Diplomarbeit

zum Thema

Übertragung des Device-Driver-Environment-Ansatzes auf Subsysteme des BSD-Betriebssystemkerns

Thomas Friebel

28.02.2006

Technische Universität Dresden
Fakultät Informatik
Institut für Systemarchitektur
Professur Betriebssysteme

Betreuender Hochschullehrer: Prof. Dr. rer. nat. Hermann Härtig
Betreuender Mitarbeiter: Dipl.-Inf. Christian Helmuth

Danksagung

Ich möchte allen meinen Freunden und Verwandten danken, die mich unterstützt haben. Besonderer Dank gilt meinen Eltern für ihre Unterstützung, Anett für ihr Vertrauen in meine Fähigkeiten und ihre ständige Motivation, Bernd dafür, dass er mich zum Sport überredet hat, Dirk für die gemeinsamen, theoretischen Versuche die Welt zu verbessern und Madeleine nicht nur für´s Korrekturlesen. Desweiteren möchte ich meinem Betreuer für die vielen Hinweise und Denkanstöße danken.

Inhaltsverzeichnis

Kapitel 1

Einführung

Alle verbreiteten PC-Betriebssysteme, wie zum Beispiel auch Microsoft Windows oder GNU/Linux, besitzen monolithische Kerne. Bei diesen sind nahezu alle Gerätetreiber Bestandteil des Kerns und werden so im privilegierten Prozessormodus ausgeführt. Damit haben sie Zugriff auf alle Ressourcen, wie zum Beispiel den Arbeitsspeicher und alle Geräte des Rechners. Ein fehlerhafter oder bösartiger Treiber, oder auch jede andere Kernkomponente, kann dadurch andere Komponenten beeinflussen und somit fehlerhaftes Verhalten hervorrufen und Teile oder das gesamte System zum Absturz bringen.

Anders dagegen bei Mikrokernbetriebssystemen wie DROPS, das zu Forschungszwecken an der TU-Dresden entwickelt wird. Hier sind so viele Komponenten wie möglich in das Nutzerland ausgelagert und arbeiten somit im deprivilegierten Modus. Im Mikrokern verbleiben nur wenige Mechanismen, die für ein darauf aufbauendes Betriebssystem notwendig sind: Im allgemeinen nur die Verwaltung von Threads und Adressräumen und die Möglichkeit für Threads miteinander zu kommunizieren (inter-process communication, IPC). Jegliche Strategien und höheren Dienste wie zum Beispiel Scheduling, Gerätetreiber, Dateisysteme und Netzwerkprotokolle werden deprivilegiert als Prozesse ausgeführt. Da sie somit alle in eigenen Adressräumen liegen, sind sie voreinander geschützt und können sich nicht mehr gegenseitig unkontrolliert beeinflussen. Die Kommunikation findet dann per IPC oder durch gemeinsam benutzte Speicherbereiche statt. Auf diese Weise stellen sie ihre Funktionalität als Server den anderen Komponenten zur Verfügung.

Gerätetreiber machen bei Linux 2.6 gemessen an der Zahl der Quelltextzeilen einen Anteil von ca. 65% aus. Die Entwicklung und Wartung von Treibern fordert also einen gewichtigen Aufwand. Dieser kann noch dadurch erhöht werden, dass Gerätehersteller nur eine unvollständige, fehlerhafte oder gar keine Spezifikation der Geräte veröffentlichen. Desweiteren sollten unbedingt Exemplare der Geräte zur Verfügung stehen, um die Treiber testen zu können. Eine Neuentwicklung von Treibern für eine größere Anzahl an Geräten fordert also einen immensen Aufwand.

Daher bietet es sich an, bei der Entwicklung eines neuen Betriebssystems (im folgenden *Zielsystem* genannt) auf die Treiberbasis eines etablierten Systems (*Legacy-System*) zurückzugreifen und diese wiederzuverwenden. Um den Aufwand dafür zu minimieren, sollten sich die Treiber des Legacy-Systems unverändert übernehmen lassen. Dazu wird ein Adapter benötigt, welcher die vom Treiber erwartete Umge-

bung emuliert und die entsprechende Funktionalität auf das Zielsystem abbildet. Mit dde_linux existiert bereits ein solcher Adapter für die Benutzung von Linux-Gerätetreibern und anderen Linuxkomponenten unter DROPS. *DDE* steht dabei für device driver environment.

Ziel dieser Arbeit soll es nun sein, den DDE-Ansatz auf Subsysteme des FreeBSD-Betriebssystemkerns abzubilden. Dazu soll eine entsprechende Emulationsbibliothek entwickelt werden, welche die grundlegende Kernfunktionalität zur Verfügung stellt. Desweiteren soll ein Server implementiert werden, der mit Hilfe des FreeBSD-ATA-Treibers unter DROPS den Zugriff auf Festplatten erlaubt.

Gliederung

Die Arbeit ist wie folgt aufgebaut: Im zweiten Kapitel werden die zum Verständnis der Arbeit notwendigen Grundlagen betrachtet. Dazu werden im ersten Teil die beteiligten Komponenten und im zweiten Teil verwandte Arbeiten vorgestellt. Das dritte Kapitel zeigt und erläutert den Entwurf des DDEs und seiner Komponenten. Dabei wird genauer auf die Vor- und Nachteile der Wiederverwendung von Legacy-Quelltext eingegangen und es werden einige Aspekte der Fehlerisolation betrachtet. Das vierte Kapitel widmet sich der Implementierung des DDEs und geht dabei auf einige Besonderheiten ein. Im fünften Kapitel werden dann Entwurf und Implementierung des ATA-Treiberservers erläutert.

Das sechste Kapitel ist in drei Teile gegliedert. Im ersten wird der Entwicklungsaufwand des FreeBSD-DDEs mit dem von dde_linux verglichen und erörtert. Der zweite Teil befasst sich mit der Messung und Bewertung der Leistung des ATA-Treiberservers, wiederum im Vergleich zum dde_linux-Pendant. Der dritte Teil des sechsten Kapitels widmet sich der Übertragbarkeit des DDE-Ansatzes in Bezug auf andere Geräteklassen, andere Legacy- und andere Zielsysteme. Das siebte Kapitel zeigt dann Möglichkeiten zur Weiterentwicklung der entstandenen Lösungen auf, und im abschließenden achten Kapitel werden die Ergebnisse dieser Arbeit noch einmal zusammengefasst.

Kapitel 2

Grundlagen und Stand der Technik

In diesem Kapitel sollen einige für die Arbeit notwendigen Grundlagen erläutert werden. Im ersten Abschnitt wird die Verwaltung von Geräten und Treibern in FreeBSD, dem verwendeten Legacy-System, näher erklärt. Danach wird kurz auf das Zielsystem DROPS eingegangen. Im dritten Abschnitt werden dann einige verwandte Arbeiten vorgestellt.

2.1 FreeBSD

FreeBSD (FBSD) steht unter dem „FreeBSD Copyright" frei zur Verfügung. Diese Lizenz erlaubt die Benutzung und Weitergabe als Quelltext oder in kompilierter Form, auch mit eigenen Veränderungen. Der Lizenztext muss allerdings in den Quelltexten erhalten bleiben und sich in der Dokumentation zu daraus entstandenen Produkten wiederfinden.

Da diese Arbeit am Beispiel des FreeBSD ATA-Treibers durchgeführt wurde, soll in diesem Abschnitt kurz dessen Einbindung in FreeBSD erläutert werden. Das beinhaltet auf der einen Seite seine Rolle in der Gerätehierarchie und auf der anderen Seite im GEOM, einem Framework zur Transformation von Blockgeräte-I/O-Aufträgen.

2.1.1 Gerätehierarchie

FreeBSD-Treiber sind mit Klassen aus dem Paradigma der objektorientierten Programmierung vergleichbar. Deren Instanzen, die logischen Geräte, sind in einer Baumstruktur angeordnet. Ein Ausschnitt eines solchen Baumes wird in Abbildung 2.1 dargestellt.

