Als Grundlage für DAS-Scheduling dient das QAS-Modell, welches im vorigen Abschnitt schon beschrieben wurde. Dieses Modell wird für das Scheduling von Festplattenaufträgen erweitert. Die Besonderheit hierbei ist, daß sowohl die Worst-Case-Ausführungszeit w als auch die Verteilung der Bearbeitungszeiten der Aufträge Eigenschaften der Platte selbst sind – nicht Eigenschaften der Task beziehungsweise des Stromes. Die Unterscheidung nach unabdingbaren und optionalen Aufträgen ist eine rein organisatorische. Sie hat damit bei Festplatten keine Auswirkung auf die Bearbeitungszeit und braucht deshalb bei deren Verteilung nicht getrennt angegeben werden. Die Periodendauern t aller Tasks sind in QAS als konstant und identisch angenommen worden. Diese können deshalb von keiner Task vorgegeben werden, sondern sind ebenfalls eine Eigenschaft des Systems. Es ergibt sich somit für die Festplatteneigenschaften das Tupel

$$D = (X, w, t) \text{ mit } \begin{array}{ll} X & \text{Verteilung der Auftrags-Bearbeitungszeiten,} \\ w & \text{Worst-Case-Ausführungszeit eines Auftrages,} \\ t & \text{Periodendauer aller Tasks.} \end{array}$$

Als einzige Parameter einer Task verbleiben damit die Anzahl der Aufträge pro Periode c und der Qualitätsparameter q, über den auch die Unterscheidung von optionalen und unabdingbaren Aufträgen erfolgt. Ist q gleich eins, so bedeutet dies eine Qualität von 100% und damit eine Bearbeitung sämtlicher, demnach unabdingbarer Aufträge. Ist q kleiner eins, so handelt es sich um optionale Aufträge. Bei DAS-Scheduling existieren also keine Echtzeit-Ströme mit optionalen und unabdingbaren Aufträgen. Werden in einem System dennoch derartige Ströme verwendet, müssen diese in je einen Strom mit den unabdingbaren Aufträgen und $q=1$ und einen mit den optionalen Aufträgen und dem entsprechenden Qualitätsparameter aufgeteilt werden. Die Beschreibung einer Task T_i reduziert sich also auf das Tupel

$$T_i = \left(c_i, q_i\right),\ i \in \mathbb{N} \text{ mit } \begin{matrix} c_i & \text{Anzahl der Aufträge pro Periode,} \\ q_i & \text{Qualitätsparamer.} \end{matrix}$$

Bisher wurde von einer Bearbeitungszeit unabhängig vom zu übertragenden Datenvolumen ausgegangen. Untersuchungen [Poh02] haben gezeigt, daß dies für SCSI-Platten auch weitestgehend zutrifft, da hier die Übermittlung von Daten oder Aufträgen parallel zu deren Bearbeitung seitens der Platte stattfinden kann. Für ATA-Laufwerke ist dies jedoch nicht der Fall. Soll DAS-Scheduling trotzdem für diese verwendet werden, ist das Modell entsprechend zu erweitern.

In das DAS, die Teilmenge der aktuell bearbeitbaren Aufträge, werden zuerst alle in der momentanen Periode noch ausstehenden unabdingbaren Aufträge aufgenommen. Die noch zur Verfügung stehende Zeit bis zum Periodenende wird um den Faktor aus der Anzahl der aufgenommenen Aufträge und der Worst-Case-Ausführungszeit w der angesprochenen Festplatte reduziert. Ist die verbleibende Zeit größer als w, werden dem DAS auch alle optionalen Aufträge hinzugefügt, deren Task ihre Reservierungszeit noch nicht überschritten hat. Die verbliebene Zeit wird nun um die restliche Reservierungszeit der entsprechenden Tasks weiter reduziert. Ist immer noch eine Restzeit größer w vorhanden, wird das DAS auch um alle Nicht-Echtzeit-Aufträge ergänzt. Optionale Aufträge, deren Task die Reservierungszeit überschritten hat, werden bei diesem Vorgehen ignoriert. Es sind aber Erweiterungen des Algorithmus denkbar, die diese als Nicht-Echtzeit-Aufträge behandeln und ihnen so eine Zweite Chance geben. Damit kann die Qualität einer optionalen Task erhöht werden.

Das DAS wird immer dann neu berechnet, wenn der Festplatten-Scheduler einen neuen Auftrag anfordert. Dies ist meist in dem Moment der Fall, in dem die Bearbeitung des vorherigen Auftrages beendet worden ist. Welche Strategie der verwendete Festplatten-Scheduler implementiert ist ihm überlassen – und dementsprechend auch die Entscheidung,

welchen Auftrag aus dem DAS er auswählt. DAS-Scheduling sorgt dafür, daß trotz dieser Freiheit alle Echtzeit-Aufträge entsprechend ihrer Qualitätszusagen bearbeitet werden.

In [RP03] wurde für den Festplatten-Scheduler das SATF-Verfahren (shortest access time first) implementiert, welches in [JW91] näher untersucht wird. Ihm liegt die Idee zugrunde, daß sich der erzielbare Durchsatz an Aufträgen erhöht, wenn der als nächster bearbeitet wird, dessen betroffene Sektoren am schnellsten von der aktuellen Position des Schreib-/Lese-Kopfes aus erreichbar sind. Dazu ist es aber nötig, eben diese aktuelle Position des Schreib-/Lese-Kopfes sowie die Position aller Sektoren auf dem Festplattenmedium zu kennen. Weiterhin wird ein Modell benötigt, welches die zum Bewegen des Kopfes und zum Lesen der Daten notwendige Zeit ermitteln kann. Ein solches Modell wurde in [Poh02] entwickelt. Darin wird die Bearbeitung eines Auftrages in mehrere Phasen unterteilt:

- die Verzögerung nach dem vorher bearbeiteten Auftrag
- die Verzögerung vor der Bearbeitung des neuen Auftrages
- die Verzögerung durch das Positionieren des Schreib-/Lese-Kopfes
- die Verzögerung durch das Warten auf das Erscheinen des zu bearbeitenden Sektors unter dem Schreib-/Lese-Kopf.

Verzögerungen vor und nach einem Auftrag entstehen, weil das Laufwerk die gelesenen Daten senden oder die zu schreibenden Daten empfangen muß. Die dazu erforderliche Zeit ist theoretisch abhängig vom zu übertragenden Datenvolumen. In der Praxis konnte bei SCSI-Laufwerken jedoch kein Zusammenhang festgestellt werden – bei ATA-Geräten dagegen schon.

Um die nötigen Modellparameter eines Laufwerkes (Sektorenverteilung auf dem Medium, Rotationsgeschwindigkeit, Verzögerung vor und nach einem Auftrag, Kopf-Positionierzeiten) zu ermitteln, werden sogenannte Microbenchmarks eingesetzt. Für einige Messungen und zur ständigen Aktualisierung des Modellzustandes im Betrieb wird eine möglichst genaue Angabe des Bearbeitungs-Endzeitpunktes eines Auftrages benötigt – das Modell kennt dann die genaue Position des Schreib-/Lese-Kopfes. Eine solche Angabe ist seitens der Festplattenprotokolle aber nicht vorgesehen. Deshalb ist das Verfahren darauf angewiesen, den genauen Zeitpunkt des mit dem Laufwerk assoziierten Unterbrechungsereignisses zu erfahren. Diese Angabe ist somit eine Forderung an den Festplatten-Treiber.

DAS-Scheduling ermöglicht die Einhaltung aller Deadlines und Qualitätsforderungen der eingehenden Aufträge. Dabei wird es einem nachgeschalteten Festplatten-Scheduler durch eine Auswahl an bearbeitbaren Aufträgen ermöglicht, einen möglichst hohen Durchsatz zu erzielen. Der dafür in [RP03] verwendete SATF-Scheduler birgt aber das Risiko, daß Aufträge unangemessen lange auf ihre Bearbeitung warten müssen – es tritt die sogenannte Starvation ein. Allerdings sorgt DAS-Scheduling dafür, daß dies nicht bei Echtzeit-Aufträgen

geschehen kann. Für Nicht-Echtzeit-Aufträge ist SATF entsprechend zu modifizieren, wie in [JW91] gezeigt. Einen Nachteil des DAS-Schedulings im praktischen Betrieb stellt die Beschränkung auf eine einheitliche, vom System vorgegebene Periodendauer dar. In der weiteren Entwicklung soll jede Task ihre eigene Periode verwenden dürfen.

2.1.3 Das I/O-Scheduling in Linux 2.6

Der Blocktreiber in Linux enthält drei wählbare I/O-Scheduler, die allesamt nicht echtzeitfähig sind: den NOOP-Scheduler, den Deadline-Scheduler und den Anticipatory-Scheduler. Wobei der NOOP-Scheduler kein eigentlicher Scheduler ist, denn er reicht die Aufträge einfach in der Reihenfolge an das Laufwerk weiter, in der er sie bekommen hat – er implementiert eine FCFS[1]-Strategie. Eine Um- oder Aussortierung erfolgt nicht.

Dem Deadline-Scheduler liegt die Elevator-Idee zugrunde. Dabei wird versucht, die zeitaufwendigen Bewegungen des Schreib-/Lese-Kopfes des Laufwerkes zu minimieren. Das wird erreicht, indem die Aufträge entsprechend der Position ihrer Daten auf dem Datenträger sortiert werden. Werden die Aufträge in dieser Reihenfolge bearbeitet, muß der Schreib-/Lese-Kopf nur eine Bewegung vom kleinsten zum größten Sektor des Mediums vollziehen. Danach findet die Rückbewegung statt. Ein Hin- und Herbewegen zwischen den Sektoren unterbleibt so. Beim Deadline-Scheduler ist ein one-way-Elevator implementiert, was bedeutet, daß die Aufträge nur in aufsteigender Reihenfolge sortiert sind, nicht aber zusätzlich absteigend. Die Kopf-Rückbewegung findet somit ohne Schreib-/Lese-Operationen statt. Der Elevator-Algorithmus birgt die Gefahr in sich, daß Aufträge, die auf Sektoren kleiner der aktuellen Position des Schreib-/Lese-Kopfes zugreifen, sehr lange auf ihre Bearbeitung warten müssen. Deshalb versieht der Deadline-Scheduler alle Aufträge mit einer maximalen Wartezeit (die Deadline; daher der Name Deadline-Scheduler). Ist diese bei einem Auftrag abgelaufen, wird er bearbeitet, obwohl er nach dem Elevator-Algorithmus noch gar nicht an der Reihe ist. Dieses Vorgehen kann zwar keine maximale Verarbeitungszeit garantieren, erreicht dieses Ziel aber in guter Näherung. Damit die Kopfbewegungen außerhalb des Elevator-Zyklus (wegen Aufträgen mit abgelaufener Wartezeit) nicht zu einem dramatischen Einbruch der Übertragungsrate führen, werden die Aufträge innerhalb des Zyklus in Gruppen einer festgelegten Größe abgearbeitet.

Der Anticipatory-Scheduler baut auf dem Deadline-Scheduler auf und erweitert ihn um einige Ideen. So läßt er eine Rückbewegung des Schreib-/Lese-Kopfes zu, falls ihr Weg weniger als die Hälfte des Weges zum nächsten Sektor in Vorwärtsrichtung beträgt. Seinen Namen hat der Scheduler aber vom englischen Verb „to anticipate“, was soviel wie „vorausahnen“ bedeutet. Für diese Vorausahnung erstellt er eine Statistik über die Aufträge, die er von jedem Prozeß erhalten hat. Ist ein Leseauftrag eines Prozesses bearbeitet, und der näch-

1 FCFS steht für **F**irst **C**ome **F**irst **S**erved und bedeutet, daß der Auftrag zuerst bearbeitet wird, der zuerst eintraf.

ste anstehende Leseauftrag wurde entweder vom selben Prozeß generiert oder aber greift auf Sektoren „nahe“ denen des letzten Auftrages zu, so wird dieser sofort ausgeführt. Andernfalls versucht der Scheduler anhand der erstellten Statistik zu ermitteln, ob der nächste Leseauftrag dieses Prozesses innerhalb eines bestimmten Zeitlimits eintrifft und ob Sektoren „nahe“ den Sektoren des letzten Auftrages betroffen sind. Ist dies der Fall, wird die weitere Verarbeitung so lange angehalten, bis der erwartete Auftrag eingetroffen oder das Zeitlimit überschritten ist. Da dieser Mechanismus sehr sensibel auf das eingestellte Zeitlimit und die Art der zugreifenden Anwendungen reagiert, erreicht er bezüglich Durchsatz und Verarbeitungszeit in vielen Fällen ein schlechteres Ergebnis als der Deadline-Scheduler.

2.1.4 Cello

Cello ist kein eigenständiger Scheduler, sondern ein Scheduler-Framework. Dahinter verbirgt sich die Idee, daß es für jeden Anwendungsfall einen optimalen Scheduler gibt. In heutigen Systemen arbeiten viele Anwendungen mit verschiedenen Anforderungen an die Bearbeitung ihrer Aufträge. So sollen die Aufträge einer Video-Anwendung möglichst konstant und innerhalb einer vorgegebenen Zeit abgearbeitet werden (das entspricht einem Echtzeit-Strom), die eines parallel dazu laufenden Texteditors jedoch so schnell wie möglich. Ein FTP-Server aber fordert, daß eine hohe Übertragungrate erzielt wird – ob es dabei zu kurzen Unterbrechungen kommt ist nebensächlich. Statt immer kompliziertere Scheduling-Algorithmen zu konstruieren, die jederzeit gute Ergebnisse liefern sollen, enthält Cello für jedes Aufgabengebiet einen weniger komplexen Scheduler, der für dieses aber sehr gut geeignet ist. Auf Cello entfällt dabei die Aufgabe, die verschiedenen Scheduler untereinander zu koordinieren und für eine einheitliche Schnittstelle zu ihnen zu sorgen.

Cello benutzt dabei zwei Scheduler-Ebenen, eine für die anwendungsspezifischen Scheduler und eine für einen unabhängigen Scheduler. Eingehende Aufträge werden an den jeweils passenden spezifischen Scheduler gegeben, welcher diesen dann seiner Strategie entsprechend in die eigene Warteschlange einsortiert. Der unabhängige Scheduler reserviert für jeden spezifischen Scheduler eine gewisse Bandbreite, anhand derer er entscheidet, wie viele Aufträge jeder in die unabhängige Warteschlange einsortieren darf. Dieses Einsortieren übernehmen die einzelnen Scheduler selbst, beziehungsweise geben an, an welcher Position ein Auftrag einzusortieren ist. Der unabhängige Scheduler prüft daraufhin, ob die rechtzeitige Bearbeitung anderer, bereits einsortierter Aufträge mit diesem Vorgang verletzt würde. Wenn dies der Fall ist, führt er den spezifischen Scheduler einem Second-Chance-Mechanismus zu und setzt die Arbeit beim nächsten Scheduler fort. Damit die spezifischen Scheduler die Einfügeposition ihres nächsten Auftrages überhaupt bestimmen können, stellt der unabhängige Scheduler ihnen Informationen über den Zustand seiner Warteschlange zur Verfügung. Die genaue Berechnung dieses Zustandes ist in [SV98] nachzulesen.

In Simulationen konnte gezeigt werden, daß der durch Cello erzeugte Mehraufwand gering ist. Andererseits haben die verwendeten Algorithmen zur Vergabe der Bandbreite an die spezifischen Scheduler einen erheblichen Einfluß auf die Gesamtleistung. Auch die Reihenfolge, in der die einzelnen Scheduler zum Einsortieren ihrer nächsten Aufträge aufgerufen werden, kann die Optimalität der Auftragssortierung negativ beeinflussen. Das liegt nicht zuletzt auch an dem Konzept, daß die einzelnen Scheduler nicht über das Wissen der jeweils anderen verfügen. Sie liefern so teilweise suboptimale Ergebnisse. Die Bereiche „Vergabe der Bandbreite“ und „Reihenfolge der Scheduler-Aufrufe“ sind Felder für weitere Untersuchungen. Cello wird bereits in QLinux [QLX] (einem zusagenfähigen Linux-Derivat) und Symphony [She98] (einem Dateisystem für Multimedia-Anwendungen) verwendet.

2.2 DROPS

DROPS [DRO], das **D**resden **R**ealtime **OP**erating **S**ystem, ist ein an der TU Dresden entwickeltes Echtzeit-Betriebssystem. Es basiert auf FIASCO, einem ebenfalls an der TU Dresden entwickelten Pendant zum L4-Mikrokern. Im Konzept der Mikrokerne sind keinerlei aktive Programmteile enthalten. Der Kern stellt lediglich Aktivitätsträger (Prozesse) und Kommunikationsmittel (IPCs) zur Verfügung. Außerdem sorgt er für den Schutz der Adreßräume der einzelnen Prozesse, welche dadurch nur über IPCs kommunizieren können. Sämtliche Kernkompetenzen eines monolithischen Kernes, wie Scheduling oder Speichermanagement, sind in Mikrokern-Systemen in einzelne Prozesse außerhalb des Kernes ausgelagert. Meist sind diese Prozesse dann auch im laufenden Betrieb austauschbar. Weiterhin gibt es in DROPS sowohl einen Treiber für SCSI-Laufwerke [Meh98], als auch einen für IDE-Laufwerke (L4IDE), welcher im Abschnitt 2.4 näher beschrieben ist. Es besteht der Wunsch, beide Systeme – und zukünftig auch weitere – zusammen mit verschiedenen Scheduling-Algorithmen in einem Treiber zu vereinen.

In DROPS gibt es einen Namensdienst, denn die Kommunikation über IPCs setzt voraus, daß die Thread-ID des Kommunikationspartners bekannt ist. In der Regel weiß ein Prozeß aber erst zur Laufzeit, welche ID er selbst besitzt. Damit der Partner (üblicherweise ein Client) diese erfährt, muß sich der Prozeß unter einem vorher bekannten systemweit eindeutigen Namen beim Namensdienst registriert haben. Der Partner kann sie jetzt unter Angabe eben dieses Namens erfragen.

Für den Datenaustausch zwischen verschiedenen Prozessen gibt es die Möglichkeit, die Daten mit einem IPC mitzusenden. Bei großen Datenmengen – wie bei Blockgeräten üblich – führt dies aber zu einem erheblichen Kopier- und damit Rechenaufwand. Eine weitere Möglichkeit des Datenaustausches besteht darin, die physische Adresse[2] der Daten im Speicher zu versenden. Das setzt voraus, daß diese bekannt ist. Wurde der Speicher über die

2 Hinter den logischen Adressen verschiedener Prozesse verbergen sich meist auch verschiedene Speicherbereiche, weshalb sie sich nicht zum Datenaustausch eignen.

üblichen Funktionen (z.B. `malloc`) allokiert, ist dies aber nicht der Fall. Außerdem kann der Empfänger-Prozeß nicht direkt, sondern nur über weitere Mechanismen wie DMA auf diese Adressen zugreifen. Das Anwendungsgebiet ist damit auf DMA-fähige Gerätetreiber beschränkt. Weiterhin ist bei Verwendung physischer Adressen die Datensicherheit gefährdet. Denn der Prozeß, der die Adresse versendet hat, kann den Zugriff auf den damit verbundenen Speicher nicht kontrollieren.

Wie bereits erwähnt, wird in DROPS ein L4-Mikrokern verwendet. Unter L4 besteht die Möglichkeit, Speicherseiten über IPC in den Adreßraum eines anderen Prozesses einzublenden, falls dieser der Operation zustimmt. Dabei kann der die Seiten sendende Prozeß entscheiden, ob der Empfänger auf die Seiten schreibend oder nur lesend zugreifen darf. Weiterhin kann er die Seiten entweder gleichzeitig aus seinem eigenen Adreßraum entfernen lassen (*grant*), also auch sämtliche Kontrolle über diese abgeben, oder sie weiterhin selbst eingeblendet lassen (*map*). Durch letzteres ist es möglich, daß sich mehrere Prozesse einen bestimmten Speicherbereich teilen (*shared memory*). Ein Prozeß kann durch *map* in andere Adreßräume eingeblendete Seiten diesen wieder entziehen (*flush*). Er behält also jederzeit die volle Kontrolle über sämtliche Seiten seines Adreßraumes, außer wenn ihm selbst Seiten entzogen werden, welche er von einem anderen Prozeß erhalten hat. Mit den genannten Mechanismen kann ein effizienter Datenaustausch zwischen Prozessen stattfinden. Weitere Einzelheiten dazu sind in [LIE96] nachzulesen.

Um den Umgang mit Speicher praktikabler zu gestalten, existiert unter DROPS das Konzept der *Dataspaces*. Auf ihnen baut die gesamte Speicherverwaltung auf. Ein Prozeß kann einen Dataspace erzeugen (`l4dm_mem_open` oder `l4dm_mem_ds_allocate`) und mit dem enthaltenen Speicher arbeiten wie bisher. Da Dataspaces aber auf den Speicherseiten-Operationen von L4 aufbauen, kann ein Prozeß seine Dataspaces auch

- mit einem anderen Prozeß teilen (`l4dm_share`; in Verbindung mit `l4rm_attach` analog L4-*map*)
- sie an einen anderen Prozeß überschreiben (`l4dm_transfer`; analog L4-*grant*)
- anderen Prozessen eventuelle Rechte an ihnen wieder entziehen (`l4dm_revoke`; analog L4-*flush*).

Weiterhin kann ein Prozeß mit `l4dm_mem_ds_phys_addr` die physische Adresse eines Dataspaces erfragen. Mit den genannten Operationen ist es möglich, daß beide Kommunikationspartner sowohl die physische als auch die virtuelle Adresse eines Speicherbereiches in Erfahrung bringen können. Sie sind somit in der Lage, über DMA und mittels prozessorgestützter Kopieroperationen darauf zuzugreifen. Trotzdem kann ein Prozeß die volle Kontrolle über seine Speicherbereiche behalten.

2.3 Generic_blk

Generic_blk ist eine in [Reu01] definierte Schnittstelle. Sie ist als Standard zum Ansprechen von Blockgerätetreibern in DROPS konzipiert und könnte deshalb auch im zu entwerfenden Blockgeräte-Framework Verwendung finden. Die Blockgerätetreiber werden bei dieser Schnittstelle als Server behandelt, Nutzerprogramme als ihre Clients. Die Kommunikation und der Datentransfer zwischen Server und Client finden dabei über IPCs statt. Dadurch ist es sogar möglich, daß sich ein Server, und damit das entsprechende Blockgerät, auf einem anderen Rechner befindet.

Generic_blk setzt voraus, daß sich der Server beim Namensdienst registriert hat. Ein Client kann diesen unter Angabe dessen Namens öffnen. Er bekommt dabei ein Handle zurück, welches er für alle weiteren Aktionen benötigt. Falls er die Thread-ID des Treibers benötigt, kann er diese ebenfalls erfragen. Nach dem Öffnen kann er Nicht-Echtzeit-Aufträge synchron oder asynchron an den Server senden. Der Unterschied besteht darin, daß der Client bei der synchronen Übertragung bis zur Beendigung des Auftrages blockiert, während er bei der asynchronen Übertragung diesen ohne Warten zum Server sendet. Er muß dann selbst eine Statusabfrage durchführen, wofür ihm generic_blk eine passende Funktion zur Verfügung stellt. Sollte diese einen Fehlerfall anzeigen, kann anschließend die entsprechende Fehlernummer ermittelt werden.

Die Arbeit mit Blockgerätetreibern erfordert meist auch Anfragen, welche nicht dem Datentransport dienen. So muß ein Client beispielsweise Einstellungen am Treiber oder seinen Geräten vornehmen oder abfragen, welche Speicherkapazität ein bestimmtes Gerät besitzt. Werden diese Kontrollanfragen an den Server gesendet, wird das Resultat direkt zurückgeliefert. Für die Übermittlung der Parameter zum und vom Server werden Puffer verwendet, deren Bedeutung von der konkreten Anfrage abhängt. Für die häufigsten Kontrollanfragen existieren bereits separate Funktionen. Diese ermitteln die Anzahl der vom Treiber verwalteten Geräte und die Größe eines bestimmten Gerätes.

Generic_blk unterstützt auch Echtzeit-Aufträge. Für diese muß vorher ein Strom erzeugt worden sein, wobei dessen Rahmenbedingungen mitgeteilt wurden. Diese sind Bandbreite, Blockgröße und der Qualitätsparameter. An den später explizit gestarteten Strom kann der Client die Aufträge mit den bekannten Funktionen zum synchronen und asynchronen Senden an den Treiber übermitteln. Um diese als Echzeit-Aufträge eines bestimmten Stromes zu kennzeichnen, werden sie mit dem entsprechenden Strom-Handle versehen. Ein Strom kann nach Beendigung der Arbeit wieder geschlossen werden.

Die für die Datenaufträge verwendete Struktur wird von einem Client mit den folgenden, für den Server wesentlichen Informationen versehen:

- ob es ein Schreib- oder ein Leseauftrag ist
- ob es ein Metadaten-Auftrag ist
- für welches Gerät der Auftrag bestimmt ist
- die Nummer des Startblocks auf dem Gerät, ab dem der Datentransfer stattfinden soll
- die Anzahl der zu übertragenden Blöcke
- zu welchem Strom der Auftrag gehört, falls es ein Echtzeit-Auftrag ist
- die Nummer des Echtzeit-Auftrages
- den Zeiger auf den Datenpuffer in Form einer Scatter/Gather-Liste
- die Anzahl der Elemente der Scatter/Gather-Liste
- ob die Elemente der Scatter/Gather-Liste physische Adressen oder Dataspace-Zeiger enthalten

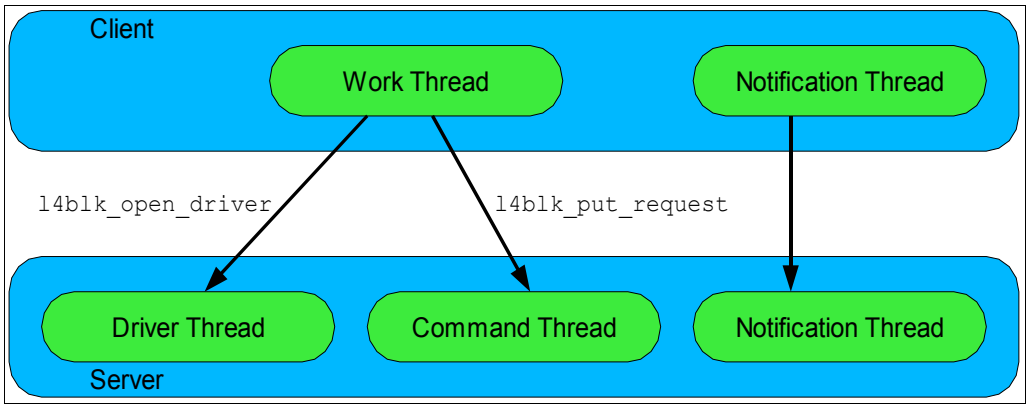

Abbildung 2: *Die Threads von Client und Server in generic_blk*

Generic_blk wurde mit dem Werkzeug DICE [DIC] erstellt. Dieses unterstützt aber keine asynchronen Schnittstellen. Deshalb müssen zur Kommunikation mehrere Threads eingesetzt werden. Diese sind in Abbildung 2 dargestellt. Ein Server enthält dabei zumindest den Driver-Thread, welcher beim Namensdienst registriert ist. Dessen ID kann so vom Client beim Öffnen des Servers in Erfahrung gebracht werden. Zusammen mit dieser Information erhält der Client auch die IDs des Command- und des Notification-Thread, welche nicht notwendigerweise verschieden sein müssen. Sie müssen nicht einmal anders lauten als die ID des Driver-Thread. Diese Entscheidung obliegt dem Server, der dadurch beispielsweise für jeden Client eigene Command- und Notification-Threads zur Verfügung stellen kann. Der Command-Thread nimmt Aufträge vom Client entgegen, welche dieser über seinen Work-Thread abgeschickt hat. Beim Notification-Thread können sich Clients Benachrichtigungen über deren Bearbeitung oder einen Fehlerfall abholen. Generic_blk erledigt dies automatisch im Hintergrund, so daß die entsprechenden Abfragefunktionen keine IPCs auslösen.

2.4 L4IDE

L4IDE [Men03] ist ein Blockgerätetreiber für den IDE-Bus und wurde an der TU Dresden für DROPS entwickelt. Dafür wurden sowohl der IDE-Treiber als auch der Blocktreiber des Linux-Kernels der Version 2.6 mittels der Emulationsumgebung DDE2.6 (für **D**evice Driver **D**evelopement **E**nvironment der Linux-Version **2.6**) nach DROPS portiert. Der Blocktreiber übernimmt dabei die allgemeinen Aufgaben wie Warteschlangen- und Partitionsverwaltung sowie das Scheduling. Der IDE-Treiber, im L4IDE-Projekt IDE-Kern genannt, bietet die Umsetzung der allgemein formulierten Plattenaufträge des Blocktreibers in Plattenaufträge nach dem IDE-Protokoll und gibt diese an die Hardware weiter. Der Blocktreiber kann auch Treiber für weitere Bussysteme, wie beispielsweise SCSI, zur Seite gestellt bekommen. In L4IDE ist aber nur der IDE-Treiber vorhanden. Als Schnittstelle zum Client dient generic_blk, welches in Abschnitt 2.3 schon näher betrachtet wurde. Allerdings unterstützt L4IDE dabei keine Echtzeitanfragen und bietet somit auch keine Echtzeit-Ströme. Eventuell kann L4IDE aber dennoch als Basis für das zu entwerfende Blockgeräte-Framework dienen oder in Teilen weiterverwendet werden.

2.4.1 Der IDE-Treiber

Der IDE-Treiber selbst ist keine aktive Komponente, beinhaltet also keine Prozesse oder Threads. Er ist lediglich eine Bibliothek für den Blocktreiber. Im Sinne einer guten Wartbarkeit wurde bei der Portierung nur ein geringer Teil seines Quellcodes verändert. Er ist dadurch einfach durch die neueste Version aus dem Linux-Kernel zu ersetzen.

Nach dem IDE-Protokoll sind die verschiedenen Laufwerksarten auch unterschiedlich anzusprechen. Optische Laufwerke (CD und DVD) unterscheiden sich hier grundlegend von Band-[3] oder Festplattenlaufwerken. Um trotzdem die vom Blocktreiber geforderte Einheitlichkeit zu erreichen, ist es Aufgabe des IDE-Treibers, von den Laufwerksarten zu abstrahieren. Dazu gehört natürlich auch, daß die durch die Geräte auftretenden Unterbrechungsanforderungen komplett im Geltungsbereich des IDE-Treibers behandelt werden. Gleiches gilt für die Abwicklung von Datentransfers via DMA.

Es gehört ebenfalls zu den Aufgaben des IDE-Treibers, während der Initialisierungsphase nach vorhandenen IDE-Chipsätzen zu suchen. Jeder der gefundenen Chipsätze wird von ihm initialisiert und nach angeschlossenen Geräten durchsucht. Jedes gefundene Gerät wiederum wird dem Blocktreiber gemeldet, welcher dafür die entsprechenden Verwaltungsstrukturen anlegt. Um die Chipsätze ansprechen zu können, muß der IDE-Treiber auf den PCI-Bus zugreifen. Dafür greift er auf die Funktionen des DDE2.6 zurück.

3 Da Bandlaufwerke keine Blockgeräte sind, werden sie natürlich nicht dem Blocktreiber gemeldet, sondern der für Zeichengeräte zuständigen Stelle im Linux-Kernel. Da sie aber IDE-Geräte sind, sind sie im IDE-Treiber dennoch enthalten.

2.4.2 Der Blocktreiber

Das zentrale Element des Datentransfers in L4IDE ist die BIO (Block I/O-Operation). Sie enthält Informationen über Quell- und Zielort der zu übertragenden Daten, das angesprochene Gerät und das Übertragungsvolumen. Der Ort der Daten auf dem Gerät ist durch die Nummer des Startsektors beschrieben, der Ort im Arbeitsspeicher durch eine Scatter/Gather-Liste. Im Linux-Treiber umfaßt ein Elemente der Scatter/Gather-Liste – der BIO_VEC – die Information über Page, Offset und Größe des entsprechenden Speicherbereiches, in L4IDE dagegen direkt dessen physische Adresse und seine Größe. Ein Index zeigt auf das aktuell zu benutzende Element. Weiterhin ist in der BIO vermerkt, welche Funktion bei ihrer vollständigen Bearbeitung oder im Fehlerfall aufgerufen werden soll.

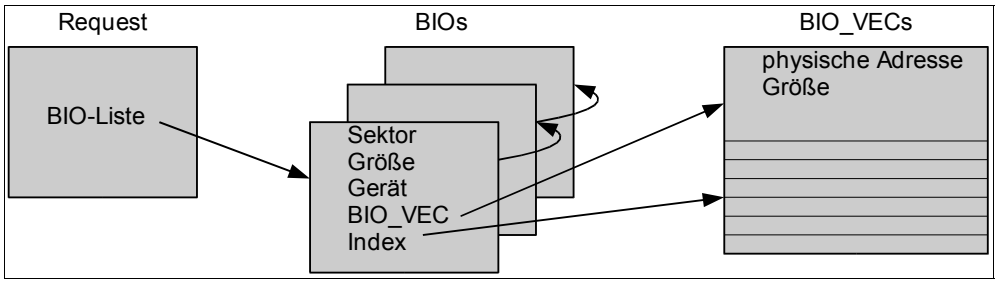

Abbildung 3: *Der Zusammenhang zwischen Request, BIO und BIO_VEC*

BIOs werden nach Eintreffen der Aufträge eines Clients erzeugt und durch den Blocktreiber zum IDE-Treiber gereicht, wo sie bearbeitet werden. Der Blocktreiber kann aber auch mehrere BIOs auf einmal bearbeiten. Deswegen sind sie in einen Request eingebettet. Dessen Zusammenhang mit den BIOs und deren BIO_VECs ist in Abbildung 3 dargestellt. Der Blocktreiber übernimmt auch die Aufgabe der Partitionsverwaltung. Dabei werden die BIOs direkt nach ihrem Eintreffen zum Anfang des physischen Laufwerkes hin ausgerichtet. I/O-Scheduler und IDE-Treiber arbeiten damit auf Basis des gesamten Mediums.

Daß ein Request mehrere BIOs enthalten kann hat den Vorteil, daß zwei Requests ohne viel Rechenaufwand zu einem zusammengefaßt werden können. Dazu werden die BIOs des einen dem anderen hinzugefügt. Der zweite Request wird nun dem treiberweiten Request-Pool rückgeführt und steht für weitere Aufträge zur Verfügung. Wegen der Lokalität von Daten[4] kommt es häufig vor, daß auf benachbarte Sektoren eines Laufwerkes zugegriffen werden soll. Um diese sequentiellen Zugriffe zu erkennen und so ein Zusammenfassen zu ermöglichen, hält der Blocktreiber die Requests eine gewisse Zeit – je nach System drei bis zehn Millisekunden – zurück. Dies geschieht aber nur, solange dies maximal vier von ihnen betrifft. Somit wird in Hochlastsituationen, in denen mehr als vier Requests gleichzeitig auf ihre Bearbeitung warten, eine zusätzliche Verzögerung vermieden. Unter günstigen Be-

4 Prozesse weisen eine hohe Lokalität auf. Sowohl der aktuell ausgeführte Programmcode als auch die gerade von ihm verwendeten Daten befinden sich in einem bestimmten Bereich um die bereits vorher angesprochenen Speicherbereiche herum. Gerade bei Multimedia-Anwendungen ist dies besonders deutlich, da hier verhältnismäßig große Datenmengen sequentiell – also mit zusammenhängenden Speicherbereichen – übertragen werden.

dingungen kann die Last so reduziert werden, daß der Kommando-Overhead eines oder sogar mehrerer Requests gespart wird. Außerdem kann damit die Zeit entfallen, die nötig wäre, wenn die Festplatte auf das erneute Auftauchen des betreffenden Sektors unter dem Schreib-/Lese-Kopf warten müßte. Diese liegt im Bereich mehrerer Millisekunden.

Damit die Requests überhaupt zurückgehalten werden können, werden sie in eine Warteschlange, die Request-Queue, eingeordnet. Eine solche existiert für jedes physische Laufwerk. Wann immer ein Laufwerk bereit für weitere Aufgaben ist, holt der IDE-Treiber den nächsten Request aus der Warteschlange und gibt ihn an dieses weiter. Die Warteschlangenverwaltung wird dabei von einem I/O-Scheduler übernommen. Er allein führt jegliche Operationen an der Warteschlange durch. Die wesentlichen dieser Operationen sind das Einstellen, das Umsortieren und das Herausgeben von Requests. Er bestimmt auch, ob zwei Requests zusammengefaßt werden dürfen. Der I/O-Scheduler ist somit in der Lage, bestimmte Requests unter einer Zielsetzung zurückzuhalten oder zu bevorzugen. So kann er in einem Echtzeitsystem dafür sorgen, daß alle Requests ihre Deadlines einhalten oder daß eine möglichst hohe Bandbreite erzielt wird. In L4IDE kommt der NOOP-Scheduler zum Einsatz, der die Warteschlange als FIFO-Puffer verwaltet. Für jede Warteschlange, und damit für jedes physische Laufwerk, ist ein eigener I/O-Scheduler wählbar.

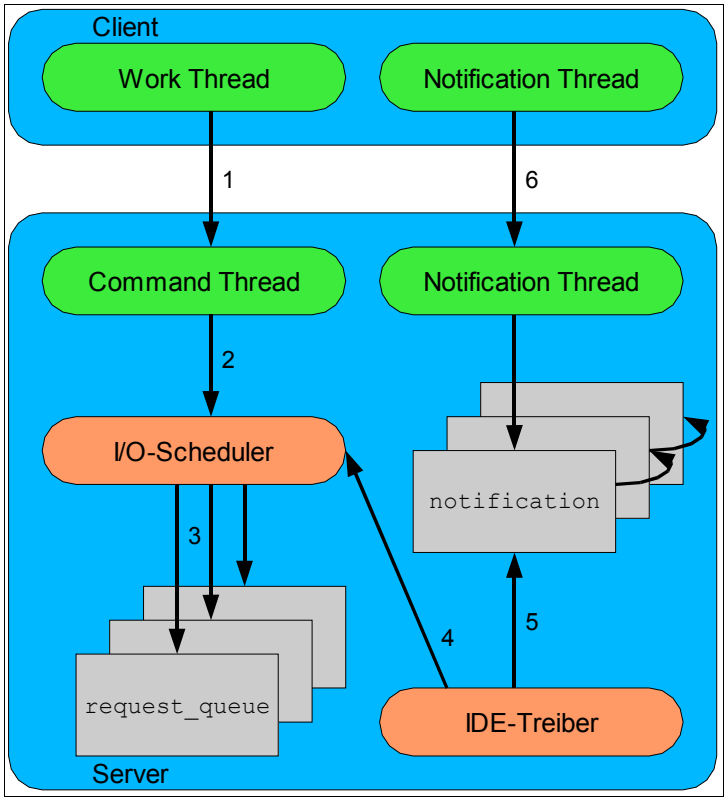

Abbildung 4: *Der Weg einer BIO durch L4IDE*

1 Der Client sendet einen Auftrag an den Server.

2 Aus den Daten des Auftrages wird eine BIO erstellt und an den I/O-Scheduler des entsprechenden Gerätes weitergegeben.

3 Der I/O-Scheduler ordnet die BIO in die zugehörige Warteschlange ein.

4 Der IDE-Treiber holt sich die BIO beim I/O-Scheduler ab.

5 Nach Bearbeitung der BIO oder im Fehlerfall erzeugt der IDE-Treiber eine Benachrichtigung, die über dieses Ereignis informiert.

6 Der Client holt sich die Benachrichtigung vom Server ab.

Abbildung 4 zeigt den Weg einer BIO – auf Teilen des Weges in einem Request verpackt – durch L4IDE. Die einzelnen Stationen sind in der Reihenfolge ihres Auftretens nummeriert und auf der rechten Seite kurz beschrieben.

Wie am Beginn dieses Abschnittes bereits erwähnt, verwendet L4IDE als Schnittstelle zu einem Client generic_blk. Von besonderem Interesse für diese Arbeit sind aber auch die Schnittstellen zu den Komponenten I/O-Scheduler und IDE-Treiber.

2.4.3 Die Schnittstelle zwischen Blocktreiber und I/O-Scheduler

Jeder Warteschlange kann einen separater I/O-Scheduler zugeordnet sein, welcher sich nicht nur im Algorithmus, sondern auch in der Anzahl und den Namen seiner Funktionen von anderen Schedulern unterscheidet. Dem Blocktreiber soll aber eine einheitliche Sicht auf die verschiedenen Scheduler ermöglicht werden. Diese Aufgabe erfüllt der Elevator[5]. Er verteilt die allgemeinen Anfragen des Blocktreibers auf die, soweit vorhanden, speziellen Implementationen des zur Warteschlange gehörenden Schedulers.

Für die Funktionalität eines I/O-Schedulers sind im Elevator mehrere Funktionen vorgesehen, die implementiert werden können. Es gibt aber auch zwei grundlegende Funktionen, welche in einem Scheduler implementiert werden müssen. Das ist zum einen eine Funktion, durch welche der Blocktreiber neue Requests an den Scheduler übergibt. Dabei kann er auch festlegen, ob der Request am Anfang oder am Ende der Warteschlange eingefügt werden soll. Zum anderen ist das eine Funktion, durch welche der Blocktreiber – genauer, der IDE-Treiber – den nächsten zu bearbeitenden Request vom Scheduler anfordert. Der Elevator geht bei einer solch minimalen Implementation davon aus, daß die Warteschlange als eine zweifach verkettete Liste von Requests vorliegt. Ist dies nicht der Fall, weil die Warteschlange beispielsweise als Baum konzipiert wurde, sind weitere Funktionen zu implementieren. Diese geben den Füllstand der Warteschlange und die Requests vor und nach einem angegebenen Request an.

Sind alle BIOs eines Requests vom IDE-Treiber abgearbeitet oder durch Zusammenfassen in einen anderen Request überführt worden, so ist dieser ohne weitere Funktion und kann aus dem System entfernt werden. Soll der Scheduler über dieses Ereignis informiert werden, muß die entsprechende Funktion implementiert sein. Block- oder IDE-Treiber dürfen einen Request auch komplett aus der Warteschlange herausnehmen, um ihn exklusiv zu benutzen. Dabei dürfen sie diesen aber nicht verändern. Später kann er wieder in die Warteschlange eingesetzt werden.

5 In früheren Kernel-Versionen wurde ein Elevator-Algorithmus implementiert. In der hier verwendeten Version ist der Algorithmus dem I/O-Scheduler überlassen. Der Name Elevator hat sich für den Funktionsverteiler aber erhalten.

Sendet ein Client dem Blocktreiber eine neue BIO, so fragt dieser den Scheduler, ob diese BIO mit einer BIO eines bereits in der Warteschlange befindlichen Requests zusammengefaßt werden kann. Der Scheduler kann dies verneinen oder den passenden Request zurückliefern. Außerdem teil er dem Blocktreiber mit, ob die BIO am Anfang oder am Ende des Requests eingefügt werden soll. Hat der Blocktreiber die Einfügeoperation ausgeführt, informiert er den Scheduler darüber. Der kann nun entsprechend seiner Scheduling-Strategie darauf reagieren. Bei erfolgreicher Einfügeoperation versucht der Blocktreiber außerdem, den Request, in den die BIO eingefügt wurde, mit dem jeweils nächsten (falls am Ende eingefügt wurde) oder vorhergehenden (falls am Anfang eingefügt wurde) zusammenzufassen. Gelingt auch dies, wird der Scheduler wiederum informiert und kann den nun leeren Request aus der Warteschlange entfernen und den nun noch mehr BIOs enthaltenden Request in der Warteschlange neu einordnen, falls die Scheduling-Strategie dies erfordert.