Die Namen der logischen Geräte werden aus dem Namen der jeweiligen Klasse und einer laufenden Nummer gebildet. Die Wurzel des Gerätebaumes ist der Knoten root0, der keine besondere Funktionalität besitzt. So reicht er zum Beispiel die Aufrufe, um den Ruhezustand des Systems zu steuern, einfach an alle seine Kindknoten weiter. Beim Starten des Systems wird dem root-Knoten eine Instanz der architekturspezifischen Klasse nexus hinzugefügt und der Autokonfigurationsmechanismus (Abschnitt 2.1.2) angestoßen. Die Klasse nexus ist für die Verwaltung und Zuordnung der Systemressourcen also I/O-Ports, eingeblendeter I/O-Speicher (memory-mapped I/O),

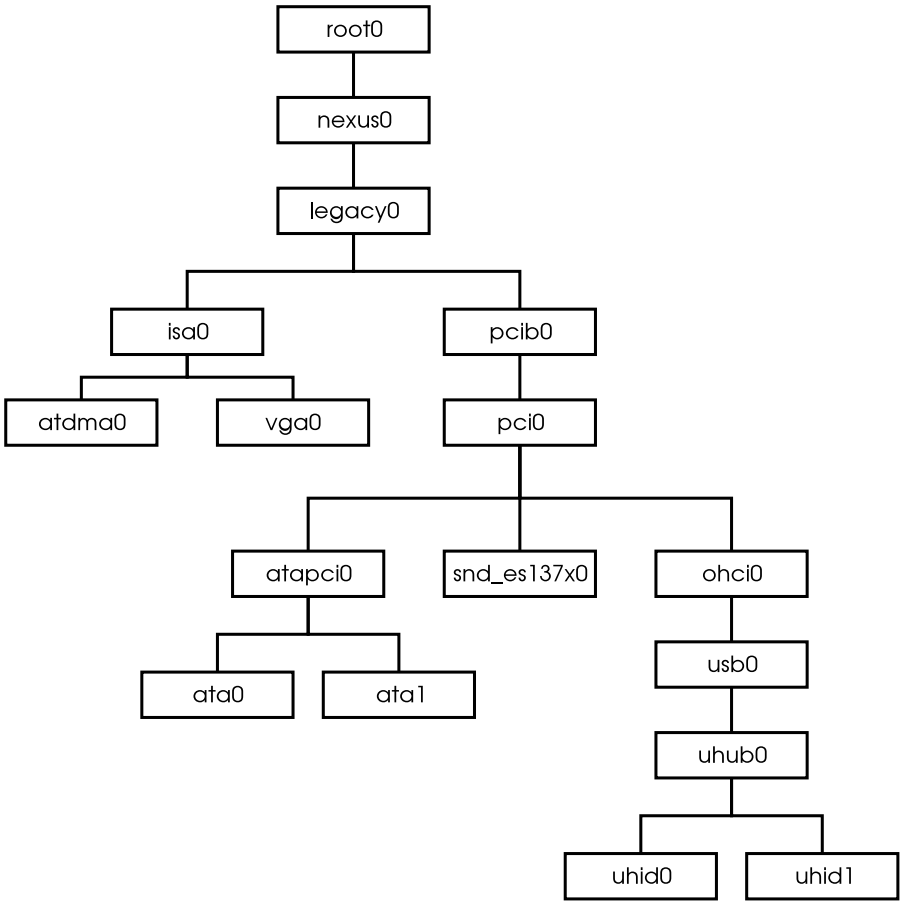

Abbildung 2.1: Beispiel für eine FreeBSD Gerätehierarchie

ISA-DMA-Kanäle und Interrupts zuständig. Darunter folgt eine Instanz der Klasse legacy, unter der die Repräsentationen der Geräte folgen.

In der ersten Ebene unterhalb legacy0 finden sich die logischen Geräte für die Systembusse. In Abbildung 2.1 sind das der ISA- und der PCI-Bus. Für letzteren gibt es einen Low-Level- (pcib) und einen darauf aufbauenden High-Level-Treiber (pci). Am ISA-Bus befinden sich zwei Geräte: ein ISA-DMA-Controller und eine Grafikkarte, am PCI-Bus sind weitere Geräte vorhanden. Einige davon, zum Beispiel der ATA-Controller, sind selbst Bustreiber und besitzen Kindknoten – hier die beiden ATA-Kanäle. Auf diese Art und Weise sind die verschiedenen Funktionen der physischen Geräte und ihre Verbindung zueinander in einer Baumhierarchie angeordnet.

2.1.2 Autokonfiguration

In jedem Treiber ist die Klasse seines Elternknotens festgelegt. Bei der Autokonfiguration oder nach dem späteren Hinzufügen eines neuen Knotens, wird anhand dieser Information die Liste der möglichen Treiber durchlaufen. Dabei wird die `probe()`-Methode eines jeden Treibers aufgerufen und ihr der neue Knoten als Parameter übergeben. Darüber kann der Treiber auf das Gerät zugreifen und prüfen, ob er es unterstützt. Durch den Rückgabewert der Funktion wird dabei der Erfolg oder Misserfolg mitgeteilt: Werte größer als Null zeigen Fehler an, oder dass das Gerät nicht erkannt wurde. Werte kleiner oder gleich Null bedeuten, dass das Gerät unterstützt wird. Der Treiber, der den höchsten Wert kleiner oder gleich Null zurück gibt, wird als der zuständige Treiber für das Gerät gesetzt und seine `attach()`-Methode wird aufgerufen. Hier initialisiert er das Gerät. Handelt es sich um einen Bustreiber, so sucht er nach daran angeschlossenen Geräten und erstellt für jedes einen Kindknoten, für den wiederum auf die gleiche Weise ein Treiber gesucht wird. So entsteht ein Gerätebaum, wie er beispielhaft in Abbildung 2.1 dargestellt ist.

Desweiteren besteht für jeden Treiber die Möglichkeit selbst nach einem kompatiblen Gerät zu suchen. Dazu wird seine `identify()`-Methode aufgerufen und ihr der zu durchsuchende Bus übergeben. Dort kann der Treiber selbst einen Kindknoten unterhalb des Busses erstellen und kann später über den probe-Mechanismus gefunden werden.

2.1.3 GEOM

Beim GEOM handelt es sich um ein Framework zur Transformation von I/O-Aufträgen für Blockgeräte. Eine typische Transformation ist zum Beispiel das RAID-Striping – das Zusammenfassen zwei- oder mehrerer Blockgeräte zu einem – oder die Einteilung in Partitionen. Die Transformationen werden *GEOM-Klassen* genannt, eine Instanz davon *Geom*. Die Geoms können beliebig miteinander verknüpft werden, so dass beispielsweise die Partitionierung einer RAID-Verbindung mehrerer Partitionen problemlos möglich ist.

Die Verknüpfung der Geoms geschieht über *GEOM-Provider* und *-Consumer*. Ein Provider stellt einen logischen Datenträger dar; ein Consumer ist ein Objekt, welches mit einem Provider verknüpft wird und durch das I/O-Aufträge an diesen gesendet werden können. So erstellt der ATA-Treiber für jede Festplatte ein Geom in Verbin-

dung mit einem Provider. Bei Erstellung eines neuen Providers werden alle Klassen darüber informiert und können prüfen ob sie sich mit ihm verknüpfen wollen. Die GEOM-Klasse DEV zum Beispiel erstellt dann ein Geom mit einem Consumer und einen Eintrag unter /dev durch den vom Nutzerland aus darauf zugegriffen werden kann.