2.4.4 Die Schnittstelle zwischen Block- und IDE-Treiber

Die Schnittstelle zwischen dem Block- und dem IDE-Treiber hat keine klare Grenze. So gibt es beispielsweise Funktionen, die zu beiden Treiberschichten gehören. Das hat vor allem zwei Gründe. Zum einen greifen beide Treiber auf dieselben Strukturen zu und verändern diese. Zum anderen ist der Blocktreiber hauptsächlich aus einem ehemals zusammengehörigen IDE-Block-Treiber entstanden, weshalb beide noch sehr miteinander verwoben sind. Im folgenden wird trotzdem der Versuch unternommen, eine Grenze zu ziehen, um so eine Schnittstelle erkennbar werden zu lassen.

Der IDE-Treiber reserviert sich eine oder mehrere Major Numbers[6] beim Blocktreiber. Damit kann er eindeutig angesprochen werden, und es ist ausgeschlossen, daß mehrere Gerätetreiber unter derselben Nummer erreichbar sind. Unter einer Major Number kann es aber auch verschiedene Abwandlungen eines Treibers geben. Beispielsweise kann an einem IDE-Kanal sowohl eine Festplatte als auch ein DVD-Laufwerk angeschlossen sein, wofür jeweils unterschiedliche Treiberteile zuständig sind. Deshalb registriert der IDE-Treiber verschiedene Bereiche an Minor Numbers. Dies dient dem Blocktreiber später dazu, zu jedem Gerät den passenden Treiberteil anzusprechen.

Der IDE-Treiber meldet dem Blocktreiber jedes angeschlossene Laufwerk und dessen Eckdaten. Der liest daraufhin – mit Hilfe des IDE-Treibers – die Partitionierungsdaten aus und legt die logischen Laufwerke an. Nun führt der IDE-Treiber eine Funktion aus, durch die für das gemeldete Laufwerk eine neue Warteschlange erzeugt und mit Standardwerten initialisiert wird. Der IDE-Treiber hat dabei auch die Funktion angegeben, welche der Blocktreiber

6 In Linux ist jeder Gerätetreiber unter einer systemweit einmaligen Nummer registriert. Diese Major Number ermöglicht erst das Grundkonzept, daß jedes Gerät als Datei ansprechbar ist. Um die verschiedenen Geräte, für die ein Treiber zuständig ist, getrennt ansprechen zu können, besitzt jedes Gerät zusätzlich eine Minor Number.

aufrufen soll, sobald die Warteschlange Requests enthält. Der dafür gewählte I/O-Scheduler wird dabei ebenfalls initialisiert.

Der IDE-Treiber hat die Möglichkeit, die Warteschlange nach den Bedürfnissen der angeschlossenen Geräte zu konfigurieren. So kann er beispielsweise angeben, welche Sektorgröße das Gerät verwendet oder wie viele Sektoren ein Request maximal umfassen darf. Sobald der IDE-Treiber die Warteschlange für die Bearbeitung von Requests freigibt, wird die vorher dafür angegebene Funktion aufgerufen. Die Bearbeitungsfunktion erfragt sich den nächsten Request und behandelt diesen. Der Blocktreiber wird dabei sowohl über die Teilschritte als auch über die vollständige Bearbeitung informiert.

Soll die Arbeit des IDE-Treibers beendet werden, so hat dieser sämtliche vorher reservierten Ressourcen freizugeben. Dazu muß er zuerst verhindern, daß weitere Requests in die Warteschlange eingestellt werden. Die noch vorhandenen Requests müssen vorher bearbeitet werden. Anschließend wird die Warteschlange zerstört. Der IDE-Treiber erklärt das zugehörige Gerät für ungültig und gibt den dafür reservierten Speicher frei. Abschließend gibt er die anfangs für sich reservierte Major Number wieder an den Blocktreiber ab.

2.4.5 Der Status des L4IDE-Projektes

In der derzeitigen L4IDE-Version kann ein Nutzer auf IDE-Festplattenlaufwerke und eingeschränkt auch auf CD/DVD-Laufwerke zugreifen. Ein Nutzerprogramm benutzt für die Zugriffe die generic_blk-Bibliothek – muß sich also nicht um IDE-spezifische Dinge kümmern. L4IDE basiert momentan auf der Linux-Kernel-Version 2.6.6. Da das Tagged Command Queueing in dieser Version noch als experimentell deklariert und damit nur für Testzwecke geeignet ist, wurde es auch in L4IDE nicht aktiviert. Die für eine spätere Nutzung nötigen Vorkehrungen sind jedoch bereits getroffen.

Kapitel 3 – Entwurf

Das zu entwerfende Block Device Driver Framework (*BDDF*) soll eine einheitliche Sicht auf die Blockgeräte verschiedener Bussysteme ermöglichen. Da für jedes Bussystem ein eigener Treiber nötig ist, sollen diese über eine gemeinsame Schnittstelle in das System integrierbar sein. Außerdem ist ein Namensschema erforderlich, über welches die einzelnen Geräte benannt werden. Die Aufträge, welche von einem Nutzerprogramm – dem Client – an ein bestimmtes Blockgerät übermittelt werden, müssen durch einen I/O-Scheduler unter bestimmten Gesichtspunkten sortierbar sein. Zu diesen Aufträgen gehören sowohl Echtzeit- als auch Nicht-Echtzeit-Aufträge. Das heißt, der Client muß angeben können, daß einige Aufträge innerhalb einer bestimmten Zeit abgearbeitet sein müssen. Weiterhin sollen verschiedene I/O-Scheduler gleichzeitig benutzbar sein, die dann eine entsprechende Scheduling-Strategie verfolgen. Für diese Funktionalität ist ebenfalls eine möglichst universelle Schnittstelle nötig. Eine Blockgeräte-Verwaltung soll für die Koordination der einzelnen Bereiche untereinander und die Buchhaltung über vorhandene Geräte sorgen. Zusätzlich soll es einem Client möglich sein, Informationen über einzelne Geräte, Scheduler oder die Blockgeräte-Verwaltung zu erfragen sowie Einstellungen (beispielsweise des DMA-Modus) an ihnen vorzunehmen.

In diesem Kapitel werden also die folgenden Komponenten des BDDF entworfen:

- Namensraum
- Schnittstelle zum Client
- Schnittstelle zum I/O-Scheduler
- Schnittstelle zum Bustreiber.

Zur besseren Übersicht sind die genannten Komponenten und deren Zusammenhänge in Abbildung 5 visualisiert. Dabei wird das gesamte Block Device Driver Framework mit möglichen Nutzern (Clients), Bustreibern, I/O-Schedulern und den jeweiligen Schnittstellen gezeigt. Eine konkrete Implementierung einer Blockgeräte-Verwaltung zusammen mit den Schnittstellen wird im folgenden *BDDF-Instanz* genannt.

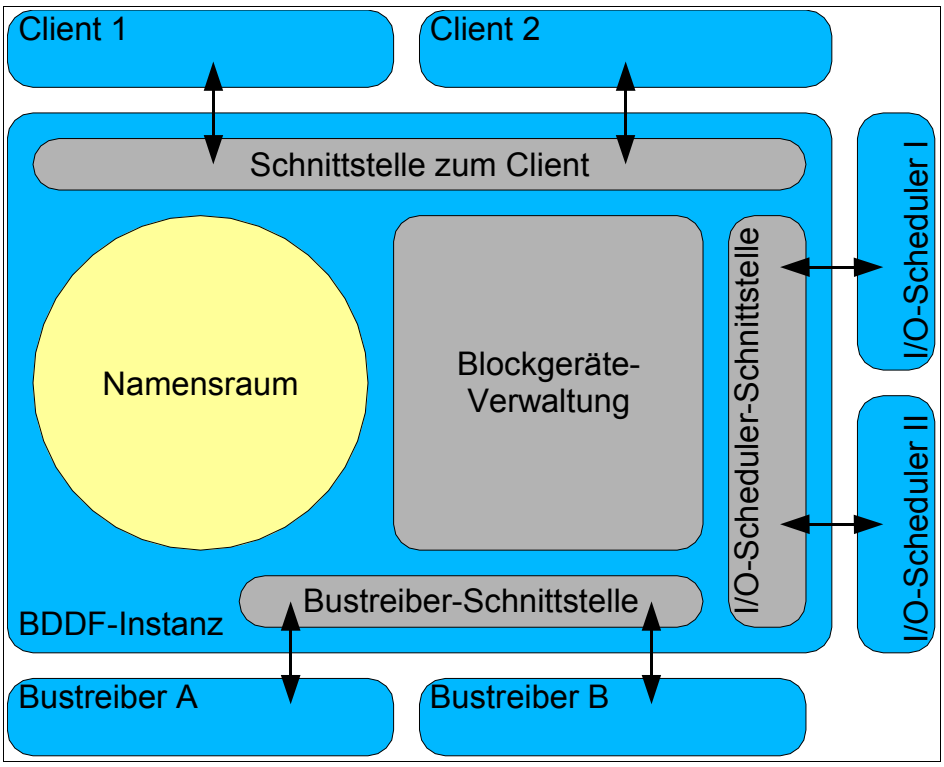

Abbildung 5: *Aufbau des Block Device Driver Framework*

Im weiteren folgt ein Abschnitt über einige Grundüberlegungen und -definitionen für den Entwurf des Block Device Driver Frameworks. Danach wird der Namensraum erarbeitet. Darauf schließen sich die Abschnitte zu den einzelnen Schnittstellen in der Reihenfolge Client, I/O-Scheduler und Bustreiber an.

3.1 Vorüberlegungen

Vor der näheren Betrachtung der einzelnen zu entwerfenden Komponenten des BDDF werden in diesem Abschnitt einige grundsätzliche Festlegungen getroffen.

Um Protokollaufwand im und zum Bustreiber zu reduzieren, sollen Aufträge, die auf benachbarte Blöcke zugreifen, zu einem Auftrag zusammengefaßt werden können. Dieses Prinzip wird seit mehreren Jahren im Linux-Blocktreiber eingesetzt und findet entsprechend auch in L4IDE Verwendung (siehe Abschnitt 2.4.2). Da es auch nach einem Zusammenfassen noch erforderlich ist, die Urheber der Teilaufträge über deren Abarbeitung zu informieren, müssen diese Teilaufträge als eine Einheit erhalten bleiben. Im Linux-Bustreiber wird diese Einheit als BIO (für Block I/O Operation) bezeichnet. Im BDDF soll sie ebenso heißen. Ein Auftrag erfüllt dann die Funktion eines Containers für eine oder mehrere BIOs und dient als Verwaltungseinheit in der BDDF-Instanz.

3.1.1 Die zu verwendende Blockgröße

Blockgeräte arbeiten – wie der Name bereits andeutet – blockorientiert. Das heißt, das kleinste verarbeitbare Daten-Quantum ist ein Block. Dabei besitzen alle Blöcke die gleiche feste Größe. Jeder Datenaustausch hat den Umfang eines Vielfachen davon. Um die gewünschten Blöcke benennen zu können, werden sie von Null beginnend durchnummeriert. Das Äquivalent zu den Blöcken der logischen Sicht auf Blockgeräte sind die Sektoren der physischen Geräte. Die BDDF-Instanz übernimmt damit auch eine Umsetzfunktion zwischen logischen Blöcken und physischen Sektoren.

Es stellt sich die Frage, wie groß die Blöcke sein sollen. Dabei sind drei Ansätze denkbar. Die Blockgröße ist

(a) identisch zur Sektorgröße des angesprochenen Gerätes
(b) konstant für alle Geräte
(c) vom Client bestimmbar.

Im ersten Fall muß sich ein Client auf unterschiedliche Blockgrößen einstellen, was einen gewissen Anpassungsaufwand für jeden einzelnen Client bedeutet. Dieser kann im Fall (b) vermieden werden. Hier muß sich ein Client nur auf eine, vor allem konstante Blockgröße einstellen. Der größte Anpassungsaufwand liegt hier bei der BDDF-Instanz und fällt nur einmal bei deren Implementierung an. Dabei ist es allerdings nahezu unerheblich, ob die unterschiedlichen Sektorgrößen der physischen Geräte an eine für alle Clients konstante oder eine variable Blockgröße angepaßt werden. Letzteres Verhalten entspricht Fall (c), wenn diese variable Blockgröße vom Client bestimmt werden kann. Das bietet den Vorteil, daß ein Client mit der für seine Zwecke idealen Größe arbeiten kann. Ein physisches Blockgerät kann aber immer nur ein Vielfaches seiner nativen Sektorgröße übertragen. Fordert ein Client eine Blockgröße, die kein ganzzahliges Vielfaches der Sektorgröße ist, so kann dies nicht erfüllt und muß von der BDDF-Instanz abgelehnt werden. Eine mögliche Alternative besteht darin, die überschüssigen Daten in einen dafür erstellten Puffer zu übertragen und dem Client so vorzuenthalten. Im Falle eines schreibenden Auftrages müssen diese Überschußpuffer aber zuvor mit den Daten der zugehörigen Sektoren gefüllt werden, um einen Datenverlust zu vermeiden. Dies erfordert zusätzliche Zugriffe auf das Medium und reduziert so den erzielbaren Durchsatz erheblich. Diese Lösung ist damit inakzeptabel. Im BDDF soll die Blockgröße also vom Client bestimmt werden können, aber auf ein Vielfaches der Sektorgröße beschränkt sein.

3.1.2 Die Granularität der einsetzbaren I/O-Scheduler

Um für verschiedene Laufwerke auch verschiedene Scheduling-Strategien durchsetzen zu können, sollen in einer BDDF-Instanz mehrere I/O-Scheduler parallel einsetzbar sein. Da ein Bustreiber aber nur Kenntnis über die vom ihm verwalteten physischen Laufwerke besitzt, muß die Partitions-Behandlung stattgefunden haben, bevor die Aufträge an ihn gesendet werden. Das bedeutet, daß die eventuell für die beteiligten logischen Laufwerke eingesetzten unterschiedlichen Scheduler untereinander koordiniert werden müssen. Denn die Aufträge für diese Laufwerke greifen auf dieselbe Ressource zu. Bei der Untersuchung der Leistung von Cello [SV98] wurde ersichtlich, daß diese Koordinierung komplex und sehr empfindlich gegen kleinste Änderungen der Systemparameter ist. Für das BDDF soll deshalb die Beschränkung auf einen I/O-Scheduler pro physischem Laufwerk gelten.

Sollten dennoch mehrere I/O-Scheduler pro physischem Laufwerk erforderlich sein, kann auch ein Scheduler-Framework wie Cello (siehe Abschnitt 2.1.4) eingesetzt werden. Dieses kann dann mehrere Unter-Scheduler verwenden und gegenüber der BDDF-Instanz trotzdem wie einer erscheinen. Für diese Funktion ist es lediglich nötig, auch die Kennung des angesprochenen logischen Laufwerkes mit dem Auftrag an den I/O-Scheduler mitzuliefern.

3.1.3 Das zu verwendende Echtzeit-Modell

Für die Ermittlung der Angaben, die ein Client dem BDDF über seine Echtzeit-Aufträge liefern muß, ist zuerst die Festlegung des im BDDF verwendeten Echtzeit-Modells nötig. Das in DROPS verwendete Modell ist das des Quality-Assuring-Scheduling (siehe Abschnitt 2.1.1). Es ist sehr allgemein, und es können alle Anwendungsfälle abgedeckt werden. Denn jeder Echtzeit-Strom ist in eine Menge durch dieses Modell darstellbare Teilströme zerlegbar. Lediglich der Fall, daß die Bearbeitung der Aufträge einer Periode eines Stromes von der erfolgreichen Abarbeitung eines zu einem anderen Strom gehörenden Auftrages abhängig ist, läßt sich damit nicht darstellen. Ein zugehöriges Zulassungskriterium ist äußerst komplex und benötigt viel Rechenzeit. Da diese Spezialfälle aber eher theoretischer Natur und dementsprechend selten sind, ist ein Kompromiß möglich. Die abhängigen Ströme werden als voneinander unabhängig betrachtet. Dadurch wird in diesen Sonderfällen mehr der Ressource reserviert als nötig – es geht Bandbreite verloren. Andererseits ist für die vereinfachte Berechnung weniger Rechenzeit erforderlich – und zwar in allen Fällen.

Das Echtzeit-Modell von QAS wird im DAS-Scheduling (siehe Abschnitt 2.1.2) speziell für die Verwendung von Festplatten angepaßt. Allerdings wird dabei vorläufig[7] von einheitlichen Perioden für alle Echtzeit-Ströme ausgegangen. Diese Periodendauer ist in DAS-Scheduling eine Eigenschaft des Systems. Da in diesem Abschnitt jedoch lediglich das ver-

7 Eine Erweiterung auf unterschiedliche Periodendauern ist vorgesehen.

wendete Modell und nicht der konkrete Scheduler interessiert, kann die Periodendauer ebenfalls als eine Eigenschaft des Echtzeit-Stromes angenommen werden. Weiterhin ist in DAS-Scheduling die Größe eines Auftrages identisch zur Blockgröße des Systems. Ein Client, der mehr Daten pro Auftrag benötigt, ist somit gezwungen, diesen in mehrere Aufträge der Blockgröße des Systems zu unterteilen. Durch den so entstehenden Protokoll-Mehraufwand kann sich die Systemleistung drastisch reduzieren. Das Aufteilen des Auftrages kann durch den zusätzlichen Parameter *Größe eines Auftrages* vermieden werden.

In einigen anderen Modellen ([Reu01], [Ven02]) wird die Angabe des Echtzeit-Stromes in *Größe eines Auftrages* und *Bandbreite des Stromes* propagiert. Aus diesen Parametern läßt sich die Auftragsfrequenz, also auch die Periodendauer, ermitteln. Allerdings gilt das nur unter der Bedingung, daß innerhalb einer Periode genau ein solcher Auftrag abgearbeitet wird. Deutlich mehr Planungsfreiheit für den Scheduler und eine genauere Darstellung der Puffermöglichkeiten des Clients wird erreicht, indem optional der Parameter *Periodendauer* angegeben werden kann. Wird dieser nicht angegeben, liegt ein aperiodischer Strom vor.

Einige I/O-Scheduling-Algorithmen (wie Cello, siehe Abschnitt 2.1.4) erfordern Wissen über die Art des Echtzeit-Stromes. Dafür ist der Parameter *Art des Stromes* vorgesehen.

Das eben von DAS-Scheduling abgeleitete Echtzeit-Modell soll als Standard für das BDDF gelten. Es beschreibt Echtzeit-Ströme durch das Tupel

$$T_i = \left(bw_i,\, s_i,\, q_i,\, t_i,\, k_i\right),\ i \in \mathbb{N} \text{ mit}$$

bw_i	Bandbreite des Stromes,
s_i	Größe eines Auftrages,
q_i	Qualitätsparameter,
t_i	Periodendauer,
k_i	Art des Stromes.

Da das Echtzeit-Modell für das BDDF indirekt auf dem von QAS basiert, lassen sich damit ebenfalls Ströme mit nur optionalen Aufträgen und Ströme mit nur unabdingbaren Aufträgen darstellen. Mischformen sind durch ein Aufteilen in optionale und unabdingbare Ströme beschreibbar. Die Darstellung sporadisch auftretender Einzelaufträge, die dennoch Echtzeit-Eigenschaften besitzen, also innerhalb einer bestimmten Zeit abgearbeitet sein müssen, ist mit diesem Modell nicht möglich. Soll ihre Deadline auf jeden Fall eingehalten werden, muß ihr Auftreten also abgeschätzt und ein separater Strom für sie reserviert werden. Ist eine Garantie für das Erreichen dieser Deadline nicht unbedingt erforderlich, genügt es, den Auftrag „schnellstmöglich“ auszuführen und ihn gegenüber den Nicht-Echtzeit-Aufträgen bevorzugt zu behandeln. Es ist die Aufgabe des Schedulers, dies so gut wie ihm möglich zu realisieren.

3.1.4 Aufgaben der Blockgeräte-Verwaltung

Die Blockgeräte-Verwaltung soll die Schnittstellen einer BDDF-Instanz untereinander verbinden und synchronisieren. Darüber hinaus hat sie die Aufgaben zu realisieren, die sich keiner Schnittstelle zuordnen lassen.

Ein physisches Gerät kann in mehrere logische Geräte (Partitionen) unterteilt sein. In der BDDF-Instanz muß also auch eine Anpassung der relativen Sektornummern der logischen Geräte an die absoluten Sektornummern des entsprechenden physischen Gerätes vorgenommen werden – die *Partitions-Behandlung* (engl. Partition Handling). Da die I/O-Scheduler nur pro physischem Laufwerk arbeiten, muß die Partitions-Behandlung stattfinden, bevor ein Auftrag an einen Scheduler gereicht wird. Sie ist damit Aufgabe der Blockgeräte-Verwaltung.

Für eine Beurteilung des Verhaltens eines bestimmten Laufwerkes oder eines bestimmten Clients kann es nötig sein, Buch über verschiedene Parameter zu führen. Derartige Parameter sind die auftretenden Auftrags-Bearbeitungszeiten oder die Häufigkeit der eintreffenden Aufträge. Prinzipiell kann diese Buchführung auch von dem zuständigen Scheduler übernommen werden. Allerdings gehen diese Informationen dann bei einem Umschalten auf einen anderen Scheduler verloren. Da ein Scheduler mittels dieser Daten bestimmte Entscheidungen schon im voraus treffen könnte, sollen sie unabhängig von der Blockgeräte-Verwaltung gesammelt werden.

Eine solche Buchführung setzt auch voraus, daß die Blockgeräte-Verwaltung Wissen über die einzelnen stattfindenden Aktivitäten besitzt, insbesondere auch Wissen über die im BDDF vorhandenen Aufträge. Deshalb soll sie die Aufträge verwalten und lediglich Teilaufgaben an die entsprechenden Instanzen weitergeben.

3.1.5 Ablauf der Auftrags-Bearbeitung

Der zentrale Vorgang im BDDF ist die Auftrags-Bearbeitung. Um die Anforderungen an die einzelnen Schnittstellen des BDDF besser erkennen zu können, ist es zunächst erforderlich, diesen Vorgang zu verdeutlichen. Er ist dazu in Abbildung 6 so dargestellt, wie er sich aus den bisherigen Festlegungen ergibt. Die angegebenen Schritte sind der Abbildung nachfolgend beschrieben.

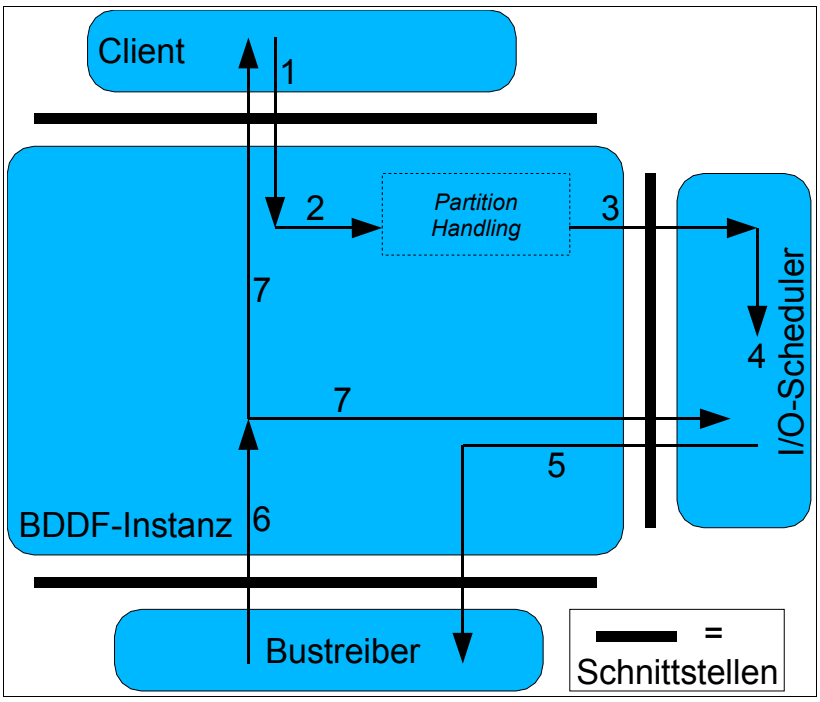

Abbildung 6: *Der Weg eines Auftrages im BDDF*

(1) Der Client erzeugt einen Auftrag und sendet diesen mit Hilfe der noch zu definierenden Client-Schnittstelle an die BDDF-Instanz.

(2) Die BDDF-Instanz berechnet aus der relativen Angabe der angesprochenen Datenblöcke die absolute Angabe der Sektoren auf dem Datenträger (*Partition Handling*).

(3) Die BDDF-Instanz gibt den Auftrag an den I/O-Scheduler, welcher ihn in die entsprechende Warteschlange einreiht. Die dafür vorgesehene Schnittstelle ist noch zu entwerfen.

(4) Der I/O-Scheduler faßt den Auftrag mit einem eventuell passenden, bereits in der Warteschlange befindlichen anderen Auftrag zusammen, sofern seine Strategie dies erlaubt.

(5) Der Auftrag wird über die noch zu entwickelnde Bustreiber-Schnittstelle an den Bustreiber übergeben, sobald dieser für die Bearbeitung frei ist.

(6) Bei beendeter Bearbeitung oder im Fehlerfall benachrichtigt der Bustreiber die BDDF-Instanz über das Ereignis.

(7) Die BDDF-Instanz informiert den I/O-Scheduler über die Bearbeitung des Auftrages und benachrichtigt danach den Client. Der Auftrag kann nun gelöscht werden.

3.2 Der Namensraum

Um Blockgeräte benennen zu können, werden ihnen Namen zugeordnet. Es gibt noch keinen allgemeingültigen Mechanismus, diese Namen fest auf dem Datenträger zu spei-

chern. Deshalb müssen andere Benennungsversuche unternommen werden, welche meistens auf einem mehr oder weniger aufwendigen Nummerierungsschema beruhen. Im Rahmen dieser Arbeit soll auch ein Namensraum entwickelt werden, der eine zufriedenstellende Benennung[8] der einzelnen Laufwerke zuläßt, ohne jedoch zusätzliche Informationen auf diesen zu hinterlegen.

3.2.1 Eindeutigkeit und Konstanz

Der Name eines Blockgerätes muß im System eindeutig und konstant sein. Wäre er nicht eindeutig, könnten verschiedene Geräte nicht voneinander unterschieden werden. Wäre er nicht konstant, könnte ein Gerät nicht identifiziert werden – es würde über die Zeit als unterschiedliche Geräte angesehen. Beide Fälle können beispielsweise in einem DOS-basierten System auftreten. Hier werden die vorhandenen Geräte einfach durchnummeriert, beginnend mit `C:` für Festplattenlaufwerke. Wird eine Partition oder eine ganze neue Festplatte, bezüglich der Nummerierung, zwischen zwei bereits existierende Geräte eingefügt, verschieben sich alle nachfolgenden Gerätenamen um eins. Ein vorher `E:` genanntes Gerät heißt jetzt `F:` – hier ist die Konstanz verletzt. Und hinter dem Namen `E:` verbirgt sich jetzt ein völlig anderes Gerät – die Eindeutigkeit wurde verletzt. Um solche Unzulänglichkeiten zu vermeiden sind die zwei Grundforderungen für die Namen in einem Namensraum also Eindeutigkeit und Konstanz.

Die einzige Möglichkeit, die Forderung der Konstanz umzusetzen, besteht per Definition darin, nach jeder Reinitialisierung jedem Gerät genau den Namen zu geben, den es auch davor besaß.

Die Eindeutigkeit läßt sich realisieren, indem jeder Name nur einmal verwendet wird. Wird das System reinitialisiert, muß die Menge der jetzt vergebenen Namen disjunkt der Menge aller jemals vorher vergebenen Namen sein. Bei dieser Vorgehensweise ist die Konstanz aber nicht gegeben, denn ein Gerät besitzt nach jeder Reinitialisierung einen anderen Namen. Es muß also eine weitere Möglichkeit gefunden werden. Dazu ist es nötig, eine in der Informatik sehr verbreitete Organisationsform näher zu betrachten – den Baum.

3.2.2 Der Baum als Organisationsform

Ein Baum besteht aus Knoten und Zweigen. Die Knoten sind in Ebenen aufgeteilt. In der ersten Ebene gibt es genau einen Knoten – den Wurzelknoten. Zweige stellen die Verbindung zwischen den Knoten benachbarter Ebenen dar. Von jedem Knoten können beliebig viele Zweige zu den Knoten der nächsten Ebene führen. Aber zu jedem Knoten, außer dem Wurzelknoten, führt genau ein Zweig.

8 Ein Blockgerät sollte so lange identifizierbar sein, wie es nicht physisch verändert wird (z.B. Aus- und Einbau).

Den Knoten werden Bezeichnungen gegeben. Dabei muß jede Bezeichnung unter allen Knoten derselben Ebene desselben Vaterknotens eindeutig sein. Die Knoten des so entstandenen Baumes lassen sich finden, indem der Weg zu ihnen beschrieben wird. Diese Wegbeschreibung besteht aus der Aneinanderreihung der Bezeichnungen der Knoten, die ausgehend vom Wurzelknoten bis zum Zielknoten der Reihe nach durchfahren werden. Die Bezeichnung des Wurzelknotens braucht dabei nicht angegeben zu werden, da sie bei allen Wegbeschreibungen als erstes Element auftritt.

Die genannten Zusammenhänge sind in Abbildung 7 an einem Beispiel dargestellt. Der Weg vom Wurzelknoten zum Knoten *o* ist markiert. Seine Beschreibung lautet *B.2.o*.

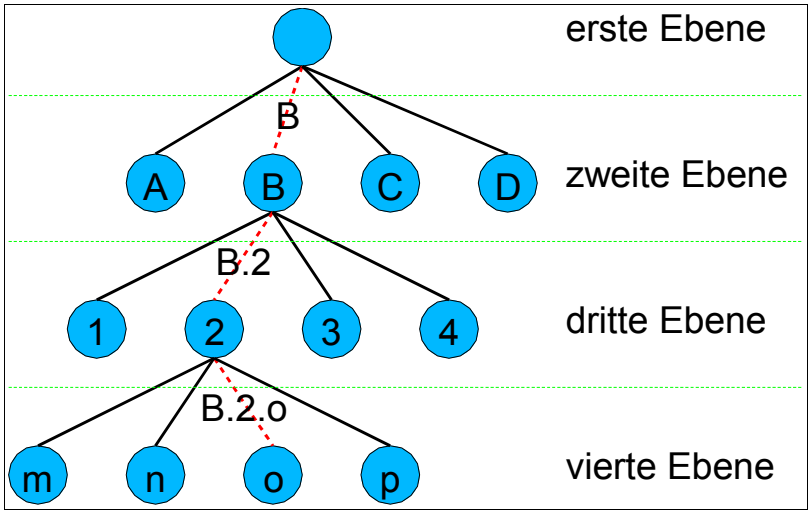

Abbildung 7: *Baum mit markiertem Weg*

Auf Grund der Natur eines Baumes – nämlich nur vorwärts zu verzweigen – gibt es zu jedem Knoten genau einen Weg, welcher von dem jedes anderen Knotens verschieden ist. Eine weitere Eigenschaft eines Baumes ist, daß sich Änderungen in einem Zweig, abgesehen von den untergeordneten, nicht auf andere Zweige auswirken. Somit ändert sich auch der Weg zu einem Knoten in einem anderen Zweig nicht.

Wird die Wegbeschreibung zu einem Knoten als Name für selbigen verwendet, gelten für diesen auch die Eigenschaften des Weges zum Knoten. So gibt es für jeden Knoten genau einen von den anderen Knoten verschiedenen Weg, und mithin auch genau einen von den anderen Knoten verschiedenen Namen. Die Grundforderung nach Eindeutigkeit ist damit erfüllt. Und da der Weg zu einem Knoten durch Änderungen in anderen Zweigen unverändert bleibt, ändert sich dabei auch der Name dieses Knotens nicht. Dies erfüllt die Grundforderung nach Konstanz.

3.2.3 Der Baum für Partitionen

Physische Laufwerke werden üblicherweise partitioniert. Das heißt, der Speicherplatz, den sie zur Verfügung stellen, wird aufgeteilt. Diese Teile können unterschiedlich groß sein und werden Partitionen genannt. Durch diese Maßnahme ist ein Nutzer in der Lage, auf nur einer Festplatte unterschiedliche Betriebssysteme mit unterschiedlichen Dateisystemen zu betreiben. Unter UNIX-artigen Betriebssystemen ist es im Sinne einer hohen Arbeitsgeschwindigkeit üblich, sowohl ein Dateisystem für die Nutzdaten als auch ein Dateisystem für das Auslagern von Speicherseiten parallel zu benutzen. Dafür werden ebenfalls mehrere Partitionen benötigt.

Damit das Betriebssystem weiß, an welcher Stelle der Festplatte sich eine bestimmte Partition befindet, ist diese Information in der Partitionstabelle eingetragen. Es gibt mehrere Typen von Partitionstabellen. Der im PC-Bereich meistverbreitete Typ ist die MSDOS/Linux-Partitionstabelle. Darüber hinaus gibt es aber noch Typen für BSD, Amiga, IBM und andere. Allen gemein ist, daß sie sich als Baum darstellen lassen. Partitionen sind entweder in einer Liste meist fester Länge eingetragen oder in einer verketteten Liste von Partitionen organisiert. Mischformen sind ebenfalls möglich (z.B. die MSDOS/Linux-Partitionstabelle). Abbildung 8 verdeutlicht die folgenden Zusammenhänge. Die einzelnen Elemente einer Liste fester oder variabler Länge lassen sich als Knoten einer Ebene darstellen. Diese tragen als Bezeichnung die jeweilige Nummer des Elementes in der Liste. Diese Vorgehensweise ist auch anzuwenden, wenn die Partitionen in einer verketteten Liste organisiert sind, in der es immer nur ein Folgeelement gibt. Andernfalls[9] bilden die Folgeelemente, zu denen ein Element direkt verweist, die Knoten der nächsthöheren Ebene. Den Wurzelknoten des Baumes für die Partitionen eines physischen Laufwerkes bildet das physische Laufwerk selbst.

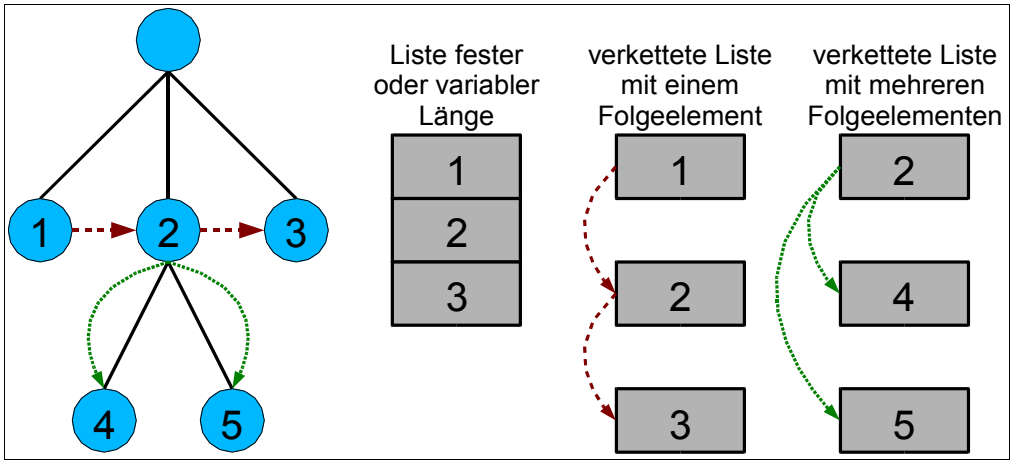

Abbildung 8: *Abbildung von Elementen aus verschiedenen Listentypen auf einen Baum*

9 Die Folgeelemente sind demnach wieder in einer Liste fester oder variabler Länge eingetragen.

3.2.4 Die Baumstruktur für das gesamte Blockgerätesystem

Das bisherige Schema betrachtet nur einzelne physische Laufwerke und deren Partitionen. In einem Gesamtsystem können aber weitere Laufwerke existieren – sogar in verschiedenen Bussystemen. Linux liefert den Lösungsansatz für dieses Problem, denn hier teilt sich der Namensraum nach Bussystemen auf. Für IDE-Laufwerke wird das Prefix *h* verwendet und für SCSI-Laufwerke das Prefix *s*. Weiterhin werden im Falle des IDE-Bussystemes alle potentiellen Laufwerke durchnummeriert. Es existiert also für jeden Anschluß ein eigener Name, unabhängig davon, ob daran ein Laufwerk angeschlossen ist oder nicht. Sowohl im IDE-System als auch im SCSI-System kann es mehrere Controller geben, welche wiederum mehrere physische Busse (oft Kanäle genannt) verwalten können. Zusätzlich kann ein SCSI-Gerät auch in mehrere Untergeräte unterteilt sein. Dadurch ergibt sich eine Fülle an möglichen Ebenen zwischen Bustreiber und physischem Laufwerk. Hierfür eine flache Hierarchie zu benutzen könnte zu Inkonsistenzen führen, weshalb auch für diese Ebenen eine Baumstruktur zu verwenden ist.

Das existierende Schema muß also entsprechend erweitert werden. Ausgehend vom Wurzelknoten des gesamten Namensraumes für Blockgeräte gibt es für jedes Bussystem einen eigenen Knoten. Diesem untergeordnet existieren beliebig viele Ebenen für Untergeräte. An deren Endknoten befinden sich nun die Knoten für die angeschlossenen physischen Laufwerke. Diese Knoten wiederum sind die in Abschnitt 3.2.3 erklärten Wurzelknoten für ein physisches Laufwerk und seine Partitionen.

Bisher wurden nur Festplatten als physische Laufwerke betrachtet. Im Falle eines optischen Laufwerkes (CD- oder DVD-Laufwerke) gilt das Schema mit der Abänderung, daß ein solches Laufwerk keine Partitionen enthält. Deshalb besitzt der entsprechende Laufwerksknoten keine weiteren Zweige.

3.2.5 Zusammenfassung

Zu Beginn wurde festgestellt, daß es noch kein allgemeingültiges Namensschema für die Benennung von Blockgeräten gibt. Ein solches müßte auch die im weiteren untersuchten Forderungen nach Eindeutigkeit und Konstanz erfüllen. Danach wurde festgestellt, daß diese Forderungen durch die Verwendung von Namensbäumen erfüllt werden, wenn als Name jedes Knotens die Aneinanderreihung der Bezeichnungen der vom Wurzelknoten zu ihm passierten Knoten verwendet wird. Dann wurden die verschiedenen Arten der Partitionstabellen auf das so entstandene Namensschema abgebildet. Abschließend wurde das Schema um die Betrachtung der verschiedenen Bussysteme, ihrer Untergeräte und der daran angeschlossenen physischen Laufwerke erweitert.

Damit ergeben sich Bäume wie in Abbildung 9 dargestellt. Dem Wurzelknoten folgt die Ebene der Knoten für die Bussysteme (hier exemplarisch SCSI, IDE und USB), beliebig viele Ebenen der Untergeräte (hier ist dies nur für IDE gezeigt), die Ebene der physischen Laufwerke und beliebig viele Ebenen der Partitionen. Die genaue Benennung der einzelnen Knoten ist ein Implementierungsdetail und könnte sich für die Bussystem- und Anschlußebene an Linux orientieren.

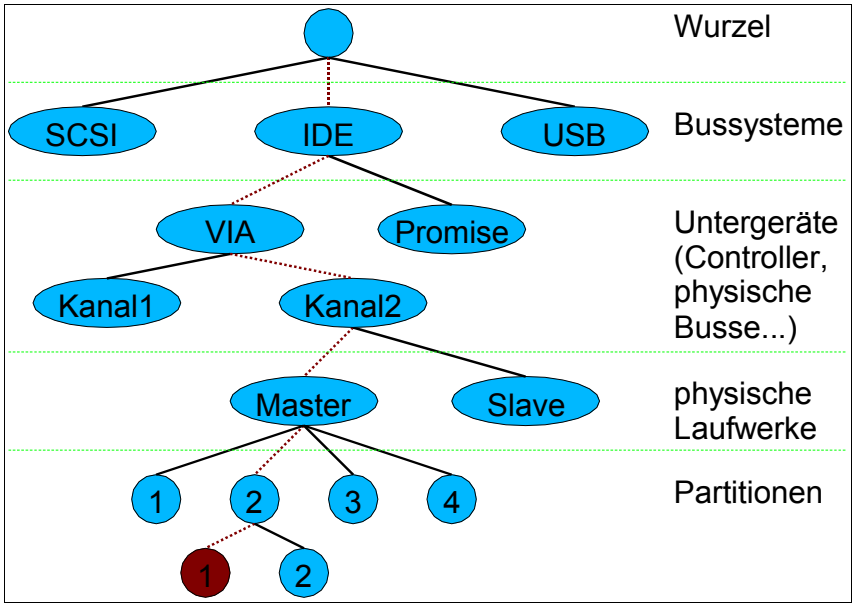

Abbildung 9: *Beispiel eines Gesamtbaumes*

In dem entstandenen Namensraum benennt ein Client eine Partition also eindeutig und konstant, wenn er die Bezeichnung des Bussystem-Knotens, gefolgt von den Bezeichnungen der Knoten der Untergeräte, den Bezeichnungen der Laufwerks-Knoten und die Bezeichnungen der von da aus zum Partitionsknoten passierten Knoten aneinanderreiht. Zur eindeutigen Schreibweise ist zwischen die einzelnen Namensglieder ein Trennzeichen – hier der Dezimalpunkt – zu setzen. Für die in Abbildung 9 hervorgehobene Partition ergibt sich demnach der Name *IDE.VIA.Kanal2.Master.2.1*.

3.3 Die Schnittstelle zum Client

Die Schnittstelle zwischen einer BDDF-Instanz und ihren Clients soll es ermöglichen, daß eben diese Clients

- Echtzeit- und Nicht-Echtzeit-Aufträge an die Instanz übersenden
- die zugehörigen Daten austauschen
- Informationen über die Instanz erfahren
- Einstellungen an der Instanz vornehmen können.

Die im folgenden zu entwerfende Schnittstelle stellt nur das Konzept dar. Ob die einzelnen Funktionen direkt über IPC beziehungsweise RPC aufrufbar sind, oder ob sie durch eine Funktionsbibliothek und einen Satz anderer Funktionen emuliert werden, ist eine Entscheidung der konkreten Implementierung.

3.3.1 Senden von Aufträgen

Eine BDDF-Instanz verwaltet mehrere logische Blockgeräte – seien es Partitionen oder physische Laufwerke. Es ist denkbar, daß nicht nur ein Client (beispielsweise ein Block-Cache) exklusiv auf alle Blockgeräte zugreift, sondern daß es mehrere Clients gibt, von denen jeder Daten mit einem anderen Gerät austauscht. Dies ist unter anderem dann der Fall, wenn auf jedem Gerät ein anderes Dateisystem enthalten ist. Jeder Dateisystem-Treiber ist aber ein Client der BDDF-Instanz. Es muß also möglich sein, daß sich ein Client eine Referenz auf ein bestimmtes Blockgerät erbittet (`open_device`) und diese nach beendeter Arbeit wieder abgibt (`close_device`). Beim Öffnen des Gerätes muß der Client auch die von ihm gewünschte Blockgröße angeben. Diese muß ein ganzzahliges Vielfaches der Sektorgröße des zu öffnenden Gerätes betragen.