2.1.4 Einbindung des ATA-Treibers

Wird ein PCI-Gerät gefunden, wird der PCI-ATA-Treiber durch den Autokonfigurationsmechanismus instanziiert. Nach erfolgreichem Probing wird er in den Gerätebaum eingetragen und erzeugt pro ATA-Kanal einen Kindknoten der Klasse ata. Der Autokonfigurationsmechanismus verknüpft diese per `probe(..)` und `attach(..)` mit dem ATA-Kanaltreiber. Dieser stellt bei der Initialisierung jeweils den Typ des Master- und des Slave-Gerätes fest und wählt die entsprechenden Funktionen zur Initialisierung der Geräte. Dabei werden dann direkt oder indirekt jeweils eine Instanz einer GEOM-Klasse (zum Beispiel DISK bei Festplatten oder ACD bei CD-ROMs) und ein dazugehöriger GEOM-Provider erstellt, an den dann andere Geoms anknüpfen können.

2.2 DROPS

DROPS (DROPS) – **D**resden **R**ealtime **Op**erating **S**ystem – bezeichnet das Forschungssystem der Betriebssystemgruppe der TU-Dresden. Der Fokus hat sich im Laufe der Zeit auf Sicherheitsaspekte eines Betriebssystems verschoben, wobei dies Echtzeitanforderungen aber weitgehend mit einschließt.

Als Kern wird Fiasco (Fiasco) verwendet, welcher die L4-Spezifikation (L4Spec) implementiert. Dabei handelt es sich um einen Microkernel, also einen minimalen Kern. Er bietet nur die nötigen Primitive – Tasks und Threads, Adressraumtrennung und IPC – und keinerlei Strategien. So werden zum Beispiel Scheduling, Speicherverwaltung und Gerätetreiber von Tasks im Nutzerland, so genannten Servern, übernommen.

Unter DROPS existieren eine Reihe solcher Server – zum Beispiel der Namensserver names, dm_phys zur Verwaltung des physischen Speichers, l4io unter anderem zum Verwalten von Interrupts und PCI-Geräten und das im nächsten Abschnitt beschriebene bddf.

Die Server names, dm_phys und l4io sind neben anderen Bestandteil des Common L4 Environment (L4Env). Dabei handelt es sich um eine Reihe von Bibliotheken und Servern, die eine grundlegende Programmierumgebung zur Anwendungsentwicklung auf L4 bereitstellen. Dies beinhaltet desweiteren die Unterstützung von Threads, Synchronisation durch Semaphoren und Mutexe, Text-/Grafikausgabe, das Starten weiterer Anwendungen und mehr.

2.2.1 bddf – block device driver framework

Beim bddf (Men04) handelt es sich um einen Server zur Verwaltung von Blockgeräten in DROPS. Bei ihm melden sich die verschiedenen Blockgerätetreiber, also zum Beispiel ein ATA-Treiber, an und registrieren die vorhandenen Geräte – typischerweise

Festplatten oder CD-Laufwerke. Das bddf kümmert sich vollständig um deren Verwaltung. Es liest die Partitionstabelle aus, erstellt die logischen Laufwerke, und es ist für das Scheduling der I/O-Aufträge zuständig. Jedem physischen Laufwerk ist dabei ein Scheduler zugeordnet, der auch zur Laufzeit ausgetauscht werden kann. Die Scheduler sind Bestandteil des bddf. Die Kommunikation zwischen bddf und Gerätetreiber erfolgt über IPC. Es besteht aber auch die Möglichkeit ein oder mehrere Treiber zum bddf hinzuzulinken, wobei die IPC dann durch direkte Funktionsaufrufe ersetzt wird.

Die Clients übertragen ihre Aufträge per IPC an das bddf. Ihre Bearbeitung erfolgt asynchron, die IPC kehrt also sofort zurück und nicht erst bei Erledigung des Auftrags. Clients können danach oder in einem anderen Thread parallel dazu per IPC auf das Ende der Auftragsbearbeitung warten.

Das bddf unterstützt auch Echtzeit-Datenströme. Die Clients müssen dazu die Ströme reservieren und erhalten dabei eine Rückmeldung, ob diese akzeptiert werden können oder nicht. Ein Echtzeit-Datenstrom wird durch die Periodendauer, die Anzahl der Aufträge pro Periode, die Größe eines einzelnen Auftrags und einen Qualitätsparameter bestimmt. Aufträge die nicht einem solchen Strom zugeordnet sind, können auch eine Deadline angeben, bis zu welcher der Auftrag abgearbeitet sein sollte. Allerdings kann eine rechtzeitige Abarbeitung nicht vom Scheduler garantiert werden, da keine vorherige Reservierung stattgefunden hat. Die komplette Echtzeitunterstützung wird vom Scheduler realisiert und muss vom Gerätetreiber nicht unterstützt werden.

2.3 Verwandte Arbeiten

Der gleichen Problemstellung, also der Wiederverwendung von Legacy-Kern-Treibern im Nutzerland, widmen sich auch die in den nächsten beiden Abschnitten vorgestellten Arbeiten – dde_linux und DD/OS – jeweils auf unterschiedliche Weise. Ein anderer interessanter Ansatz ist der von Project Evil, auf den in Abschnitt 2.3.3 eingegangen wird.

2.3.1 dde_linux

dde_linux ist die Bezeichnung für einen schlanken Emulationslayer für Linux-Gerätetreiber auf DROPS. Der zu verwendende Treiber muss dabei im Quelltext vorliegen. Er wird neu kompiliert und mit zwei Bibliotheken verlinkt. Die eine Bibliothek stellt die allgemeine Linux-Kernfunktionalität zur Verfügung, die von jedem Treiber benötigt wird. Die andere Bibliothek stellt die geräteklassen-spezifische Funktionalität bereit. Momentan existieren bereits die spezifischen Bibliotheken für ATA-, USB-, Netzwerk- und Audiotreiber. Die Portierung des Linux-IP-Stacks auf DROPS basiert ebenfalls auf dde_linux.

Der Zugriff auf die Gerätetreiber wird dann über einen geräteklassen-spezifischen Server, der den Treiber und die Bibliotheken beinhaltet, dem gesamten System zur Verfügung gestellt. Wenn immer nur ein Client den Treiber benutzt, kann er auch zu diesem direkt dazu gelinkt werden.

2.3.2 DD/OS

In (LU04) werden zwei Varianten eines anderen Ansatzes beschrieben: Die erste Variante ist die Ausführung des Legacy-Kerns in einer virtualisierten Umgebung. Interessanter für diese Arbeit ist die zweite Variante, die hier betrachtet werden soll: Der gesamte Legacy-Kern wird paravirtualisiert. Er bleibt also in seiner gesamten Struktur erhalten und wird nur an wenigen Stellen an das Zielsystem angepasst. Auf dem Zielsystem läuft der paravirtualisierte Legacy-Kern dann als Task im Nutzerland. Er kann dabei beliebige Legacy-Treiber enthalten und wird auch nur dafür benutzt – daher der Name DD/OS (device driver operating system).

Der Zugriff auf die Treiber erfolgt durch ein Legacy-Kernmodul oder durch eine Task im Legacy-Nutzerland. Diese stellen dann die Treiber dem gesamten Zielsystem zur Verfügung. Auf Basis von DD/OS wurde schon der Zugriff auf ATA-Geräte und Netzwerkkarten realisiert.

2.3.3 Project Evil

Project Evil ist der Arbeitstitel für NDISulator (NDISu), mit dessen Hilfe Windowstreiber für Funknetzwerkkarten unter FreeBSD benutzt werden können. Ein ähnliches Projekt für Linux ist NdisWrapper (NdisW). Beide implementieren die Network Driver Interface Specification (NDIS) – die Windows-Treiber-API für Netzwerkkarten – und die notwendigen Teile der Windows-Kern-API als Kernmodul des Zielsystems. Im Unterschied zu dde_linux und DD/OS steht dazu nicht der Legacy-Quelltext von Kern und Treiber zur Verfügung. Deshalb wird der binäre (kompilierte) Windows-Treiber zur Laufzeit zu dem Kernmodul und damit zum FreeBSD-Kern oder entsprechend zu Linux dazu gelinkt. NdisWrapper unterstützt viele Funknetzwerkkarten am PCI-, USB- und am Cardbus. Bei NDISulator fehlt momentan noch USB-Unterstützung.