Ein Client kann *nur lesend* oder *lesend und schreibend* auf ein Blockgerät zugreifen. Dabei ist es durchaus möglich, daß mehrere Clients auf dasselbe Gerät zugreifen wollen. So könnte ein Prozeß eine lesende Laufwerksüberprüfung durchführen, während der Dateisystemtreiber weiter seine Schreib-/Lese-Referenz auf dieses Gerät besitzt. Eine BDDF-Instanz sollte jedoch verhindern, daß zwei Clients gleichzeitig auf dasselbe Gerät schreiben können. Andernfalls droht eine Inkonsistenz der Daten, die zum Verlust derselben führen kann. Entsprechend darf die Instanz pro Gerät – inklusive dessen untergeordneten Geräten (siehe Abschnitt 3.2.3) – nur eine Schreib-/Lese-Referenz vergeben. Lese-Referenzen dürfen dagegen beliebig viele verteilt werden. Weiterhin kann es sein, daß ein Client zwar nur lesend auf ein Gerät zugreift, aber anderen Clients den Zugriff auf dieses Gerät verbieten will. Deshalb ist es nötig, daß ein Client bei `open_device` zusätzlich zur Angabe der Zugriffsmodi *nur lesend* oder *lesend und schreibend* angibt, ob er das *exklusiv* tun möchte.

Für den eigentlichen Zugriff auf das gewünschte Blockgerät muß ein Client seine Aufträge an die BDDF-Instanz senden können. Die Bearbeitung eines Auftrages kann aber eine sehr lange Zeit in Anspruch nehmen – bis zu mehreren hundert Millisekunden. Da diese Zeit durch Warten auf die Auftrags-Beendigung verschenkt würde, sollte der Client entscheiden können, ob er eine synchrone (`send_request_sync`) oder asynchrone (`send_request_async`) Benachrichtigung wünscht. Bei einer synchronen Benachrichtigung blockiert die Programmausführung so lange, bis der Auftrag komplett bearbeitet wurde. Bei einer asynchronen Benachrichtigung wird die Programmausführung direkt nach dem Absenden des Auftrages fortgesetzt. In diesem Fall wird der Client über die Bear-

beitung des Auftrages durch Aufruf einer von ihm angegebenen Rückruf-Funktion informiert, oder er muß selbständig bei der BDDF-Instanz den momentanen Status des Auftrages erfragen. Dazu soll eine Funktion `get_request_status` existieren. Sollte der Status lauten, daß ein Fehler bei der Bearbeitung aufgetreten ist, so muß dem Client die Möglichkeit gegeben werden, eine genauere Fehlerbeschreibung zu erhalten.

3.3.2 Die Behandlung von Echtzeit-Strömen

Für die Bearbeitung von Echtzeit-Strömen (siehe Abschnitt 2.1) muß der BDDF-Instanz vom Client mitgeteilt werden, mit welchem Blockgerät der Strom assoziiert ist und welche Eigenschaften er besitzt. Die Instanz kann dann mit Hilfe des zuständigen I/O-Schedulers entscheiden, ob ein solcher Strom akzeptiert werden kann. So könnte beispielsweise nicht genügend Bandbreite zur Befriedigung seiner Bedürfnisse zur Verfügung stehen. Es kann vorkommen, daß ein Client zuerst die Zusicherung aller seiner Ströme abwarten möchte, bevor er diese startet. Er hat so die Möglichkeit, deren Eigenschaften zu ändern und erneut anzufragen, wofür ihm natürlich der Grund der Ablehnung mitgeteilt werden muß. Der Aufbau eines Echtzeit-Stromes muß also in die beiden Phasen *Reservierung* (`reserve_stream`) und *Start* (`start_stream`) aufgeteilt werden. Ist ein Strom akzeptiert, erhält der Client eine Referenz auf diesen. Er kann den Strom jetzt starten und die zugehörigen Echtzeit-Aufträge – versehen mit der Referenz – an die BDDF-Instanz senden. Wird ein Strom nicht mehr benötigt, muß der Client dies der Instanz mittels einer Funktion `close_stream` mitteilen, damit diese die dafür reservierten Ressourcen wieder freigeben kann.

Aus dem in Abschnitt 3.1.3 entwickelten Echtzeit-Modell für das BDDF ergeben sich die vom Client per `reserve_stream` zu übermittelnden Strom-Eigenschaften *Anzahl der Aufträge pro Periode*, *Größe eines Auftrages*, *Periodendauer*, *Qualitätsparameter* und *Art des Stromes*. Sollte der für das angesprochene Gerät zuständige I/O-Scheduler nur eine feste Periodendauer unterstützen, muß er entweder eine Umsetzung auf diese vornehmen oder den Strom ablehnen.

Ist ein Echtzeit-Strom von der BDDF-Instanz akzeptiert worden, muß er vom Client gestartet werden, um Aufträge abarbeiten zu können. Dies kann entweder explizit durch eine Funktion `start_stream` oder implizit durch das Senden des ersten Auftrages für den Strom geschehen. Ein Client ist durch das verwendete Echtzeit-Modell gezwungen, Ströme mit sowohl optionalen als auch unabdingbaren Aufträgen in mehrere Teilströme zu trennen. Die Aufträge einer Periode des Ursprungs-Stromes sollten dann aber innerhalb der entsprechenden Perioden der Teilströme abgearbeitet werden, um weiterhin synchron zu arbeiten. Beim impliziten Starten des Stromes ist nicht sichergestellt, daß die Aufträge der Teilströme auch innerhalb derselben Periode eintreffen. Es können so Verschiebungen der

Startzeitpunkte der Teilströme um ein oder sogar mehrere Periodendauern auftreten – die Ströme sind nicht synchron. Aus diesem Grunde ist ein explizites Starten unter Angabe des Zeitpunktes nötig, zu dem der erste Auftrag abgearbeitet worden sein soll. Hierfür ist eine einheitliche Zeitbasis von Client und BDDF-Instanz erforderlich.

3.3.3 Der Datenaustausch

Der eigentliche Datenaustausch findet in Form der übermittelten Aufträge statt. Diese enthalten dazu die wesentlichen Informationen

- ob es ein Lese- oder ein Schreib-Auftrag ist
- ob es sich um einen Echtzeit-Auftrag handelt
- welchem Echtzeit-Strom der Auftrag eventuell zugehört
- an welches Gerät der Auftrag gerichtet ist
- welche Blöcke des Gerätes zu lesen oder zu schreiben sind
- Art und Ort des zugehörigen Puffers.

Das angesprochene Gerät kann durch die mit `open_device` erhaltene Referenz angegeben werden. Ebenso kann mit der Angabe des Echtzeit-Stromes verfahren werden, welcher durch die mit `reserve_stream` gegebene Referenz darstellbar ist. Bis auf die Art und den Ort des Puffers sind alle weiteren Informationen ein Detail der konkreten Implementierung.

Die Speicherverwaltung unter DROPS setzt auf dem Konzept der Dataspaces auf. Je nach Bustreiber wird zur Bearbeitung eines Auftrages entweder die physische Adresse des entsprechenden Puffers oder dessen virtuelle Adresse im Adreßraum des Treibers benötigt. Beides kann durch einen Dataspace-Zeiger mit Hilfe entsprechender Funktionen erhalten werden, wie bereits in Abschnitt 2.2 gezeigt wurde. Es ist denkbar, daß beispielsweise ein Dateisystem-Treiber einen Dataspace für eine große Datei anlegt, diesen aber nur stückweise mit Daten füllen will. Daher muß es möglich sein, daß als Puffer einer Datenübertragung nicht nur ein Dataspace an sich, sondern auch ein Teilbereich daraus verwendbar ist. Die Funktion `l4dm_share` läßt es aber nur zu, den kompletten Dataspace zwischen mehreren Prozessen zu teilen. Dabei muß der Client dem Block- und dem Bustreiber vertrauen, denn diese könnten somit lesend oder gar schreibend auf den gesamten Dataspace zugreifen. Ist dies nicht akzeptabel, muß pro Auftrag ein separater Dataspace angelegt werden. Ein Restrisiko bleibt allerdings immer, denn hat ein anderer Prozeß erst einmal die physische Adresse eines Dataspace-Bereiches in Erfahrung gebracht, kann er diese auch nach Entzug seiner Rechte am Dataspace weiter nutzen. Es besteht somit grundsätzlich ein gewisses Vertrauensverhältnis zwischen Client und Treiber.

Unter dieser Einschränkung sind Dataspace-Zeiger für die Angabe des Puffers geeignet und sollen von der Client-Schnittstelle verwendet werden. Dabei muß es auch möglich sein, diese Zeiger in Form einer Scatter/Gather-Liste anzugeben. Denn insbesondere von Clients wie Block-Caches oder Dateisystemtreibern werden Aufträge gesendet, welche aus Teilaufträgen verschiedener ihrer Clients zusammengestellt wurden. Diese enthalten deshalb auch mehrere Pufferbeschreibungen.

Die Operationen zum Erfragen der physischen Adresse eines Dataspaces, zum Anfordern einer Einblendung in den Adreßraum des Bustreibers und zum Teilen des Dataspaces mit anderen Prozessen benötigen verhältnismäßig viel Rechenzeit. Da die Operationen bei jedem zu bearbeitenden Auftrag fällig werden, erzeugen sie so eine nicht unerhebliche Last. Die Clients einer BDDF-Instanz werden häufig Dateisystemtreiber sein, die in aller Regel einen Blockcache enthalten. Dieser besteht meist aus einem reservierten kontinuierlichen Speicherbereich, von welchem sich leicht die physischen Adressen ermitteln lassen. Für den Fall eines nicht-kontinuierlichen Speicherbereiches besteht außerdem die Möglichkeit, die physischen Adressen der Teilbereiche im voraus zu ermitteln, so daß diese Arbeit nicht während der Auftragserstellung oder -bearbeitung anfällt. Wird diese physische Adresse mit dem Auftrag mitgesendet, entfallen die sonst nötigen Operationen. Es ergibt sich so ein großes Einsparungspotential. Es muß in einem Auftrag also vorgesehen werden, daß ein Client die physische Adresse des Puffers bereits mitliefert.

Zusätzlich zu den Aufträgen, die den Datenaustausch mit einem Blockgerät übernehmen, gibt es noch jene, die Einstellungen vornehmen oder direkt mit dem Bussystemtreiber in Verbindung treten, um beispielsweise Informationen von diesem zu erfragen. Auf Grund der Vielfalt der möglichen Aufträge und deren Anforderungen an einen Puffer (nicht nur Dataspaces sind denkbar), muß einem Client die Möglichkeit eingeräumt werden, einen generischen Auftrag zu senden. Dieser Auftragstyp gehört weder einem Echtzeit-Strom an, noch muß angegeben werden, ob er lesend oder schreibend zugreift. Für die nötige Flexibilität brauchen lediglich je ein Puffer für die an das Gerät zu sendenden und für die vom Gerät zu empfangenden Daten angegeben werden. Damit die Zugriffe auf die Puffer weitestgehend auf den Fall der Übertragung größerer Datenmengen reduziert werden, sollte ein generischer Auftrag eine Kurzbeschreibung enthalten und das angesprochene Gerät bei seiner Beendigung einen Rückgabewert liefern können. Somit können kurze Anfragen nach bestimmten Einstellungen komplett ohne eine Pufferverwendung stattfinden, was die Antwortzeiten reduziert. Die möglichen Kurzbeschreibungen sind dabei von dem Gerät abhängig, an das der Auftrag adressiert ist.

Um auch für zukünftige Mechanismen des Datenaustausches vorbereitet zu sein, ist es erforderlich, die Puffer so allgemein wie möglich anzugeben. Diese Angabe gliedert sich deshalb in den Puffertyp, einen Zeiger auf die Pufferbeschreibung und deren Größe. Die Angabe des Zielgerätes ist weiterhin obligatorisch. Da die BDDF-Instanz im Namensraum

einen eigenen Namen besitzt, ist sie ebenfalls über generische Aufträge ansprechbar. Über sie können alle allgemeinen Einstellungen vorgenommen werden. Die Aufträge für andere Geräte werden dagegen an den zuständigen Bustreiber oder Scheduler gereicht.

Mit den generischen Aufträgen ist es nun möglich, beispielsweise die Existenz oder Größe eines bestimmten Gerätes zu erfragen. Weiterhin läßt sich die unter Linux als IOCTRL bekannte Funktionalität auf die generischen Aufträge abbilden. Es besteht dadurch auch die Möglichkeit, modellspezifische Informationen über ein Gerät zu erfragen – zum Beispiel der eingestellte DMA-Modus oder die aktuelle Drehzahl eines optischen Laufwerkes. Es ist ebenfalls möglich, die Rohdaten einer CD auszulesen, statt nur die Nutzdaten, welche bereits über die normalen Aufträge lesbar sind – vorausgesetzt, der entsprechende Bustreiber unterstützt diese Anfragen.

Die Benutzung eines generischen Auftrages stellt eine Ausnahmesituation dar. Meist handelt es sich dabei um Einstellungsarbeiten, bei denen sich die Rahmenbedingungen für den regulären Betrieb ohnehin ändern (andere Übertragungsraten, lange Verzögerungen durch CD-Wechsel oder ähnliches). Generische Aufträge brauchen somit nicht vom I/O-Scheduler beachtet werden, und ihre Bearbeitung kann getrennt von regulären Aufträgen stattfinden. Allerdings sind generische Aufträge potentiell „wichtiger“ als reguläre und sollten diesen gegenüber bevorzugt behandelt werden. Somit ist ausgeschlossen, daß im praktischen Betrieb das Kopieren einer großen Datei beispielsweise das Auswerfen einer CD blockiert. Entscheidungen, ob derartige Synchronisationen von Aufträgen erforderlich sind, können ohnehin nicht vom BDDF, sondern nur von einer höheren Instanz getroffen werden.

3.3.4 Mögliche Weiternutzung von generic_blk

Im Sinne einer Aufwandsreduzierung wäre es günstig, wenn generic_blk als Schnittstelle zum Client weitergenutzt werden könnte. Im folgenden werden deshalb die eventuell nötigen Veränderungen untersucht.

In generic_blk ist es nicht möglich, explizit eine Referenz auf ein bestimmtes Gerät zu erhalten. Stattdessen werden die Aufträge direkt an das Gerät gesendet. Dieses Vorgehen ist mit der eben entworfenen Schnittstelle nicht vereinbar. Generic_blk ist somit um die entsprechenden Funktionen `open_device` und `close_device` zu erweitern.

Die Funktionen zur Behandlung von Echtzeit-Stömen unterscheiden sich leicht in den übergebenen Parametern, da BDDF und generic_blk unterschiedliche Echtzeit-Modelle zugrunde liegen. Also sind die Funktionen `l4blk_create_stream` und `l4blk_start_stream` entsprechend ihrer BDDF-Pendants `reserve_stream` und `start_stream` zu modifizieren.

Die regulären Aufträge in BDDF und generic_blk ähneln sich naturbedingt sehr. Lediglich die Angabe des Puffers ist in BDDF auf Dataspace-Zeiger beschränkt, bei generic_blk hingegen kann ein beliebiger Typ verwendet werden. Die nötigen Änderungen sind hier aber nicht grundsätzlicher Natur. Das ist bei der Unterstützung generischer Aufträge anders, da diese bei generic_blk nicht vorhanden ist. Hierzu müssen die entsprechenden Strukturen in generic_blk eingeführt werden. Die bisher verwendete Funktion `l4blk_ctrl` kann dagegen entfallen, da deren Funktionalität durch die generischen Aufträge erreicht wird.

Es zeigt sich, daß, trotz einiger nicht unwesentlicher Unterschiede, generic_blk auf die Funktionalität der BDDF-Schnittstelle zum Client angepaßt werden kann. Eine absolute Abwärts-Kompatibilität läßt sich allerdings nicht erreichen. Der Vorteil besteht also lediglich in der Nutzung bereits vorhandener Definitionen und Ideen zur Umsetzung der Funktionalität auf IPCs.

3.4 Die Schnittstelle zum I/O-Scheduler

Der I/O-Scheduler hat die Aufgabe, die von den Clients eingehenden Aufträge zwischenzuspeichern und gemäß einer bestimmten Zielstellung sortiert an den entsprechenden Bustreiber weiterzureichen. Da die Art der Warteschlange (Liste, Baum und andere) vom verwendeten Scheduling-Algorithmus abhängig ist, muß diese auch komplett vom Scheduler verwaltet werden. Sie kann somit als ein integraler Bestandteil von diesem angesehen werden. Das BDDF braucht sie deshalb auch nicht zur Verfügung stellen.

Der I/O-Scheduler ist keine aktive Komponente, also auch kein eigener Prozeß. Er ist lediglich eine Sammlung von Funktionen, die bei ihrem Aufruf die entsprechenden Sortieroperationen vornehmen. Da diese Funktionen nicht systemnah operieren, sind, selbst wenn sie aus anderen Systemen übernommen wurden, weder Systembibliotheken noch Emulationsumgebungen nötig. Der Scheduler selbst kann also direkt in die BDDF-Instanz einkompiliert oder als Bibliothek angebunden werden. Die Schnittstelle kommt deshalb ohne IPC aus. Sollte dennoch eine weiterführende Kommunikation mit dritten Prozessen erforderlich sein, kann diese weiterhin seitens des Schedulers über IPC erfolgen. Die allgemeinen Einstellungen können über die in Abschnitt 3.3.3 beschriebenen generischen Aufträge erfolgen. Diese müssen dazu an den I/O-Scheduler, das heißt an das entsprechende physische Laufwerk adressiert sein. Mit dieser Methode ist es auch möglich, daß mehrere Scheduler untereinander kommunizieren. So läßt es sich beispielsweise erreichen, eine Laufwerks- und damit Scheduler-übergreifende Bandbreitenbegrenzung für ausgewählte Prozesse durchzusetzen. Denn dazu müssen die Scheduler sich untereinander über deren bisher in Anspruch genommenen Ressourcen informieren.

Nach Abschnitt 3.1.2 gibt es genau einen I/O-Scheduler pro physischem Laufwerk. Die zugehörige Warteschlange muß bei der Anmeldung des Laufwerks im System eingerichtet und bei Abmeldung wieder zerstört werden.

Die Schnittstelle muß Möglichkeiten für die folgenden grundlegenden Aktivitäten schaffen:

- Einstellen eines Auftrages in die Warteschlange
- Entnehmen des nächsten auszuführenden Auftrages aus der Warteschlange
- Einrichten einer Warteschlange für ein physisches Laufwerk
- Zerstören einer Warteschlange.

In den folgenden Abschnitten werden diese Möglichkeiten diskutiert und definiert.

3.4.1 Die Basisfunktionen

Damit die BDDF-Instanz die über die Client-Schnittstelle eingehenden Aufträge der Warteschlange des Zielgerätes zuführen kann, muß der entsprechende I/O-Scheduler eine Funktion `add_request` zur Verfügung stellen, die diese Aufgabe übernimmt. Der Scheduler hat mit dieser Funktion auch die Möglichkeit, die Sortierkriterien neu zu berechnen und die Aufträge in der Warteschlange gegebenenfalls umzusortieren.

Das Entnehmen des nächsten auszuführenden Auftrages aus der Warteschlange erfolgt durch eine Funktion `get_next_request`. Die Funktion darf nicht blockieren, wenn keine Aufträge auszuführen sind. Dies würde im Falle eines Umschaltens des zuständigen I/O-Schedulers während des Blockierens zu einem undefinierten Verhalten führen. Stattdessen muß die BDDF-Instanz dafür sorgen, daß die Funktion nur dann aufgerufen wird, wenn tatsächlich Aufträge zum Weitersenden an den Bustreiber verfügbar sind. Dies erfordert aber auch, daß der BDDF-Instanz vom Scheduler mitgeteilt wird, wann dies der Fall ist.

Die BDDF-Instanz muß bei der Beendigung der Bearbeitung eines Auftrages dessen Urheber über das Ereignis in Kenntnis setzen. Dafür sind die im Auftrag gespeicherten Informationen erforderlich. Deshalb darf der I/O-Scheduler Aufträge nach Aufruf der Funktion `get_next_request` nicht aus dem System entfernen. Stattdessen hat er sie in einer ihm freigestellten Art als *in Bearbeitung befindlich* zu kennzeichnen. Ein erneutes Ausführen von `get_next_request` soll den dann nächsten, nicht in Bearbeitung befindlichen Auftrag liefern. Um den Zustand *in Bearbeitung befindlich* wieder aufheben zu können, muß der I/O-Scheduler über eine Funktion `completed_request` über die Abarbeitung oder einen Fehlerfall des Auftrages informiert werden. Danach kann der Auftrag vom Scheduler aus dem System entfernt werden.

Die eigentliche Einrichtung einer Warteschlange erfolgt, wenn der BDDF-Instanz vom Bustreiber ein neues Laufwerk gemeldet wurde. Dafür muß für den Scheduler eine Funktion `create_queue` implementiert worden sein. Die Warteschlange muß wieder entfernt werden, wenn das zugehörige Laufwerk vom Bustreiber abgemeldet wird. Die zuständige Funktion `destroy_queue` darf die Warteschlange jedoch erst dann tatsächlich zerstören, wenn alle darin noch anstehenden Aufträge abgearbeitet sind. Der Scheduler darf bis dahin allerdings keine weiteren Aufträge annehmen.

3.4.2 Weitere Überlegungen

Die einzige Komponente in einer BDDF-Instanz, die über die Zulassung oder Ablehnung eines Echtzeit-Stromes entscheiden kann, ist der I/O-Scheduler. Denn das Zulassungskriterium ist von seiner Strategie abhängig. Die BDDF-Instanz muß die entsprechenden Anfragen also an den Scheduler weiterreichen. Die Funktionen `reserve_stream`, `start_stream` und `stop_stream` der Client-Schnittstelle (siehe Abschnitt 3.3.2 „Die Behandlung von Echtzeit-Strömen“) gehören also ebenfalls der Scheduler-Schnittstelle an und sind vom Scheduler zu implementieren.

Einige I/O-Scheduling-Algorithmen – insbesondere der in DAS verwendete SATF-Algorithmus (siehe Abschnitt 2.1.2) – benötigen genaue Informationen über die Bearbeitungszeitpunkte eines Auftrages. Sie können diese über eine Zeitnahme bei `get_next_request` und `completed_request` erhalten. Allerdings ist dies durch im System entstehende Verzögerungen ungenau. Um eine präzise Zeitmessung zu erreichen, sollte der Bustreiber die Zeitstempel für die Ereignisse *Auftrag an Laufwerk gesendet* und *Auftragsbearbeitung von Laufwerk beendet* bei der Meldung dessen Abarbeitung mitliefern, soweit ihm das möglich ist. In einer konkreten Implementierung ist hier darauf zu achten, daß Bustreiber und BDDF-Instanz dieselbe Zeitbasis benutzen.

Für einige Scheduling-Strategien ist es nötig, regelmäßige Aufträge zur Kalibrierung des internen Festplattenmodells zu senden oder beim Systemstart eine Reihe von Messungen zur Ermittlung von Festplattenparametern durchzuführen. Deshalb steht es dem I/O-Scheduler frei, selbst Aufträge zu erzeugen und in die Warteschlange einzureihen.

Für I/O-Scheduler, welche nach der Art des Auftrages unterscheiden und optimieren, ist es erforderlich, daß sie eben diese Art auch mitgeteilt bekommen. Ein Beispiel für einen solchen Scheduler ist das in Abschnitt 2.1.4 beschriebene Framework Cello. Da auf der Seite des Clients unendlich viele Auftragsarten denkbar sind, aber die letztlich beachteten Arten vom Scheduler abhängen, kann das BDDF ihre komplette Definition nicht übernehmen. Lediglich für einige häufig anzutreffende Arten ist eine Definition möglich. Weitere Arten sind Scheduler-spezifisch und müssen vom Client durch einen generischen Auftrag erfragt

werden. Es muß demnach eine Möglichkeit geschaffen werden, mit der ein Client die Art des Auftrages mitteilt – ob vordefiniert oder vom Scheduler zusätzlich unterstützt. Es ist dadurch auch möglich, Aufträgen Prioritäten zu vergeben. Dies kann geschehen, indem für jede vorhandene Priorität eine eigene Auftragsart definiert wird.

Für die Verwendung des TCQ-Mechanismus ist es nötig, daß mehrere Aufträge aus der Warteschlange anforderbar sind, bevor der erste abgearbeitet ist. Der Bustreiber kann sich so selbst eine Teilmenge anstehender Aufträge bilden und diese nach eigenen Strategien optimieren. Der I/O-Scheduler hat aber nach wie vor die Möglichkeit, dieses Verhalten zu unterbinden, indem er die Funktion `get_next_request` einfach so lange blockiert, bis der vorhergehende Auftrag abgearbeitet ist.

3.4.3 Weiternutzung von DAS-Scheduling und der I/O-Scheduler aus Linux

Die Basisfunktionen der I/O-Scheduler-Schnittstellen im BDDF und in Linux sind naturgemäß sehr ähnlich und lassen sich somit aufeinander abbilden. Linux unterscheidet sich aber in zwei wesentlichen Punkten vom BDDF. Zum einen wird von Linux eine einfache Warteschlange angeboten, die der Scheduler nutzen kann, falls keine besonderen Anforderungen an diese existieren. Zum anderen werden die Aufträge bei Bearbeitung komplett aus der Warteschlange entfernt und eventuell später wieder eingereiht. Im BDDF verbleiben sie in der Warteschlange und werden lediglich als *in Arbeit befindlich* gekennzeichnet. Es muß für die Weiternutzung der Linux-Scheduler also entweder eine kleine Emulationsumgebung entwickelt werden, oder sie werden an die Schnittstelle im BDDF angepaßt.

Die I/O-Scheduler-Schnittstelle des BDDF ist besonders für die Weiternutzung des in Abschnitt 2.1.2 beschriebenen DAS-Schedulers ausgelegt. Er stellt somit keine besonderen Anforderungen an das BDDF oder die I/O-Scheduler-Schnittstelle.

3.5 Die Schnittstelle zum Bustreiber

Der Bustreiber hat die Aufgabe, die bereits auf absolute Sektorennummern umgerechneten und vom I/O-Scheduler sortieren Aufträge von der BDDF-Instanz anzufordern und ihre Abarbeitung vorzunehmen. Dabei kann für jedes Bussystem ein separater Bustreiber existieren. In kompakten Systemen mit nur einem Bustreiber soll es möglich sein, diesen direkt in die BDDF-Instanz zu integrieren, um Rechenaufwand und Verzögerungszeit zu reduzieren. In Systemen mit einer Vielzahl an Bustreibern müssen diese, auf Grund ihrer voraussichtlich verschiedenen Herkunft und Ansprüche an das Betriebssystem, als jeweils eigener Prozeß mit eventuell eigener Emulationsumgebung realisiert werden. Das erzwingt, daß die Schnittstelle zwischen Bustreiber und BDDF-Instanz auch per IPC realisierbar ist. Der dabei entstehende Geschwindigkeitsnachteil gegenüber der direkten Anbindung muß dadurch

reduziert werden, daß für die zeitkritischen Operationen so wenig wie möglich Kommunikation erforderlich ist. Aus diesem Grunde ist die Bustreiber-Schnittstelle aus L4IDE (siehe Abschnitt 2.4.4) für eine Weiterverwendung nicht geeignet. Sie erfordert viele IPC-Operationen und würde die Leistung einer BDDF-Instanz drastisch verringern.

Damit ein Bustreiber leicht zwischen direkter und IPC-Anbindung umstellbar ist, muß die Schnittstelle so beschaffen sein, daß sie für beide Fälle dieselben Funktionen zur Verfügung stellt. Dabei dürfen allerdings keine Annahmen über die Thread-Struktur des Client getroffen werden, um diesem darüber die volle Entscheidungsgewalt zu geben.

3.5.1 Die Basisfunktionen

Um Aufträge bearbeiten zu können, muß der Bustreiber diese von der BDDF-Instanz anfordern. Außerdem muß er die Instanz über die vollständige Bearbeitung oder einen dabei aufgetretenen Fehler informieren. Für diese beiden grundlegenden Aktionen sind in Abschnitt 3.4.1 schon die Funktionen `get_next_request` und `completed_request` vorgesehen worden. Sie können deshalb einfach übernommen werden.

Damit die BDDF-Instanz Kenntnis über die vom Bustreiber verwalteten physischen Laufwerke hat und die zugehörigen I/O-Scheduler deren Warteschlangen einrichten können, muß der Bustreiber der Instanz seine Laufwerke mittels einer Funktion `register_device` melden. Damit die Instanz einen dem Namensschema (siehe Abschnitt 3.2.4) entsprechenden Namen für das Laufwerk bilden kann, muß der Bustreiber dieser Funktion dessen Anschlußnummer mitliefern. Bei einigen Laufwerkstypen kann es vorkommen, daß sie im laufenden Betrieb aus dem System entfernt werden. Ein Beispiel dafür sind via USB angeschlossene Massenspeicher. Auch dieses Ereignis muß der Instanz vom entsprechenden Bustreiber mitgeteilt werden, damit die zugehörige Warteschlange zerstört werden kann. Dazu dient eine Funktion `unregister_device`, welche solange von der Instanz blockiert wird, bis die noch ausstehenden Aufträge abgearbeitet sind.

Die Partitionsverwaltung der BDDF-Instanz muß für ihre Arbeit die verwendete Sektorgröße des physischen Laufwerkes kennen, denn Partitionen sind immer an Sektoren ausgerichtet. Der Funktion `register_device` ist die Sektorgröße deshalb als Parameter mitzuliefern. Um Zugriffe über das „Ende“ des Laufwerkes hinaus abzulehnen und für informative Zwecke ist die Kenntnis der Gesamtgröße eines Laufwerkes nötig. Diese wird für Partitionen vom entsprechenden Eintrag in der Partitionstabelle geliefert. Für das physische Laufwerk selbst gibt es für diesen Zweck aber nur Bussystem-spezifische Abfragemechanismen – meist über gesonderte Laufwerksbefehle. Da die BDDF-Instanz aber keine Kenntnis über diese Mechanismen besitzt, muß der Bustreiber diese Information der Funktion `register_device` übergeben.

Auch ohne angeschlossene Laufwerke kann es nötig sein, Einstellungen am Bustreiber vorzunehmen – möglicherweise sind seine Laufwerke erst danach verfügbar. Für diese Einstellungen muß er aber im Namensraum sichbar sein. Deshalb hat er sich bei der BDDF-Instanz an- und bei seiner Terminierung auch wieder abzumelden. Dafür sind in der Schnittstelle die Funktionen `register_driver` und `unregister_driver` vorgesehen.

Soll ein Software-RAID-Treiber implementiert werden, muß dieser auf andere Geräte und Bussysteme zugreifen können. Er muß dazu selbst Aufträge erstellen und an die BDDF-Instanz senden. Da ein Bustreiber ohne weiteres zusätzlich als Client registriert sein kann, werden hierzu keine weiteren Funktionen benötigt.

3.5.2 Zur Bearbeitung nötige Informationen

Die zur Bearbeitung eines Auftrages nötigen Informationen über die betroffenen Sektoren und die Schreibrichtung (vom Medium lesend oder auf das Medium schreibend) liegen bei dessen Abholung durch `get_next_request` bereits vor. Die Umrechnung von Blöcken in Sektoren hat die Partitionsverwaltung vorgenommen. An welches seiner Laufwerke der Auftrag gerichtet ist, kann der Bustreiber entweder explizit durch einen mitgelieferten Parameter oder implizit durch gezieltes Anfragen der zum Laufwerk gehörenden Warteschlange erfahren. Ein gezieltes Anfragen ist durch das Blockieren der Funktion `get_next_request` durch die BDDF-Instanz nur dann möglich, wenn zu jedem physischen Laufwerk ein separater Thread gehört. Da die Thread-Struktur aber ein Implementierungsdetail darstellt – bei direkter Anbindung sind hierzu keine Threads nötig –, ist auch die Entscheidung über explizite oder implizite Angabe des Laufwerkes ein solches und kann hier nicht endgültig geklärt werden.

Die unterschiedlichen Arten von Laufwerken haben verschiedene Anforderungen an die zu benutzenden Puffer. Die meisten Laufwerke tauschen die Daten via DMA aus. Für diese muß die physische Adresse eines Speicherbereiches vorliegen. Andere Laufwerke, wie RAM-Disks, kopieren die Daten prozessorgestützt, wofür die logische Adresse des Puffers benötigt wird. Wie in Abschnitt 3.3.3 bereits gezeigt, sind Dataspace-Zeiger für beide Übertragungsarten geeignet und finden deshalb in den Aufträgen des BDDF Verwendung. Wegen der Forderung einer Schnittstelle, die möglichst wenig Kommunikation erfordert, ist es besonders wichtig, daß alle zur Benutzung des Puffers nötigen Daten schon im Auftrag selbst vorliegen. Es dürfen keine weiteren Nachfragen seitens des Bustreibers nötig sein. Aus diesem Grunde müssen auch bei zusammengefaßten Aufträgen sämtliche Teilpufferbeschreibungen in einem Stück übertragen werden. Dies kann beispielsweise in Form einer Scatter/Gather-Liste geschehen.

Kapitel 4 – Implementierung

In diesem Kapitel wird gezeigt, wie die Vorgaben aus dem Entwurfs-Kapitel umgesetzt wurden und welche Besonderheiten dabei zu beachten waren.

4.1 Namensraum und Partitionsverwaltung

Der Namensraum wurde so implementiert, daß für die Benennung der einzelnen Zweige Zeichenketten benutzt werden können. Dadurch sind Gerätenamen wesentlich einprägsamer als bei einem reinen Nummernschema. Sind die üblicherweise vergebenen Namen aber zu lang, zu wenig aussagekräftig oder nicht eindeutig, so hebt dies den Vorteil der besseren Einprägsamkeit wieder auf. Außerdem reduziert sich durch zu lange Gerätenamen der Nutzwert des Schemas. Beispielsweise lautet ein üblicher Gerätename im BDDF *ide.via.m1.2*. Dasselbe Gerät heißt in Linux *hda2*. Schon in diesem einfachen Fall ist für einen Benutzer des BDDF mehr Schreibarbeit nötig. Partitionsnamen werden nur von einigen Partitionstypen – zum Beispiel LDM – erlaubt. Diese konnten mangels Testmöglichkeiten noch nicht auf ihre typische Namensnutzung untersucht werden. Aus diesem Grunde werden den Partitionen seitens der implementierten Partitionsverwaltung weiterhin nur Nummern zugeordnet.

Im Namensraum registrierte Geräte müssen sehr oft nach ihrem Namen gesucht werden – so bei der Registrierung neuer Geräte oder bei jedem Öffnungsversuch seitens eines Clients. Deshalb ist es wichtig, daß sie schnell gefunden werden können. Dazu wurde allen Geräten ein sogenannter Hash-Wert zugeordnet. Dieser wird aus dem Namen berechnet und variiert schon bei kleinen Namensänderungen. Außerdem wurde eine Tabelle eingerichtet, in der sämtliche im System registrierten Geräte nach ihrem Hash-Wert geordnet eingetragen sind. Für eine Suchoperation ist damit im Normalfall lediglich ein Vergleich zweier Hash-Werte, anstatt zweier Zeichenketten nötig. Sollten zwei Gerätenamen den gleichen Hash-Wert liefern, ist dennoch ein Zeichenketten-Vergleich durchzuführen.

Die Programmteile, welche ein Laufwerk nach vorhandenen Partitionen durchsuchen und diese dann dem System melden, wurden aus L4IDE – und damit indirekt aus Linux 2.6 – übernommen. Da Linux erweiterte Partitionen aber nur durchnummeriert, waren hier Anpassungen an das komplexere Namensschema des BDDF nötig. Insbesondere gilt das für

die MSDOS-Partitionstabellen, da die Partitionen hier als verkettete Liste mit mehreren Elementen organisiert sind. Eine erweiterte Partition dient dabei lediglich als Container für weitere Partitionen und wird unter Linux deshalb nicht registriert. Im BDDF sind sie aber dennoch als Elternelemente erforderlich und dürfen nicht unterschlagen werden. Als unangenehmer Nebeneffekt führt das dazu, daß im BDDF mehr Geräte angezeigt werden als in Linux. Die zusätzlichen Geräte sind aber nicht für den Einsatz als Datenspeicher geeignet, sondern erfüllen lediglich organisatorische Zwecke.

Der Baum des Namensraumes ist potentiell sehr komplex. Insbesondere beim SCSI-Bussystem gibt es sehr viele Anschlüsse für physische Laufwerke. Außerdem sind die Anzahlen der Ebenen für Untergeräte und Partitionen unbestimmt und können somit ebenfalls sehr groß sein. Die Komplexität des Baumes wächst mit diesen Anzahlen *n* der Ebenen für Untergeräte und Partitionen exponentiell. Die Suche nach existierenden Partitionen kann somit sehr viel Zeit in Anspruch nehmen. Um dieses Problem zu umgehen, wurde eine Funktion `bddf_get_next_physical_device` implementiert, die nacheinander die Namen aller physischen Laufwerke liefert. Die Funktion `bddf_get_next_logical_device` hingegen liefert die Namen aller Partitionen innerhalb eines bestimmten physischen Laufwerkes. Die Ordnung dieser beiden Funktionen ist nur noch *O(n)*.

Es erfolgt mit `bddf_get_next_physical_device` eine Iteration über alle vorhandenen Geräte in der Ebene der physischen Laufwerke des Namensraumes. Für eine komplette Suche kann `NULL` als Startelement übergeben und die Funktion so lange ausgeführt werden, bis erneut `NULL` als Name des nächsten Laufwerkes geliefert wird. Die Funktionsweise von `bddf_get_next_logical_device` ist ähnlich. Der Unterschied besteht darin, daß als Startelement der Name eines physischen Gerätes verwendet wird und die Funktion die Namen jedes Knotens der untergeordneten Zweige liefert.

Abbildung 10 verdeutlicht das Verhalten der beiden Funktionen. Mit `NULL` als Startparameter liefert `bddf_get_next_physical_device` (grüne, gestrichelte Pfeile) die Namen *ide.via.m1*, *ide.via.m2*, *ide.via.s2* und danach wieder `NULL`. Dabei wurde im Beispiel davon ausgegangen, daß lediglich am IDE-Bus-Controller *via* Laufwerke angeschlossen sind. Ausgehend von *ide.via.m2* liefert `bddf_get_next_logical_device` (rote, gepunktete Pfeile) hingegen die Namen *ide.via.m2.1*, *ide.via.m2.2*, *ide.via.m2.2.1*, *ide.via.m2.2.2*, *ide.via.m2.3* und *ide.via.m2.4*. Bei der nächsten Ausführung würde `NULL` geliefert.

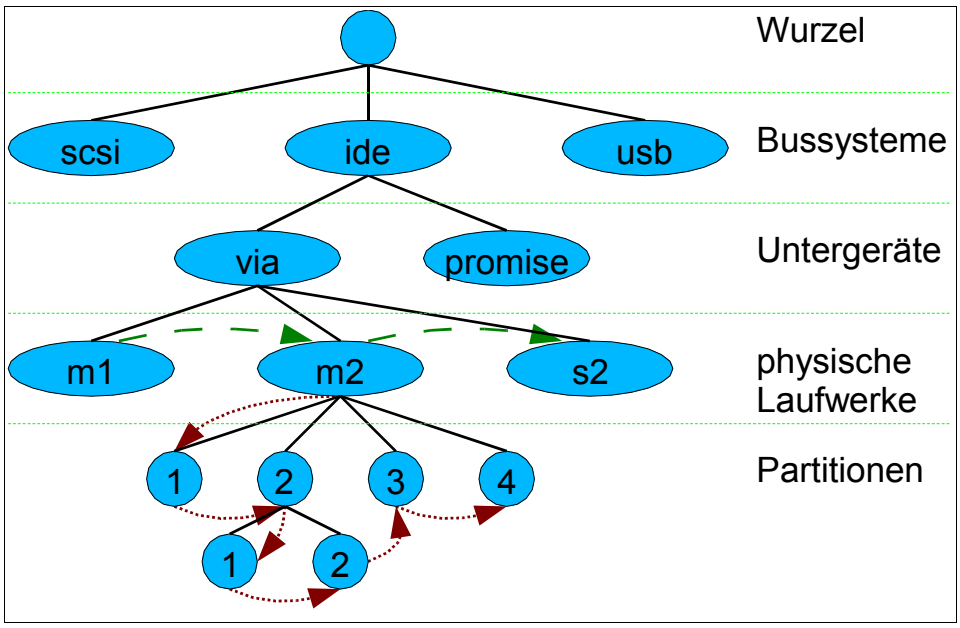

Abbildung 10: *Baum des Namensraumes mit Suchwegen*

4.2 Schnittstelle zum Client

Da die Clients einer BDDF-Instanz fast ausnahmslos eigenständige Prozesse sein werden, muß die Kommunikation mit ihnen über IPC stattfinden. DROPS stellt zur Erstellung IPC-basierter Schnittstellen das Werkzeug DICE [DIC] zur Verfügung, welches auch hier benutzt wurde. Dabei sind die tatsächlichen IPC-Funktionen durch eine Funktionsbibliothek für den Client gekapselt. Ein Client braucht dadurch auch keine Kenntnis über die verwendeten Threads besitzen. Viele Ansätze und Ideen zur Verwaltung von Strukturen, Handles und die Umsetzung der Client-Schnittstelle auf IPC wurden von generic_blk übernommen.

Da die BDDF-Instanz selbst nicht über die Zulassung oder das Starten von Echtzeit-Strömen entscheiden kann, übernehmen die zugehörigen Funktionen lediglich die Überprüfung der Zugriffsberechtigung des anfragenden Clients. Zur weiteren Bearbeitung werden die entsprechenden Funktionen der I/O-Scheduler-Schnittstelle aufgerufen.

4.2.1 Die Thread-Struktur

Bevor ein Client überhaupt mit einer BDDF-Instanz kommunizieren kann, muß er zunächst ihre Thread-ID in Erfahrung bringen. Die Funktion `bddf_open_driver` erwartet dazu die Kennung der gewünschten Instanz und fragt beim Namensdienst nach dem zugehörigen Thread. Die Instanz erzeugt bei der folgenden Registrierung alle weiteren Threads und gibt deren IDs und eine Kennung an die Schnittstelle zurück. Mit `bddf_close_driver` wird diese Kennung wieder für ungültig erklärt. Durch die Reservierung von Kennungen ist

sichergestellt, daß die BDDF-Instanz auch so lange existent ist, wie ein Client mit ihr kommunizieren möchte.

Damit sich die Clients beim Senden von Aufträgen nicht gegenseitig blockieren können, wird für jeden Client ein eigener Kommando-Thread eingerichtet. Außerdem wird clientseitig ein Benachrichtigungs-Thread benutzt, damit für das Erfragen des aktuellen Auftragsstatus keine zeitintensiven IPC-Operationen auszuführen sind, sondern diese nur bei einer tatsächlichen Statusänderung anfallen. Ein separater serverseitiger Thread stellt die Benachrichtigungen zu, welche direkt zu einer Änderung des Status in der Auftrags-Struktur führen. Dadurch entfallen die in Abschnitt 3.3.1 vorgesehenen Funktionen zur Abfrage von Status und Fehlernummer. Der Kommando- und der Benachrichtigungs-Thread werden beim Anmelden des Clients an der BDDF-Instanz erzeugt. Generische Aufträge treten im Verhältnis zu regulären Aufträgen selten auf und sind wenig zeitkritisch. Außerdem müssen sie auch ohne bereits vorhandenen Kommando-Thread sendbar sein. Aus diesen Gründen existiert serverseitig für jeden Client genau ein Thread für generische Aufträge.