Kapitel 3

Entwurf

Der Ansatz von dde_linux, der dieser Arbeit zu Grunde liegen soll, zeichnet den Entwurf schon teilweise vor: Bei dem DDE soll es sich um eine oder mehrere Bibliotheken handeln, welche die von einem FreeBSD-Gerätetreiber oder -Subsystem erwartete Funktionalität zur Verfügung stellt. Allerdings soll die Vorgehensweise von (Hel01) nicht nur wiederholt werden – falls möglich sollen neue Ansätze und Ideen ausgearbeitet werden. Den Spielraum dafür bieten einerseits der Entwurf zur Implementierung der DDE-Funktionalität und andererseits die Anbindung des DDEs an das Zielsystem. Auf letzteres soll hier zuerst eingegangen werden, da es die Grundlage des DDEs bildet. Die darauffolgenden beiden Abschnitte befassen sich mit dem Entwurf der DDE-Komponenten und erörtern die Wiederverwendung von Legacy-Quelltext bei ihrer Implementierung. Abschließend wird auf einige Sicherheitsaspekte im Zusammenhang mit Gerätetreibern eingegangen.

3.1 ddekit – der DDE-Baukasten

Bei dde_linux wird die DDE-Funktionalität direkt auf die DROPS-Anwendungsumgebung L4Env abgebildet. Dadurch sind beide sehr eng miteinander verknüpft und das DDE lässt sich nur sehr schwer auf andere Zielsysteme portieren. Darüber hinaus kann somit eine einzelne Veränderung an der L4Env-Schnittstelle viele Anpassungen in großen Teilen des DDEs erfordern. Daher soll hier eine Zwischenebene – das *ddekit* – eingeführt werden. Um zusätzliche Abhängigkeiten zu vermeiden, sollte das ddekit die einzige Schnittstelle eines DDEs zum darunter liegenden System darstellen. Desweiteren gibt es sicher bei unterschiedlichen DDEs – trotz unterschiedlicher Legacy-Systeme – einige gemeinsame Komponenten. Diese sollten in einer gemeinsam verwendeten Bibliothek abgelegt werden, wobei das ddekit ebenfalls diese Funktion einer Bibliothek von allgemeinen DDE-Komponenten übernehmen könnte. Das ddekit dient so als Grundlage zur Entwicklung eines DDEs – als DDE-Baukasten.

Abbildung 3.1 zeigt nochmal die Gliederung des DDEs in die zwei Teile: den unteren Teil, das ddekit, als Grundlage und zusätzliche Abstraktionsebene zum darunter liegenden System für den oberen Teil, das eigentliche DDE, welches die Umgebung für Legacy-Kernkomponenten bildet. Analog zu dde_linux soll das eigentliche DDE für FreeBSD hier *dde_fbsd* heißen.

Auf die sich aus dieser Aufteilung ergebenden Vor- und Nachteile wird in den

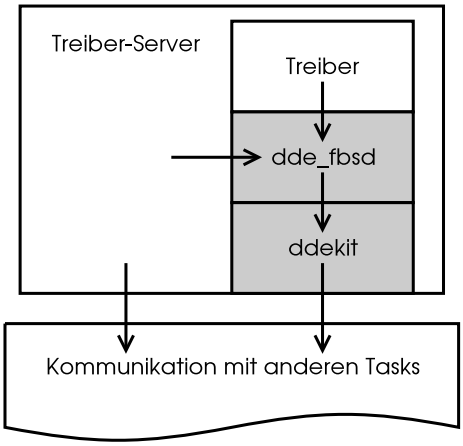

Abbildung 3.1: Trennung des DDEs (grau unterlegt) in das FreeBSD-spezifische dde_fbsd, und das unspezifische ddekit als Grundlage. Die Pfeile stellen die Abhängigkeiten der beteiligten Komponenten dar.

nächsten beiden Abschnitten eingegangen. Danach soll die notwendige Funktionalität des ddekits erläutert werden.

3.1.1 Vorteile

Da mit dem ddekit eine Art Baukasten zur Verfügung steht, sollte sich mit seiner Hilfe die Entwicklung weiterer DDEs für andere Legacy-Systeme stark vereinfachen. Der Aufwand für dde_fbsd wird in Abschnitt 6.1 analysiert und verglichen. Desweiteren lassen sich mehrere auf dem ddekit basierende DDEs leichter warten. Bei Veränderungen der Schnittstellen des darunter liegenden Systems ist nur das ddekit anzupassen, das nach oben hin, also zum jeweiligen DDE, seine Schnittstellen beibehalten kann.

Das ddekit kann ebenfalls als kleines Treiberframework gesehen werden, da mit seiner Hilfe auch Treiber neu entwickelt werden können. Ebenso ist die Entwicklung eines auf dem ddekit basierenden vollständigen Treiberframeworks möglich. Es sollte auch möglich sein, mehrere Treiber unterschiedlicher Systeme gleichzeitig mit einer ddekit-Instanz zu verwenden. So könnte ein Treiber-Superserver entstehen, der alle verschiedenartigen Treiber vereint, wie er in Abbildung 3.2 dargestellt ist.

3.1.2 Nachteile

Aus der Einführung einer zusätzlichen Zwischenschicht ergibt sich der Nachteil, dass damit auch zusätzliche Funktionsaufrufe eingeführt werden – zuerst wird die Zwischenschicht betreten, bevor diese die Funktion des Zielsystems aufruft. Das kann die Leistung des DDEs beeinträchtigen. Dem kann aber dadurch begegnet werden, dass die leistungsrelevanten Funktionen inline definiert werden.

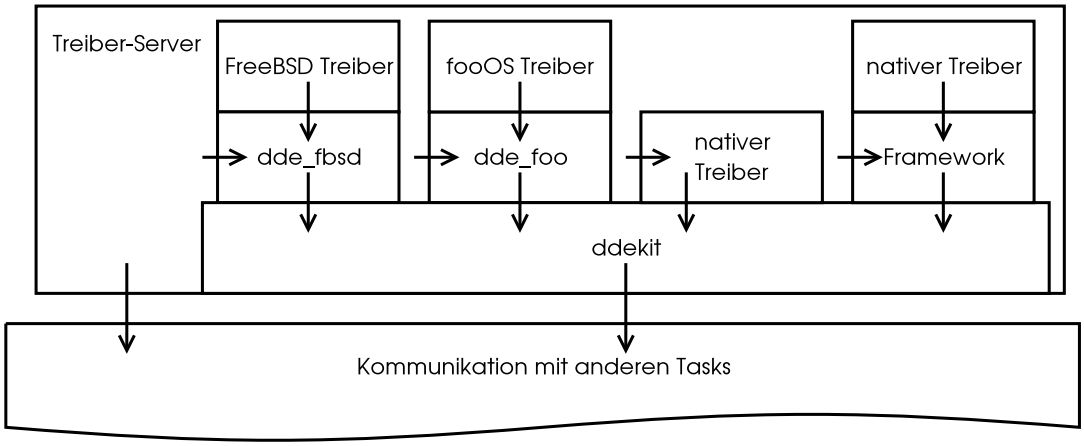

Abbildung 3.2: Ein Treiber-Superserver mit vielen Treibern, auch unterschiedlicher Herkunft, ist vorstellbar.

3.1.3 Funktionalität

Da das ddekit die einzige Schnittstelle eines DDE zum darunter liegenden System sein soll, muss es jegliche von einem DDE benötigte Systemfunktionalität zur Verfügung stellen. Zusätzlich soll es noch die Funktionen beinhalten, die bei der Implementierung eines DDE häufig benötigt werden. Die gesamte Funktionalität kann hier jedoch noch nicht festgelegt werden, da bei der Entwicklung eines DDEs für ein weiteres Legacy-System möglicherweise zusätzliche Anforderungen auftreten.

Threads

Beim Erstellen neuer Threads durch `dde_thread_create(..)` kann diesen zur besseren Unterscheidbarkeit, zum Beispiel bei der Fehlersuche, ein Name zugeordnet werden. Der Rückgabewert der Funktion ist ein Zeiger auf eine opake Struktur und dient als Handle, um einen Thread zu spezifizieren. Mittels `dde_thread_msleep(..)` und `dde_thread_usleep(..)` kann sich ein Thread für die in Milli- bzw. Mikrosekunden angegebene Zeit schlafen legen. Mit einem Aufruf von `dde_thread_sleep(dde_lock_t*lock)` dagegen legt sich ein Thread für unbestimmte Zeit schlafen. Bei dem übergebenen `lock` handelt es sich um einen Mutex, der gehalten, also gesperrt sein muss, und freigegeben wird, sobald der Thread angehalten wurde und mittels `dde_thread_wakeup(..)` wieder aufgeweckt werden kann. Desweiteren können mit `dde_thread_get/set_data(..)` und `dde_thread_get/set_my_data(..)` thread-lokale Daten hinterlegt beziehungsweise abgefragt werden.

Speicherverwaltung

Speicher kann auf zwei verschiedene Arten angefordert beziehungsweise wieder frei gegeben werden: einerseits werden für kleine Speicherbereiche (weniger als eine Speicherseite) Slab-Caches angeboten, andererseits für die Anforderung gan-

zer oder mehrerer Speicherseiten die Funktion `large_malloc(..)`. Letztere können mit `dde_large_free(..)` wieder frei gegeben werden.

Mit Hilfe von Slab-Caches können Objekte gleicher Größe schnell allokiert und wieder frei gegeben werden. Ein Slab-Cache wird mit `dde_slab_init(size)` erstellt. Der Parameter `size` legt die Größe der zu verwaltenden Speicherobjekte fest. Mittels `dde_slab_alloc(..)` und `dde_slab_free(..)` erfolgt die Allokation beziehungsweise die Freigabe der Objekte und durch `dde_slab_destroy(..)` wird ein Slab-Cache wieder entfernt.

Desweiteren wird für jeden verwendeten Speicher die Zuordnung von virtueller zu physischer Adresse verwaltet. Dies geschieht durch klassische Seitentabellen, die bei jeder Allokation und jeder Freigabe von Speicherseiten aktualisiert werden. Mittels `dde_get_pte(..)` kann die Information abgefragt, mittels `dde_set_ptes(..)` für eine oder mehrere Seiten gesetzt, und durch `dde_clear_ptes(..)` wieder gelöscht werden.

Interrupts

Die Funktionalität bezüglich Interrupts wird auf den L4IO-Server abgebildet. Mittels `dde_interrupt_attach(irq,th_init,handler,..)` kann ein Interrupt-Handler registriert werden. Dazu muss neben der Nummer des Interrupts und einem Zeiger auf den zu installierenden Handler noch ein Zeiger auf eine Funktion zur Initialisierung eines Threads (`th_init`) übergeben werden. Für eine parallele Abarbeitung soll die Interruptbehandlung in einem eigenen Thread pro Interruptnummer stattfinden. Da das Legacy-System das Vorhandensein einiger thread-eigener Strukturen voraussetzt, ruft der neu erstellte Thread die per `th_init` übergebene Funktion zur Initialisierung auf, bevor er Interrupts empfängt.

I/O-Ports, I/O-Speicher, PCI-Geräte

Zur Reservierung und Wiederfreigabe von IO-Ports, eingeblendetem I/O-Speicher und ISA-DMA-Kanälen werden die entsprechenden Funktionen `dde_request_io(..)`, `dde_release_io(..)` u.s.w. angeboten. Der Zugriff auf den PCI-Bus wird über eine Funktion zum Suchen von Geräten und mehrere Funktionen zum Lesen und Schreiben der Konfiguration bereitgestellt. Dies alles wird ebenfalls auf den L4IO-Server abgebildet.

Initcalls

Initcalls dienen dazu, dass bestimmte Initialisierungsfunktionen beim Starten einer Task aufgerufen werden. Dazu werden die entsprechenden Funktionen im Quelltext mittels `DDE_INITCALL(..)` markiert und bei einem Aufruf von `dde_do_initcalls()` ausgeführt. Die Implementierung kann aus dde_linux übernommen werden: Es werden Zeiger auf die entsprechenden Funktionen in einem eigenen Abschnitt der ELF-Binärdatei abgelegt, die innerhalb `dde_do_initcalls()` abgearbeitet werden.

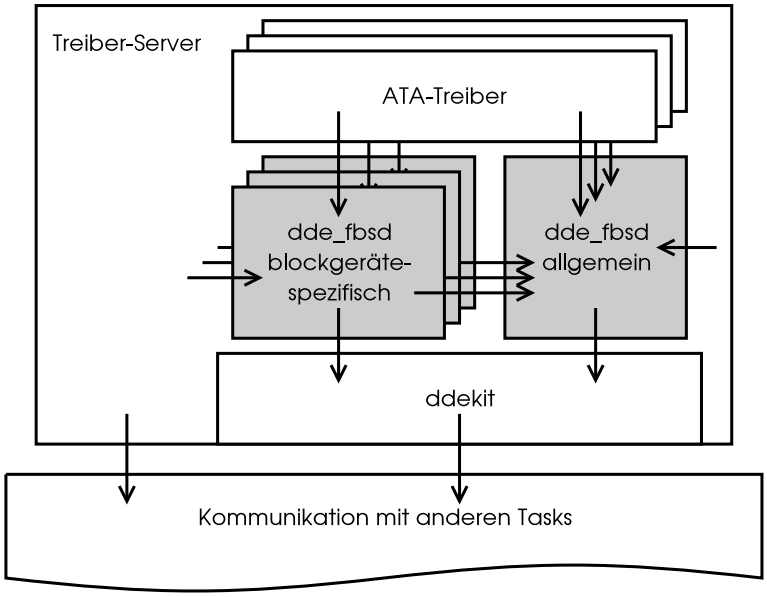

Abbildung 3.3: Die Komponenten von dde_fbsd (grau unterlegt) und ihre Abhängigkeiten anhand eines Treiber-Servers mit mehreren FreeBSD-Treibern

sonstiges

Zur Synchronisation von Threads untereinander werden Mutexe, Semaphore und Bedingungsvariablen angeboten. Desweiteren besteht die Möglichkeit der Textausgabe auf dem Logserver.

3.2 dde_fbsd – eine Umgebung für FreeBSD-Treiber

Der Entwurf des FreeBSD DDEs beinhaltet neben der grundsätzlichen Architektur die Entscheidung, inwiefern Legacy- also FreeBSD-Code wiederverwendet werden sollte. Desweiteren stellt sich die Frage nach der Abbildung von FreeBSD-Kernthreads auf L4-Threads. Architektur und Abbildung der Threads werden in den nächsten beiden Abschnitten betrachtet. Abschnitt 3.3 behandelt die Wiederverwendung von Legacy-Quelltext.

3.2.1 Architektur

Das dde_fbsd implementiert eine Treiberumgebung für FreeBSD-Gerätetreiber. Es basiert auf dem ddekit und hat außer ihm keine weiteren Abhängigkeiten. Abbildung 3.3 zeigt den Aufbau von dde_fbsd. Die Gliederung erfolgt wie bei dde_linux: Es besteht aus einer gemeinsamen allgemeinen Bibliothek, welche die von allen Treibern genutzte Grundfunktionalität beinhaltet, und aus mehreren spezifischen Bibliotheken – je einer pro Geräteklasse. Indem nur die nötigen Bibliotheken gelinkt werden, muss zum Beispiel ein IDE-Treiber-Server nicht das für ihn verzichtbare Audio-Framework enthalten. Allerdings ist es auch möglich, die verschiedenen spezifischen

Teile beliebig zu kombinieren, um so einen Server, der mehrere verschiedenartige Treiber beinhaltet, herstellen zu können.