Ein Client soll Aufträge sowohl synchron als auch asynchron senden können. Damit das Senden von Aufträgen aus anderen Threads des Clients nicht durch das Warten auf die Fertigstellung eines synchronen Auftrages blockiert wird, werden synchrone Aufträge intern ebenfalls asynchron gesendet. Die Synchronisierung erfolgt in diesem Fall durch ein Blockieren der Sendefunktion auf die entsprechende Benachrichtigung.

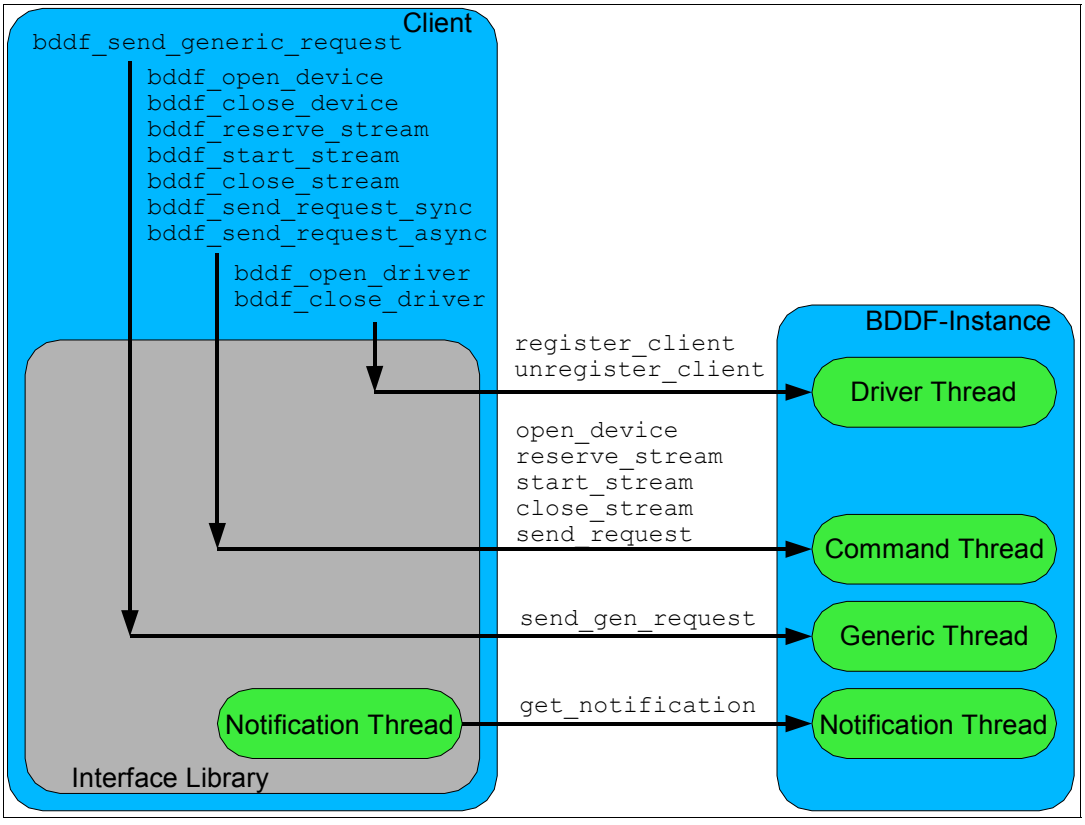

Abbildung 11: *Threads und Funktionen der Client-Schnittstelle*

Abbildung 11 zeigt die Funktionen, die dem Client angeboten werden und ihre Schnittstellen-internen Pendants. Außerdem ist durch Pfeile dargestellt, an welche Threads die Funktionen jeweils gerichtet sind.

4.2.2 Darstellung regulärer und generischer Aufträge

Die Pufferdarstellung erfolgt im gesamten BDDF durch die Struktur `ds_list_element`. Diese enthält die folgenden Informationen:

```
struct ds_list_element {
    l4dm_dataspace_t ds_id;       /* Dataspace, der den Puffer bildet */
    l4_offs_t        ds_offset;   /* Offset des Puffers im Dataspace */
    l4_size_t        ds_size;     /* Größe des Puffers */
    phys_addr_t      phys_addr;   /* physische Adresse des Puffers */
};
```

Ein Auftrag wird vom Client in Form der Struktur `client_bio_t` dargestellt, welche wie folgt definiert ist:

```
typedef struct client_bio {
    int              type;        /* zeigt Schreib- oder Lese-Auftrag an */
    block_t          start_block; /* erster zu übertragender Block */
    int              block_num;   /* Anzahl der zu übertragenden Blocks */
    Time_t           deadline;    /* spätester Bearbeitungszeitpunkt */
    int              tag;         /* Scheduler-spezifische Auftragsart */
    device_handle_t  device;      /* Zielgerät des Auftrages */
    stream_handle_t  stream;      /* Echtzeit-Strom, zu dem der Auftrag
                                     eventuell gehört */
    struct ds_list_element * ds_list;
                                  /* Scatter/Gather-Liste von Puffer-
                                     beschreibungen */
    int              element_num; /* Anzahl der Elemente in ds_list */
} client_bio_t;
```

Der Parameter `deadline` ist dabei ein Wunsch des Clients und keine Forderung. Er wurde aufgenommen, damit auch sporadisch auftretende und damit keinem Strom zuordenbare Echtzeit-Aufträge darstellbar sind. Da ihnen aber keine Reservierung vorausging, kann für ihre rechtzeitige Abarbeitung nicht garantiert werden. Es obliegt dem Scheduler, diese Aufträge weitestgehend vor Ablauf ihrer Zeitschranke auszuführen. Der Parameter `tag` ist die Darstellung der Auftragsart nach Abschnitt 3.4.2. Die Angabe des Puffers erfolgt durch eine Scatter/Gather-Liste von Elementen der Struktur `ds_list_element`. Die weiteren nötigen Informationen sind in der Struktur `client_request_t` gespeichert. Diese Struktur wird dabei nicht an die BDDF-Instanz gesendet, sondern dient lediglich der clientseitigen Auftragsverwaltung. Ihre Definition lautet:

```
typedef struct client_request {
    driver_handle_t driver;      /* die BDDF-Instanz, an die der Auftrag
                                    gesendet werden soll */
    done_fn *       done_func;   /* die Funktion, die bei der Auftrags-
                                    Abarbeitung aufgerufen werden soll */
    client_bio_t *  bio;         /* die BIO dieses Auftrages */
    void *          client_priv; /* vom Client frei verwendbare
                                    Information */
    l4semaphore_t   done_sem;    /* zeigt die Auftrags-Abarbeitung an */
    req_handle_t    req_handle;  /* Kennung des Auftrages in der Instanz */
    int             status;      /* aktueller Status des Auftrages */
    int             error;       /* Fehlernummer, wenn der Status einen
                                    Fehler anzeigt */
} client_request_t;
```

Das Semaphor `done_sem` wird von der Client-Schnittstelle für das synchrone Senden des Auftrages verwendet. Im asynchronen Fall kann es vom Client frei verwendet werden. Generell frei verwendet werden dürfen `done_func` und `client_priv`.

Im Gegensatz zu regulären Aufträgen existiert keine Struktur für generische Aufträge. Ihre Parameter werden als Argumente an die entsprechenden Funktionen übergeben. Diese Parameter sind:

`device_name`	Name des angesprochenen Gerätes
`descriptor`	Kurzbeschreibung des Auftrages
`in_value`	zu übermittelnder Wert
`in_buffer_type`	Typ des Sendepuffers
`in_buffer_size`	Größe des Sendepuffers
`in_buffer`	der Sendepuffer
`out_buffer_type`	Typ des Empfangspuffers
`out_buffer_size`	Größe des Empfangspuffers
`out_buffer`	der Empfangspuffer
`return_value`	vom Gerät zurückgegebener Wert

Als Puffertyp ist bereits `BDDF_GENERIC_TYPE_STRING` definiert. Wird ein Puffer nicht verwendet, so ist der Typ `BDDF_GENERIC_TYPE_UNUSED` zu wählen. Die Verwendung des Gerätenamens bietet den Vorteil, daß keine Kennung des Gerätes angefordert werden muß. Dies ist in einigen Fällen nämlich unnötig oder gar unmöglich, weil kein entsprechendes Gerät existiert.

Für einige unbedingt nötige oder häufig benutzte generische Aufträge sind auf Basis der allgemeinen Funktion `bddf_send_generic_request` bereits spezielle Funktionen implementiert. Diese sind:

bddf_get_sector_size	liefert die Größe der Sektoren des angegebenen Gerätes
bddf_get_device_size	liefert die Kapazität des angegebenen Gerätes
bddf_get_next_physical_device	liefert das auf das angegebene Gerät folgende physische Gerät (siehe Abschnitt 4.1)
bddf_get_next_logical_device	liefert das auf das angegebene Gerät folgende logische Gerät (siehe Abschnitt 4.1)
bddf_get_available_schedulers	liefert eine Liste aller in der Instanz registrierten Scheduler
bddf_get_scheduler	liefert den Namen des momentan für das angegebene Gerät zuständigen Schedulers
bddf_set_scheduler	ändert den für das angegebene Gerät zuständigen Scheduler

4.3 Schnittstelle zum I/O-Scheduler

Die I/O-Scheduler-Schnittstelle ist ähnlich der in Linux 2.6 aufgebaut. Jeder I/O-Scheduler registriert sich mittels der Funktion `scheduler_add` bei der BDDF-Instanz. Dabei übergibt er eine für ihn gültig initialisierte Struktur `scheduler`. Sie enthält die folgenden Elemente:

name	der Name des Schedulers
scheduler_data	frei vom Scheduler verwendbar
scheduler_init_fn	Funktion zum Initialisieren des Schedulers
scheduler_exit_fn	Funktion zum Beenden des Schedulers
scheduler_gen_req_fn	Funktion zum Senden eines generischen Auftrages
scheduler_create_queue_fn	Funktion zum Erzeugen einer Warteschlange
scheduler_destroy_queue_fn	Funktion zum Zerstören einer Warteschlange
scheduler_add_bio_fn	Funktion zum Einstellen einer BIO in die Warteschlange
scheduler_get_next_req_fn	Funktion zum Erfragen des nächsten Auftrages
scheduler_completed_req_fn	Funktion zum Erklären der Auftrags-Abarbeitung
scheduler_reserve_stream_fn	Funktion zum Reservieren eines Echtzeit-Stromes
scheduler_start_stream_fn	Funktion zum Starten eines Echtzeit-Stromes
scheduler_close_stream_fn	Funktion zum Beenden eines Echtzeit-Stromes

Die Funktionen `scheduler_init_fn` und `scheduler_exit_fn` sind optional. Die letzten drei Funktionen sind genau dann erforderlich, wenn der Scheduler Echtzeit-Ströme unterstützt. Sind hier keine Funktionen angegeben, werden Anfragen nach dem Reservieren eines Echtzeit-Stromes von der BDDF-Instanz abgelehnt. Von der Funktion `scheduler_reserve_stream` wird eine Struktur `bddf_stream` als Argument gefordert. Diese wurde von der BDDF-Instanz bereits erzeugt und mit den Parametern des Stromes initialisiert. In ihr kann der Scheduler über den Zeiger `sched_priv` ebenfalls spezifische Daten ablegen.

Da die Warteschlangen im BDDF komplett vom entsprechenden I/O-Scheduler verwaltet werden, kann auch nur er über ein Zusammenfassen von Aufträgen entscheiden. Insbesondere bei Echtzeit-Aufträgen ist eine solche Entscheidung wegen eventueller Deadline-Überschreitungen nicht trivial. Das BDDF stellt ihm allerdings Hilfsfunktionen zur Verfügung. Zum einen ist das die Funktion `scheduler_merge`, die sonst jeder Scheduler separat definieren müßte. Sie führt den eigentlichen Zusammenfassungs-Vorgang aus und wird vom Scheduler aufgerufen, wenn dieser ein Zusammenfassen zweier Aufträge für möglich hält. Zum anderen ist das die Funktion `scheduler_is_mergeable`, welche eine allgemeine Überprüfung der Aufträge hinsichtlich ihrer Zusammenfaßbarkeit vornimmt. Dabei wird untersucht, ob sie auf die gleiche Art (lesend oder schreibend) auf benachbarte Blöcke zugreifen. Die Funktion teilt außerdem mit, ob ein Auftrag vor oder nach dem anderen einfügbar ist. Eventuell vorhandene weitere Kriterien muß der Scheduler dann selbst prüfen.

Im Entwurf wurde festgestellt, daß zum Berechnen der genauen Auftrags-Bearbeitungszeiten eine einheitliche Zeitbasis nötig ist. Zu diesem Zweck wurde die Bibliothek `bddf_timebase` geschaffen. Diese bietet die Funktion `bddf_get_timestamp`, welche den Time Stamp Counter (TSC) des Prozessors ausliest und dessen Wert zurückliefert. Die Funktion `bddf_get_time_difference` berechnet die Zeit, die zwischen zwei so ermittelten Zeitpunkten vergangen ist. Die Differenz wird dabei in Nanosekunden angegeben. Die Funktionen sind so gewählt, daß in späteren Implementationen auf eine andere Quelle für Zeitstempel umgestellt werden kann. Dies ist beispielsweise auf Systemen der Fall, die keinen TSC besitzen.

4.4 Schnittstelle zum Bustreiber

Die Bustreiber-Schnittstelle wurde in zwei Versionen implementiert. Die eine verwendet eine interne Schnittstelle per IPC und die andere ist direkt an die BDDF-Instanz angebunden. Einem Bustreiber werden in beiden Versionen dieselben Funktionen zur Verfügung gestellt. Um zwischen den Anbindungen zu wechseln, ist lediglich das Einbinden eines anderen Header-Files in den Bustreiber und das Ändern der Datei `Makefile` nötig.

Da die exakte Reproduktion der BDDF-internen Strukturen im Bustreiber per IPC zu viel Prozessorzeit erfordert hatte, wurde bei der Implementation eine andere Vorgehensweise gewählt. Für den Bustreiber wurden eigene Strukturen erstellt, die nur die wesentlichen Informationen enthalten und leicht per IPC zu übertragen sind. Da die an den Bustreiber gesendeten Aufträge durch vorheriges Zusammenfassen mehrere Teilaufträge und damit auch mehrere Pufferbeschreibungen enthalten können, werden diese Beschreibungen in Form einer Liste übertragen. Die Elemente dieser Liste sind vom Typ `ds_list_element`, welche bereits in Abschnitt 4.2.2 dargestellt ist.

Ein Puffer kann damit als Teilbereich in einem Dataspace mit eventuell schon bekannter physischer Adresse angegeben werden. Die weiteren Informationen sind in der folgend dargestellten Struktur `bus_request_t` zusammengefaßt.

```
typedef struct bus_request {
    int      type;          /* zeigt Schreib- oder Lese-Auftrag an */
    Sector_t start_sect;    /* erster zu übertragender Sektor */
    int      sect_num;      /* Anzahl zu übertragender Sektoren */
    req_id_t req_id;        /* interne Kennung des Auftrages */
    int      ds_list_count; /* Anzahl der Pufferbeschreibungen */
    void *   ds_list;       /* Liste der Pufferbeschreibungen */
    void *   bus_priv;      /* vom Bustreiber frei verwendbar */
    Time_t   start_time;    /* Startzeitpunkt der Auftragsabarbeitung */
    Time_t   end_time;      /* Endzeitpunkt der Auftragsabarbeitung */
} bus_request_t;
```

Ein Bustreiber muß sich mit der Funktion `bus_register_busdriver` bei der BDDF-Instanz unter Angabe seines Namens registrieren. Dabei wird im Namesraum ein entsprechender Knoten angelegt und dessen Kennung dem Bustreiber zurückgegeben. Zur besseren Übersicht kann der Bustreiber mit `bus_register_sub_device` Untergeräte registrieren, welche allerdings keine weitere Funktion besitzen. Dabei muß neben dem Namen auch die Kennung des jeweiligen Vaterknotens angegeben werden. Zu jeder Registrierfunktion existiert auch eine entsprechende Funktion, um das Gerät wieder abzumelden.

Die vom Bustreiber verwalteten physischen Geräte können mit der Funktion `bus_register_phys_device` bei der BDDF-Instanz angemeldet werden. Dabei muß die Kennung des Vaterknotens sowie der Name, die native Sektorgröße und die Sektoren-Anzahl des Gerätes angegeben werden. Außerdem ist die Angabe der ID des Threads nötig, der letztlich die Auftragsbearbeitung vornimmt. Denn mit diesem muß ein Client den Dataspace seines Auftragspuffers teilen (mittels `l4dm_share`). Ebenfalls erforderlich ist das Bereitstellen einer Funktion, welche während des Anmeldevorgangs aufgerufen wird. Diese muß das Gerät auf die Auftragsabarbeitung vorbereiten. Nach ihrem Aufruf muß das Gerät bereits auf Aufträge von der BDDF-Instanz warten. Dies ist nötig, weil es zur Vollendung der Registrierung nach existierenden Partitionen durchsucht wird.

Der Bustreiber kann den nächsten auszuführenden Auftrag mit der Funktion `bus_get_next_req` ermitteln. Diese blockiert so lange, bis ein solcher Auftrag verfügbar ist. Mittels `bus_completed_req` teilt der Bustreiber die Abarbeitung des Auftrages mit. Dabei gibt er auch eine Fehlernummer an, welche der BDDF-Instanz übermittelt wird.

In der Schnittstellen-Version mit direkter Anbindung werden keine zusätzlichen Threads erzeugt. Alle Aktivitätsträger der Schnittstelle sind die des Bustreibers selbst. Lediglich in der IPC-Version werden zum Start der BDDF-Instanz der Kommando-Thread und beim Registrieren eines physischen Laufwerkes je zwei weitere Threads erzeugt. Der eine wartet auf den nächsten auszuführenden Auftrag und der andere auf die Benachrichtigungen über die beendete Bearbeitung eines Auftrages. Dadurch ist eine von anderen Geräten, insbesondere anderer Bustreiber, unabhängige Auftragsabarbeitung möglich. Seitens der Schnittstelle bestehen keine Forderungen nach einer bestimmten Thread-Struktur des Bustreibers. Außer der, daß das Warten auf neue Aufträge in einem anderen Thread stattfindet als dem, der die Funktion `bus_register_phys_device` aufruft. Andernfalls tritt ein Blockieren des Systems ein.

4.5 Weitere Details

In der BDDF-Instanz wird eine bestimmte Anzahl Pufferbeschreibungen des Typs `ds_list_element` in einer Liste vorallokiert. Diese Liste ist als sogenannter Slab-Cache organisiert. Dadurch ist eine schnelle Reservierung von Pufferbeschreibungen möglich. Diese fällt jedesmal bei der Erstellung einer BIO an. Durch die ohne diese Liste erforderliche Nutzung der rechenintensiven Funktionen `malloc` und `free` würde die Auftragsbearbeitung erheblich verlangsamt.

Die Aufträge sind BDDF-intern ähnlich denen in Linux 2.6 aufgebaut. Die vom Client gesendeten Aufträge werden in eine Struktur `bddf_bio` übertragen – die BIO. Dabei werden die Angaben des Startblocks und der Blockanzahl von der Partitionsverwaltung in die Angaben Startsektor und Anzahl der Sektoren gewandelt. Die Angabe des Puffers erfolgt in Form einer Scatter/Gather-Liste von Pufferbeschreibungen. Eine BIO ist folgendermaßen definiert:

```
struct bddf_bio {
    /* Basis-Informationen */
    Sector_t             start_sect;    /* erster zu übertragender
                                           Sektor */
    int                  sect_num;      /* Anzahl zu übertragender
                                           Sektoren */
    struct ds_list_element * ds_list;   /* Scatter/Gather-Liste von
                                           Pufferbeschreibungen */
    int                  ds_list_count; /* Anzahl der enthaltenen
                                           Pufferbeschreibungen */
    Time_t               deadline;      /* Zeitschranke zur
                                           Auftragsabarbeitung */
    int                  tag;           /* Auftragstyp */
    struct bddf_device * device;        /* Zielgerät des Auftrages */
    struct bddf_stream * stream;        /* Echtzeit-Strom, zu dem der
                                           Auftrag gehört */
    /* Informationen zum BIO-Management */
    l4semaphore_t        completed_sem; /* zeigt die Auftragsabarbeitung
                                           an */
    int                  status;        /* Status des Auftrages */
    int                  error;         /* Fehlernummer, falls der Status
                                           einen Fehler anzeigt */
    struct bddf_bio *    next_bio;      /* nächste BIO dieses Auftrages */
    void *               sched_priv;    /* vom Scheduler frei verwendbar */
    req_handle_t         req_id;        /* interne Kennung des Auftrages */
    struct notify_thread_info * notify_info;
                                        /* nötige Informationen über den
                                           zu informierenden Client */
};
```

Der I/O-Scheduler kann mehrere BIOs zu einem Auftrag zusammenfassen. Dafür wird ein Container generiert, der ein oder mehrere BIOs aufnehmen kann. Dieser Container ist die Struktur `bddf_request`, die im folgenden dargestellt ist.

```
struct bddf_request {
    int               type;       /* zeigt Schreib- oder Leseauftrag an */
    struct bddf_bio * first_bio;  /* erste BIO dieses Auftrages */
    struct bddf_bio * last_bio;   /* letzte BIO dieses Auftrages */
    int               bio_count;  /* Anzahl der enthaltenen BIOs */
    Time_t            start_time; /* Zeitstempel des Starts der
                                     Auftragsbearbeitung */
    Time_t            end_time;   /* Zeitstempel des Endes der
                                     Auftragsbearbeitung */
    void *            sched_priv; /* frei vom Scheduler verwendbar */
};
```

Die BIOs eines Auftrages sind untereinander in der Reihenfolge der Sektoren auf die sie zugreifen verkettet. Der Auftrag enthält einen Verweis auf die erste enthaltene BIO und einen auf die letzte. Damit wird beim Zusammenfassen ein schnelles Anhängen einer weiteren BIO am Anfang oder am Ende ermöglicht.