3.2.2 Threads und Scheduling

Für die FreeBSD-Threads sind zwei verschiedene Varianten der Abbildung auf ddekit-also L4-Threads denkbar. Die erste Möglichkeit ist, nur einen Thread für das dde_fbsd zu erzeugen und den FreeBSD-Scheduler zu übernehmen. Dieser würde wie in einem nicht modifizierten FreeBSD arbeiten; nur die Kontextwechsel müssten angepasst werden. Das entspräche dann der Ausführung auf einer Ein-Prozessor-Maschine. Ebenso ist es auch möglich zwei oder mehr Threads zu erzeugen, die dann jeweils zwei oder mehr CPUs entsprechen würden. Ein Nachteil dieser Vorgehensweise ist, dass der Scheduler regelmäßig aktiviert werden muss und Rechenzeit benötigt. Außerdem sind Probleme beim Durchführen von IPC zu erwarten: ein wartender, empfangender Thread stößt nicht das FreeBSD-Scheduling an, sondern das von L4. Obwohl andere FreeBSD-Threads möglicherweise bereit sind, wird die CPU freigegeben. Wenn dann später der FreeBSD-Scheduler aktiv wird und vom wartenden Thread zu einen neuen wechselt, wird das Warten auf IPC abgebrochen. Diesen Problemen bei der Nutzerland-Implementierung von Threads widmet sich unter anderem (Cla05). Darin wird die Erweiterung der L4-Schnittstelle vorgeschlagen, da keine der bisherigen L4-Versionen die für Nutzerland-Threads notwendige Unterstützung bietet.

Die zweite Möglichkeit ist, jeden FreeBSD-Thread auf einen ddekit-Thread abzubilden. Ein eigener Scheduler für das dde_fbsd entfällt dann – für das Scheduling der Threads ist L4 verantwortlich. Der Nachteil dieser Methode ist, dass einige Eigenschaften des FreeBSD-Schedulers möglicherweise nur unzureichend abgebildet werden können. Dies ist bei der Funktion `sched_switch(..)` der Fall, welche die Ausführung des laufenden Threads unterbricht. Ihr kann ein Parameter übergeben werden, der den als nächstes zu aktivierenden Thread angibt. Diese Funktionalität steht mit der verwendeten L4-Implementierung aber nicht zur Verfügung. Sie wird jedoch in FreeBSD nur in den Scheduler-Implementierungen selbst benutzt – auf eine Unterstützung dafür kann also im DDE verzichtet werden. Da bei FreeBSD keine weiteren Probleme offensichtlich sind, fällt die Entscheidung zugunsten der 1:1-Abbildung von FreeBSD- auf ddekit-Threads.

3.3 Wiederverwendung von Legacy-Quelltext

Da FreeBSD frei verfügbar ist und im Quelltext vorliegt, können die Treiber-Quellen verwendet werden. Es muss nicht, wie bei NDISulator, der kompilierte Treiber zum DDE hinzugelinkt werden, was einen höheren Entwicklungsaufwand bedeuten würde. Da nicht nur die Treiber, sondern der gesamte Kern frei im Quelltext vorliegt, bietet es sich an, in Betracht zu ziehen, die Implementierung einiger Kernkomponenten von FreeBSD für das DDE zu übernehmen. Auf die beiden Extreme, die nahezu vollständige und die nur sehr geringe Wiederverwendung des Legacy-Kerns, stößt man bei DD/OS beziehungsweise bei dde_linux.

3.3.1 Wiederverwendung bei DD/OS

Bei einem DD/OS (Abschnitt 2.3.2, (LU04)) in einer virtualisierten Umgebung wird der Legacy-Kern vollständig und unverändert übernommen. Bei einem paravirtualisierten DD/OS kann dieser auf die vom enthaltenen Gerätetreiber benötigten Komponenten reduziert werden. Der größte Vorteil des DD/OS-Ansatzes ist, dass die Treiber in nahezu ihre originale und damit authentische Umgebung eingebettet werden und somit kaum Inkompatibilitäten entstehen können. Ein Problem ist die Größe der Treiberumgebung, beziehungsweise die höhere Nutzung des Arbeitsspeichers insgesamt: Da die verschiedenen Treiber eines Systems sich zur Laufzeit gegenseitig nicht oder nur möglichst wenig beeinflussen können sollen, läuft jeder als eigenständiger Prozess. Mit der Anzahl an DD/OS-Instanzen vervielfacht sich somit auch die Speicherauslastung. Dem wird in (LUSG04) auf mehrere Weisen begegnet – unter anderem indem sich mehrere Instanzen eines DD/OS durch gemeinsam benutzten Speicher den gleichen Kern teilen, oder indem Speicherseiten, die nicht dem Working-Set angehören auf die Festplatte ausgelagert ausgelagert oder im Speicher komprimiert werden. Durch diesen zusätzlichen Entwicklungsaufwand kann die Speicherauslastung reduziert werden: Das Working-Set eines passiven Linux 2.6-DD/OS umfasst ca. 144kB, das eines aktiven mit einem Netzwerkkartentreiber ca. 2200kB.

3.3.2 Wiederverwendung bei dde_linux

Bei der Implementierung von dde_linux (Abschnitt 2.3.1) wurde dagegen selten der Linux-Quelltext wiederverwendet – es wurde ein möglichst schlanker Emulationslayer entwickelt. Der größte Vorteil dieser Methode ist entsprechend auch der geringere Umfang des DDEs. Bei der Implementierung konnte darauf eingegangen werden, dass nur ein oder wenige Treiber in einer Umgebung laufen. Der daraus entstehende Nachteil ist, dass ein erhöhter Aufwand nötig ist, um die Kompatibilität zum Legacy-Kern sicherzustellen. Dies ist zum einen problematisch, wenn Treiber die Schnittstellen zum Kern nicht immer einhalten – zum Beispiel Datenstrukturen direkt verändern, anstatt die dafür vorgesehene Methode aufzurufen – oder Seiteneffekte der jeweiligen Implementierung ausnutzen. Zum anderen können Inkompatibilitäten entstehen, wenn die Implementierung im Legacy-System nicht genau ihrer Beschreibung entspricht.

3.3.3 Mittelweg

Ziel dieser Arbeit ist es nun, einen Weg zu finden, der die Vorteile der beiden Ansätze vereint und dabei ihre Nachteile beschränkt. Man will also eine hohe Wiederverwendung des Legacy-Quelltextes, um die Kompatibilität zum Legacy-System sicherzustellen und den Implementierungsaufwand zu verringern, ohne dabei das DDE unnötig aufzublähen.

FreeBSD ist in einzelne Komponenten unterteilt. Eine Darstellung, wie einige der Komponenten voneinander abhängen, zeigt Abbildung 3.4. Ein Pfeil beginnt dabei immer bei der abhängigen Komponente und zeigt dahin, worauf sich die Abhängigkeit bezieht. Die Malloc-, Semaphore- und die Mutex-Komponente werden da-

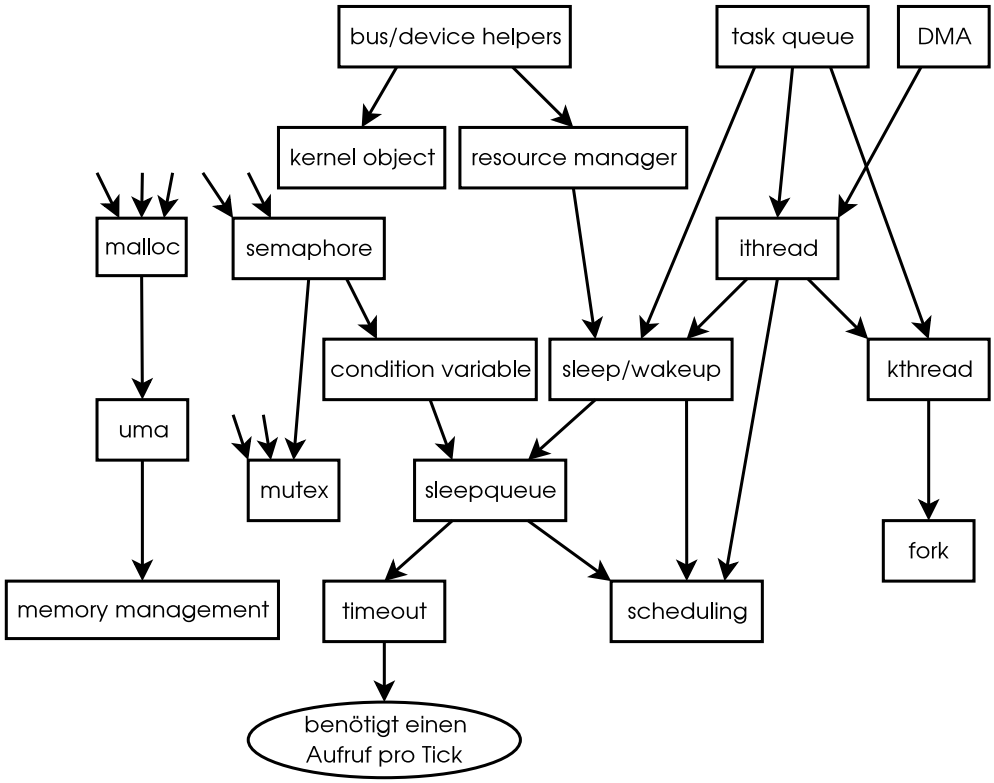

Abbildung 3.4: Abhängigkeitsgraph ausgewählter FreeBSD-Komponenten. Pfeile ohne Ursprungskomponente sollen die Abhängigkeit sehr vieler Komponenten darstellen.

bei jeweils von fast allen anderen im Bild benutzt, worauf die zusätzlichen, bei ihnen endenden Pfeile hindeuten sollen.

Man erkennt in Abbildung 3.4, dass die FreeBSD-Komponenten untereinander stark vernetzt sind. Doch durch deren saubere Trennung voneinander, kann die Entscheidung für oder gegen die Verwendung der Legacy-Implementierung für jede Komponente einzeln gefällt werden. Wie diese Entscheidung für die gezeigten Komponenten ausfällt, wird in Abschnitt 4.3 beschrieben.

Gründe für die Wiederverwendung

Für die Verwendung der Legacy-Implementierung sollte sich entschieden werden, wenn

- die Implementierung der darunter liegenden Komponenten, also von denen die betreffende abhängig ist, einfacher scheint, da diese weniger komplex sind, oder
- die darunter liegenden Komponenten aus anderen Gründen für das DDE benötigt werden, indem sie zum Beispiel von Treibern direkt genutzt werden.

Die Speicherallokation mittels malloc basiert zum Beispiel nur auf UMA – einer Slab-Cache-Implementierung, also Caches für Speicherobjekte. Da Treiber auch die UMA-Komponente direkt benutzen, ist ihre Implementierung ohnehin notwendig und die Malloc-Implementierung kann von FreeBSD einfach übernommen werden.

Gründe gegen die Wiederverwendung

Gegen die Verwendung der Legacy- und damit für eine eigene Implementierung sollte sich indes entschieden werden, wenn

- dadurch eine bessere Performance erreicht werden kann. Dies kann der Fall sein, wenn eine weniger flexible und damit weniger komplexe Implementierung genügt. So wird zum Beispiel bei der Einbindung des ATA-Treibers in L4 bei Festplatten das GEOM umgangen, wodurch die I/O-Aufträge in kürzerer Zeit abgearbeitet werden können. Dies wird in Abschnitt 5.2.1 genauer beschrieben.

  Manchmal kann auch dadurch eine höhere Performance erreicht werden, dass bestimmte Eigenschaften des Zielsystems genutzt werden. Beispielsweise basiert die FreeBSD-Implementierung von Timeouts darauf, dass 100 bzw. 1000 Mal pro Sekunde eine Methode aufgerufen wird. Eine eigene Implementierung kann viel effizienter vorgehen. In Abschnitt 4.3.4 wird genauer darauf eingegangen.

- die Legacy-Implementierung Abhängigkeiten zu weiteren Teilen des Legacy-Kernels nach sich zieht, deren Implementierung dann ebenfalls betrachtet werden müsste, aber sonst keine Verwendung findet. Das würde nur den Entwicklungsaufwand und die Größe des DDEs erhöhen. Dies ist zum Beispiel bei der FreeBSD-Implementierung von Kernthreads der Fall: Sie basieren auf fork, das sonst nicht von Treibern verwendet wird. Eine Implementierung von fork anstelle der Kernthreads würde das DDE nur unnötig vergrößern.

3.3.4 Fazit

Der Übergang zwischen Legacy-Implementierungen und eigenen sollte an der Stelle stattfinden, an der die Interfaces die geringste Komplexität aufweisen, wobei eine Abwägung zwischen dem Aufwand der Implementierung und der Größe des entstehenden DDEs stattfinden sollte. Oder anders ausgedrückt: Die eigene Implementierung sollte möglichst nah am Treiber stattfinden – unter Verwendung von Legacy-Implementierungen, wo diese sich anbieten –, aber so weit entfernt vom Treiber wie nötig, um Kompatibilität sicherzustellen und den Implementierungsaufwand zu minimieren.

Für die Entscheidung zugunsten oder gegen Wiederverwendung bleibt in vielen Fällen einiger Spielraum. Eine eindeutige Lösungsformel kann aufgrund der Komplexität hier nicht gefunden werden. Meist sollte wohl zuerst die Variante gewählt werden, die sich mit dem geringsten Aufwand umsetzen lässt und erst danach und im Anschluss an eine Analyse des Ergebnisses sollte man die Bereiche optimieren, die den größten Erfolg versprechen.

In Abschnitt 4.3 werden einige dieser Entscheidungen erläutert, die bei der Entwicklung von dde_fbsd getroffen wurden.

3.4 Fehlerisolation

In diesem Abschnitt wird der Umgang mit einigen sicherheitsrelevanten Ressourcen betrachtet, die im Entwurf behandelt wurden: I/O-Ports, I/O-Speicher, Interrupts und Speicherschutz. Ein fehlerhafter Treiber sollte durch sein Verhalten andere Tasks im System nicht beeinträchtigen oder gar zum Absturz bringen können. Weitergehende Sicherheitsaspekte, wie Vertraulichkeit oder Verfügbarkeit, sollen aufgrund der Komplexität des Themas außen vor bleiben.

Die hier gezeigten Probleme beziehen sich auf das gesamte Betriebssystem und können oder sollen nicht im DDE behandelt werden. Der Vollständigkeit halber werden sie aber trotzdem hier erwähnt.

3.4.1 Gerätespezifische Ressourcen

I/O-Ports und -Speicher

Jedes Gerät kann über mehrere I/O-Port- und -Speicher-Bereiche verfügen, über die darauf zugegriffen werden kann. Die Zugriffe können für beide Ressourcentypen beschränkt werden ((IA32)): Für eine Zugriffsbeschränkung auf I/O-Ports kann die so genannte I/O-Permission-Bitmap benutzt werden. Dabei handelt es sich um eine Datenstruktur, durch die dem Prozessor für jeden Port einzeln mitgeteilt werden kann, ob der Zugriff erlaubt ist oder nicht. Auf I/O-Speicher-Bereiche kann dagegen nur zugegriffen werden kann, wenn diese im virtuellen Adressraum einer Task eingeblendet sind. Daher lässt sich hier eine seitengranulare Zugriffsbeschränkung durchsetzen.