Für die Ermittlung der auf einem physischen Laufwerk vorhandenen Partitionen wurde derselbe Quellcode verwendet wie in Linux 2.6. Aus diesem Grunde werden auch dieselben Partitionstypen unterstützt. Eine Ausnahme bilden lediglich die Typen *NEC98* und *IBM*, da diese auf Geometrie-Informationen zurückgreifen, die in DROPS nicht verfügbar sind. Der

Partitionstyp *LDM* wird noch nicht unterstützt, weil dafür sehr viel Anpassungsaufwand erforderlich ist.

In [Men03] war L4IDE als ein Zwischenschritt zum BDDF konzipiert. Um L4IDE abzulösen, muß das BDDF einen IDE-Bustreiber besitzen. Deshalb wurde der IDE-Kern aus L4IDE herausgelöst und ohne den Linux-Blocktreiber als über IPC angebundener Bustreiber für das BDDF implementiert. Um die gute Wartbarkeit von L4IDE zu erhalten, wurde auf eine Anpassung des IDE-Kerns an die neue Bustreiber-Schnittstelle verzichtet und stattdessen eine Emulation der nötigen Funktionen des Linux-Blocktreibers vorgenommen. Die allgemeine Linux-Kernel-Umgebung wird nach wie vor durch das DDE2.6 zur Verfügung gestellt.

Im BDDF soll ein Bustreiber den Start- und Endzeitpunkt der Bearbeitung des Auftrages vom entsprechenden Laufwerk liefern. Der entstandene IDE-Treiber liefert als Startzeitpunkt den Moment des Sendens des Kommandos an das Laufwerk und als Endzeitpunkt den Moment des Aufrufens der für die Behandlung des Unterbrechungsereignisses zuständigen Funktion. Da dieses Ereignis von L4 durch eine IPC-Operation zugestellt und in DDE2.6 noch eine Vorbehandlung erfährt, tritt hierbei eine gewisse Verzögerung auf. Die Konstanz der in Kapitel 5 ermittelten Werte legt aber nahe, daß diese Verzögerung sehr gering oder sehr konstant ist. Sie kann damit als zur Bearbeitungszeit gehörig angesehen werden.

Um die entstandene Schnittstelle zum I/O-Scheduler zu testen, ist ein NOOP-Scheduler implementiert worden. Dieser sorgt lediglich für ein Zwischenspeichern und Zusammenfassen der Aufträge. Eine Umsortierung findet nicht statt. Die Aufträge werden dadurch in ihrer Ankunftsreihenfolge an den entsprechenden Bustreiber gereicht. Außerdem ist ein Elevator-Scheduler entstanden, welcher versucht, die nötigen Kopfbewegungen der Festplatte zu reduzieren. Dabei gibt er immer den Auftrag an den Bustreiber, der auf Sektoren zugreift, die von bestimmten anstehenden Aufträgen das Minimum darstellen. Dabei werden die Aufträge in Betracht gezogen, deren Startsektornummer größer oder gleich der des aktuell unter dem Schreib-/Lesekopf befindlichen ist. Gibt es keinen solchen Auftrag, wird der an den Bustreiber gegeben, der von allen Aufträgen die kleinste Startsektornummer besitzt. Dadurch tritt eine Kopfrückbewegung ein. Weiterhin beachtet der Elevator-Scheduler die Zeitschranke von Aufträgen. Er weicht von der bevorzugten Bewegungsrichtung des Schreib-/Lesekopfes ab, wenn zur Zeitschranke eines Auftrages noch weniger als zweimal die Worst-Case-Ausführungszeit übrig ist. Dieser Auftrag wird dann als nächster ausgeführt. Die Worst-Case-Auftragszeit wird auf das Maximum aller jemals gemessenen Ausführungszeiten gesetzt. Sie ist somit ein dynamischer Wert.

Die Blockgeräteverwaltung des BDDF enthält keine Threads und ist so nicht selbst aktiv. Alle Aktionen werden durch die Threads der Schnittstellen ausgeführt. Die Thread-Struktur des BDDF ist damit identisch zu der ihrer Schnittstellen.

Kapitel 5 – Leistungsbewertung

Um die Leistung des entstandenen Frameworks beurteilen zu können, ist es nötig, es mit anderen entsprechenden Projekten zu vergleichen. Eine geeignete Vorarbeit dazu liefert [Men03] in Kapitel 5. Darin wurden der IDE-Treiber aus Linux 2.6.0-9 und L4IDE hinsichtlich der Zeit untersucht, die sie zur Bearbeitung von 10000 Aufträgen zum Lesen derselben Sektoren benötigen. Die Auftragsgröße von 32kB war so gewählt worden, daß die Aufträge komplett im Cache der verwendeten Festplatten Platz fanden. So konnten Einflüsse durch Positionierzeiten des Schreib-/Lesekopfes ausgeschlossen werden. Die beiden verwendeten Testsysteme stehen für den Test des BDDF immer noch zur Verfügung. Die gemessenen Zeiten können somit wiederverwendet werden. Die 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)

Durch die Messungen werden eventuell die Stellen lokalisiert, an denen die Auftragsbearbeitung am stärksten verzögert wird. Diese können später einer Optimierung unterzogen werden. Dazu ist es aber nötig, die Verweilzeit eines Auftrages an den einzelnen Stationen zu messen. Wie in [Men03] wurde zur präzisen Zeitnahme der Time Stamp Counter (TSC) des Prozessors verwendet. Die genaue Bearbeitungszeit eines Auftrages im physischen Laufwerk wird im BDDF generell schon vom Bustreiber geliefert. Für das Auslesen der anderen intern ermittelten Werte wurden generische Aufträge verwendet.

Es wurden fünf Meßreihen aufgenommen und ihr arithmetisches Mittel errechnet. Die Schwankungsbreite bewegte sich lediglich im einstelligen Promille-Bereich. Aus den Differenzen der gemessenen Zeiten wurden die in Abbildung 12 dargestellten Werte ermittelt. Diese sind die Zeit der physischen Auftragsbearbeitung (Verweilzeit im Laufwerk), die Verweilzeit im IDE-Treiber, die Verweilzeit in der Bustreiber-Schnittstelle, die Verweilzeit im

BDDF und die Verweilzeit in der Client-Schnittstelle. In Abbildung 12 ist außerdem noch angegeben, wieviel tausend Takte die Bearbeitung eines Auftrages an den einzelnen Stationen erfordert hat.

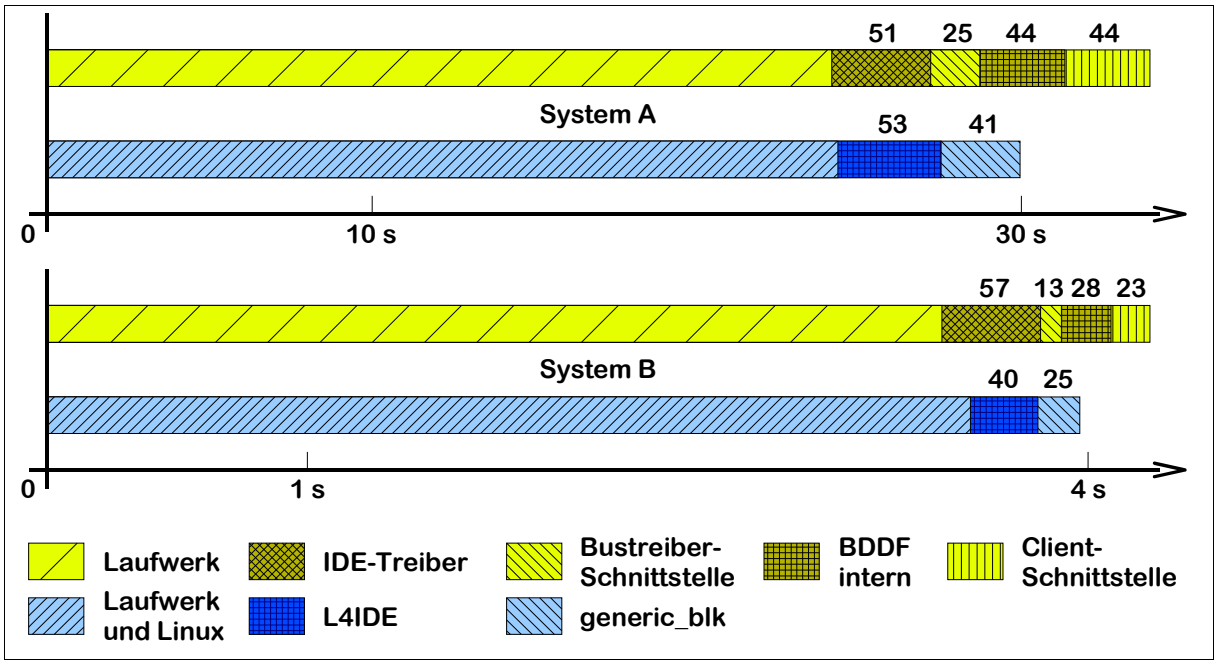

Abbildung 12: *Bearbeitungszeit eines Auftrages an den Stationen seiner Verzögerung*

Beim Vergleich der Verweilzeit im Laufwerk und den Werten für Linux wird deutlich, daß Linux sehr effizient arbeitet. Die Differenz zu L4IDE ohne generic_blk und dem IDE-Treiber des BDDF läßt sich nur durch die Verwendung des DDE2.6 erklären, da ansonsten jeweils derselbe Quellcode verwendet wurde.

Der zwischen den Meßpunkten der Bustreiber-Schnittstelle befindliche Quellcode (inklusive dem von DICE generierten) kopiert lediglich wenige hundert Byte an Daten und führt einige Vergleiche durch. Die Verzögerung von 25.000 beziehungsweise 13.000 Takten kann also nur durch das Transportieren der Daten via IPC und die damit verbundenen Umschaltungen des aktiven Threads entstanden sein. Es zeigt sich auch, daß generic_blk und die Client-Schnittstelle des BDDF ähnlich schnell arbeiten. Auch hier werden lediglich einige hundert Byte kopiert und die Daten via IPC transportiert. Allerdings enthält die Client-Schnittstelle viele Multiplikationen, welche relativ viele Takte kosten. Außerdem werden zur Allokation und Freigabe der BIOs die Funktionen `malloc` und `free` verwendet, welche ebenfalls viel Zeit erfordern. Der Mehraufwand bei der Auftrags-Übermittlung der Client-Schnittstelle gegenüber der Bustreiber-Schnittstelle läßt sich damit leicht erklären.

Im BDDF selbst treten ebenfalls Speicherallokationen auf. Außerdem besteht das BDDF mit seinen Schnittstellen aus mehreren untereinander synchronisierten Threads. Mit jedem ein-

treffenden Auftrag treten somit mehrere Thread-Umschaltungen auf. Für eine genauere Bestimmung der Bereiche größerer Verzögerungen ist allerdings eine detailliertere Untersuchung nötig.

Die im BDDF auftretenden Verzögerungen der Auftragsbearbeitung sind weitestgehend unvermeidlich. Dennoch bieten sich einige Ansatzpunkte für Optimierungsaufgaben. So effizient wie der Blocktreiber in Linux wird das BDDF auf Grund der Verwendung von IPCs und der langen Thread-Umschaltzeiten der L4-Basis aber nicht sein können.

Kapitel 6 – Zusammenfassung und Ausblick

Bisher waren unter DROPS lediglich ein SCSI- und ein IDE-Treiber vorhanden. Zur Unterstützung weiterer Bussysteme und zur Vereinigung deren Blockgeräte wurde ein Blockgeräte-Framework benötigt. Dieses sollte auch mehrere Scheduling-Verfahren unterstützen, sowie ein einheitliches Namensschema für die verwalteten Geräte und eine Unterstützung für Echtzeit-Aufträge bieten.

Zu Beginn der Arbeit wurden einige I/O-Scheduler näher betrachtet, um einen Überblick über die Ansprüche der Verfahren zu erhalten. Danach sind die Funktionsweisen von generic_blk und L4IDE untersucht worden. Beide sind potentiell als Teil oder Basis des zu entwerfenden Frameworks geeignet.

Im anschließenden Entwurf wurden Namensschema, die Client-Schnittstelle, die I/O-Scheduler-Schnittstelle und die Bustreiber-Schnittstelle entwickelt. Dabei wurde allgemeingültig vorgegangen und viel Spielraum für konkrete Implementierungen gelassen. Es stellte sich jedoch heraus, daß generic_blk nicht für das entworfene Framework geeignet ist. Es unterstützt beispielsweise nicht die geforderte flexible, vom Client bestimmbare Blockgröße der Geräte.

Bei der Implementierung konnten einige Ansätze oder Quellcode aus den bestehenden Projekten übernommen werden. So basiert die Partitions-Behandlung auf der von Linux 2.6. Ebenso konnte der IDE-Kern aus L4IDE in Form eines IDE-Bustreibers weitergenutzt werden. Beide Teile sind damit sehr leicht auf einem aktuellen Stand zu halten. Bisher sind nur ein NOOP-Scheduler und ein Scheduler nach dem Elevator-Prinzip implementiert worden. Der Elevator-Scheduler unterstützt die Angabe einer Zeitschranke für einzelne Aufträge. Die Implementierung eines Schedulers nach dem DAS-Prinzip soll demnächst folgen.

Das entstandene BDDF arbeitet zwar zufriedenstellend schnell, bietet jedoch noch ausreichend Ansatzpunkte für Optimierungen. So kann durch eine intelligentere Vorgehensweise die Benutzung der Funktionen `malloc` und `free` zum Allokieren und Freigeben der BIOs weitestgehend vermieden werden. Dies könnte durch eine dynamische, lastabhängige Vorallokation geschehen. Weiterhin könnten eventuell durch eine andere Thread-Struktur einige der zeitaufwendigen Thread-Umschaltungen eingespart werden.

Es ergeben sich auch Erweiterungsmöglichkeiten des BDDF. Um die Auftragsbearbeitung beim Einsatz eines RAID-Treibers[10] zu beschleunigen, könnte die Client-Schnittstelle, ähnlich der für Bustreiber, zusätzlich in einer direkt angebundenen Version implementiert werden. Weiterhin entspricht die Bezeichnung der physischen Geräte des IDE-Treibers noch der in Linux. Dies könnte dahingehend geändert werden, daß eine Unterteilung in Chipsatz und Kanal erfolgt. Außerdem ist mit dem IDE-Treiber noch kein TCQ möglich. Es ist im Linux bereits implementiert, aber noch deaktiviert. Es ist auch zweckmäßig, einen generischen Auftrag zu implementieren, welcher Informationen und eine Übersicht über den aktuellen Zustand eines I/O-Schedulers liefert. Durch die Suchfunktionen ist es möglich, ausgesuchte Geräte nur bestimmten Clients sichtbar zu machen oder bestimmten Clients zu verbergen. So kann es beispielsweise die Systemstabilität gefährden, eine Swap-Partition einem anderen Client als dem Speichermanager zugänglich zu machen.

Mit dem BDDF ist eine flexible Möglichkeit geschaffen worden, die Blockgeräte verschiedener Bussysteme einheitlich unter DROPS anzusprechen. Dabei ist es durch den Einsatz unterschiedlichster I/O-Scheduler möglich, exakt auf die Bedürfnisse von Nutzerprogrammen zu reagieren. Die Verarbeitungsgeschwindigkeit zeigt aber, daß es, auch in DROPS, noch viel Optimierungsarbeit erfordert, bis das BDDF ähnlich schnell arbeitet wie der vergleichbare Blocktreiber unter Linux. An Flexibilität ist das BDDF diesem allerdings überlegen.

10 Ein RAID-Treiber läßt sich im BDDF realisieren, indem er sich ebenfalls als Client anmeldet und die erhaltenen Aufträge über die Client-Schnittstelle an die entsprechenden Geräte weiterreicht.

A Glossar

ATA

AT Attachment – bezeichnet sowohl ein Protokoll als auch ein Bussystem, um Speichergeräte an einen Rechner zu koppeln.

DMA

Direct **M**emory **A**ccess – bezeichnet ein Verfahren, bei dem der Datenaustausch unabhängig von Prozessor erfolgt. Es ist sehr schnell und der reguläre Programmablauf wird nicht unterbrochen.

IDE

IDE ist eine veraltete, aber noch sehr verbreitete Bezeichnung für ATA.

IPC

Inter **P**rocess **C**ommunication – meint die Kommunikation der Prozesse untereinander. Dabei wird IPC sowohl für die Bezeichnung der Kommunikation als solche als auch für das dafür verwendete Dienstprimitiv verwendet.

Scatter/Gather-Liste

Ein Speicherbereich kann fragmentiert vorliegen. In solch einem Fall kann er durch Aneinanderreihen von Elementen beschrieben werden, die Ort und Größe der einzelnen Fragmente angeben. Diese Beschreibung nennt man Scatter/Gather-Liste (von engl. *to scatter* = verstreuen und *to gather* = sammeln).

TCQ

Tagged **C**ommand **Q**ueueing. Damit Laufwerke die erhaltenen Aufträge zur schnelleren Verarbeitung selbständig umsortieren können, müssen diese eindeutig gekennzeichnet sein. Durch das dafür vergebene Tag weiß ein Treiber genau, welcher Auftrag soeben vom Laufwerk bearbeitet wurde.

Thread

Innerhalb eines Prozesses kann es mehrere parallele Programmabläufe geben. Diese werden als Threads bezeichnet. Sie dienen beispielsweise dazu, mehrere Instanzen ein und desselben Programmteiles zu bilden.

TSC

Time **S**tamp **C**ounter – bezeichnet einen Zähler, der bei jedem Taktschlag des Prozessors um eins erhöht wird. Dadurch ist eine besonders feingranulare Zeitmessung möglich.

B Abbildungsverzeichnis

C Literaturverzeichnis

[DIC] *DROPS IDL Compiler*. URL http://os.inf.tu-dresden.de/dice. Aktualisierungsdatum: 09.01.2004. Technische Universität Dresden, Institut für Systemarchitektur

[DRO] *Dresden Realtime Operating System*. URL http://os.inf.tu-dresden.de/drops. Aktualisierungsdatum: 03.12.2003. Technische Universität Dresden, Institut für Systemarchitektur

[HLR+01] C.-J. HAMAN, J. LÖSER, L. REUTHER, S. SCHÖNBERG, J. WOLTER, H. HÄRTIG. *Quality-Assuring Scheduling – Using Stochastic Behavior to Improve Resource Utilisation*. In: *Proceedings of the 22nd IEEE Real-Time Systems Symposium (RTSS)*. ISBN 0-7695-1420-0. S. 119-128. London, UK. Dezember 2001

[JW91] D. M. JACOBSEN, J. WILKES. *Disk scheduling algorithms based on rotational position*. Technical Report HPL-CSP-91-7rev1. HP Laboratories. Palo Alto, CA, USA. März 1991

[LIE96] JOCHEN LIEDTKE. *L4 Reference Manual*. GMD - German National Research Center for Information Technology, Arbeitspapier 1021. September 1996

[Meh98] FRANK MEHNERT. *Ein zusagenfähiges SCSI-Subsystem für DROPS*. Technische Universität Dresden, Institut für Betriebssysteme, Datenbanken und Rechnernetze. Diplomarbeit. Januar 1998

[Men03] MAREK MENZER. *Portierung des DROPS Device Driver Environment (DDE) für Linux 2.6 am Beispiel des IDE-Treibers*. Technische Universität Dresden, Institut für Systemarchitektur. Großer Beleg. September 2003

[Poh02] MARTIN POHLACK. *Ermittlung von Festplatten-Echtzeiteigenschaften*. Technische Universität Dresden, Institut für Systemarchitektur. Großer Beleg. Mai 2002

[QLX] *QLinux*. URL http://lass.cs.umass.edu/software/qlinux. Aktualisierungsdatum: 13.02.2002. University of Massachusetts, Amherst, Department of Computer Sience

[Reu01] LARS REUTHER. *DROPS Block Device Driver Interface Specification*. Technische Universität Dresden, Institut für Systemarchitektur. 2001

[RP03] L. REUTHER, M. POHLACK. *Rotational-Position-Aware Real-Time Disk Scheduling Using a Dynamic Active Subset (DAS)*. In: *Proceedings of the 24th IEEE Real-Time Systems Symposium (RTSS 2003)*. ISBN 0-7695-2044-8. S. 374-385. Cancun, Mexico. Dezember 2003

[She98] PRASHANT J. SHENOY. *Symphony: An Integrated Multimedia File System*. University of Texas at Austin, Department of Computer Science. PhD Thesis. August 1998

[SV98] PRASHANT J. SHENOY, HARRICK M. VIN. *Cello: A Disk Scheduling Framework for Next Generation Operating Systems*. In: *Proceedings of the 1998 ACM SIGMETRICS Joint International Conference on Measurement and Modeling of Computer Systems*. ISSN 0163-5999. S. 44-55. Madison, WI, USA. Juni 1998

[Ven02] BADRINATH VENKATACHARI. *Better Admission Control and Disk Scheduling for Multimedia Applications*. Worcester Polytechnic Institute, Department of Computer Science. Worcester, MA, USA. MS Thesis. Mai 2002