Ein fehlerhafter Treiber kann durch Zugriffe auf I/O-Ports- und -Speicher eines fremden Geräts dieses steuern und eine korrekte Funktionsweise des Geräts und des zu-

gehörigen Treibers verhindern. Daher sollte ein Treiber nur auf die Ressourcen der von ihm verwalteten Geräte zugreifen können.

Interrupts

Geräte können bei Statusänderungen Interrupts auslösen, um den Treiber darüber zu informieren. Teilen sich mehrere Geräte eine Interrupt-Leitung (*IRQ-Sharing*), müssen alle zugehörigen Treiber bei einem Interrupt informiert werden, da nicht festgestellt werden kann, welches Gerät den Interrupt ausgelöst hat.

Um zu verhindern, dass sich Treiber gegenseitig beeinflussen, sollten die Geräte möglichst so konfiguriert werden, dass kein IRQ-Sharing stattfindet. Falls dies nicht möglich ist, muss verhindert werden, dass ein fehlerhafter Treiber dadurch, dass die Ausführung seiner Interruptbehandlungsroutine sehr lange dauert, die Funktionsweise anderer Treiber beeinträchtigen kann.

Handhabung unter DROPS

Die drei Ressourcen I/O-Ports, I/O-Speicher und Interrupts werden in DROPS von L4IO verwaltet. Dabei handelt es sich um einen Server, dem beim Systemstart alle Ressourcen zugewiesen werden und der die entsprechenden Rechte an andere Threads weitergeben kann. Die bisherige Implementierung sorgt nur dafür, dass keine Zuweisung doppelt erfolgt, also nicht zwei Treiber auf die gleiche Ressource zugreifen können. An dieser Stelle muss eine anspruchsvollere Kontrolle beziehungsweise Zuordnung durchgesetzt werden. Jeder Treiber sollte nur die Rechte zum Zugriff auf die Ressourcen der von ihm verwalteten Geräte bekommen. Die notwendigen Mechanismen stellt Fiasco bereits zur Verfügung.

L4IO übernimmt ebenfalls die Verwaltung und Konfiguration der PCI-Geräte. Dabei kann jeder Treiber auf alle Geräte zugreifen und diese konfigurieren. Hier sollte eine Zuordnung von Gerät zu Treiber stattfinden, die verhindert, dass fehlerhafte Treiber auf fremde Geräte zugreifen können. Desweiteren sollte jeder Konfigurationsvorgang dahingehend überprüft werden, ob damit die Funktionsweise anderer Geräte beeinträchtigt werden könnte.

3.4.2 Speicherschutz

Verschiedene Gerätetreiber laufen als unterschiedliche Tasks, also in getrennten Adressräumen. Dadurch kann kein Treiber durch Fehlfunktion oder Absicht auf den Speicher eines anderen Treibers zugreifen, außer auf Speicherbereiche, welche sich die Treiber bewusst teilen. Eine Gefahr stellt allerdings die Fähigkeit der PCI-Geräte dar, auf den gesamten physischen Speicher per DMA zugreifen zu können. Dies lässt sich mit Hilfe von I/O-MMUs einschränken, die im nächsten Abschnitt beschrieben werden. Im darauffolgenden Abschnitt wird eine Lösung für Architekturen ohne I/O-MMU erläutert.

I/O-MMUs

Hier kann die Verwendung von I/O-MMUs Abhilfe schaffen. Ihre eigentliche Bestimmung ist die Abbildung der 32-bit-Adressen des PCI-Bus auf 64-bit-Adressen des Speicherbusses. Die Arbeitsweise einer I/O-MMU ähnelt der einer Prozessor-MMU: Zur Abbildung benutzt sie Seitentabellen im Arbeitsspeicher und speichert die Ergebnisse zum schnelleren Zugriff in einem TLB (translation look-aside buffer) zwischen.

Wie in (LeHe03) dargestellt, kann dieser Mechanismus auch dazu benutzt werden, den PCI-Geräten nur Zugriff auf bestimmte Bereiche des Hauptspeichers zu ermöglichen. Jedoch kann bei der beschriebenen Methode jedes Gerät auch auf die Speicherbereiche der anderen zugreifen. In (LUSG04) wird dafür ein Lösungsansatz dargestellt: Die I/O-MMU wird jeweils für eine Zeitscheibe für nur ein Gerät programmiert und der PCI-Bus so eingestellt, dass nur dieses Gerät zugreifen darf. Ein Scheduling-Mechanismus, unabhängig vom Task-Scheduling, übernimmt die Umschaltung zwischen den Geräten und die Zuordnung der Zeitscheiben. Dies funktioniert jedoch ohne Kenntnis der beteiligten Geräte nicht immer fehlerfrei. Eine bessere Lösung wäre, wenn sich die I/O-MMU für jedes Gerät einzeln programmieren ließe oder für jedes Gerät eine eigene I/O-MMU zur Verfügung stehen würde.

Mediatoren

In (HLM+03) wird eine Lösung für Architekturen ohne I/O-MMU vorgestellt. Hier überwacht ein *Emulator* die Zugriffe des Treibers auf I/O-Ports und eingeblendeten I/O-Speicher. Treiber und Emulator besitzen nicht die nötigen Rechte, um selbst auf I/O-Ports oder I/O-Speicher zuzugreifen. Der Emulator fängt jeden Zugriff ab, interpretiert ihn, und überträgt ihn an den *Mediator*. Dieser überprüft, ob der Zugriff erlaubt ist, und führt ihn gegebenenfalls aus. Werden mit dem Zugriff Speicheradressen für DMA übertragen, werden diese ebenfalls untersucht. An dieser Stelle lässt sich also jeder DMA-Zugriff überprüfen und gegebenenfalls verhindern.

Neben der Beschränkung der Zugriffsrechte auf I/O-Ports und -Speicher ist hier der Mediator die einzige Komponente, der vertraut werden muss. Der Nachteil dieses Ansatzes ist, dass Emulator und Mediator geräte- oder geräteklassenspezifisch sind, da sie die I/O-Zugriffe interpretieren beziehungsweise überprüfen müssen.

Kapitel 4

Implementierung

4.1 Trennung ddekit/dde_fbsd

Bei der Implementierung zeigte sich, dass die entwurfsmäßige Trennung von ddekit und dde_fbsd auch rein technische Vorteile hat. Während der Entwicklung wurden zwischenzeitlich einige DROPS-Funktionen direkt im dde_fbsd-Quelltext benutzt. Das brachte an einigen Punkten Schwierigkeiten mit sich, da eine handvoll bekannter Funktionen, wie zum Beispiel malloc(), in FreeBSD mit einer gegenüber der üblichen veränderten Parameterliste definiert sind. Da FreeBSD in C implementiert ist, stehen zur Lösung auch keine Namensräume zur Verfügung.

Die Headerdateien des ddekits binden deshalb keine L4- oder Standard-Header ein. So können sie beliebig im dde_fbsd-Quelltext benutzt werden, ohne Interferenzen zu erzeugen. In dde_fbsd werden also auch aus technischen Gründen nur die FreeBSD- und ddekit-Header verwendet.

4.2 ddekit

Jegliche vom ddekit angebotene Funktionalität wird auf das Common L4 Environment (Abschnitt 2.2, (L4Env)) abgebildet. Die Implementierung beschränkt sich daher in den meisten Fällen auf eine triviale Umsetzung der Funktionsaufrufe. Die Ausnahmen umfassen die Bedingungsvariablen, die auf Semaphoren und Mutexe abgebildet werden, die Verwaltung von Seitentabellen, die selbst implementiert wird und die Interruptbehandlung, da sich hier die Schnittstellen von ddekit und L4Env stark unterscheiden. Ein interessanter Aspekt findet sich bei der Abbildung der Slab-Caches, auf den im nächsten Abschnitt eingegangen wird.

4.2.1 Thrashing bei Slab-Caches

Erste Testläufe mit dem FreeBSD-ATA-Treiber zeigten eine sehr schlechte Performance im Vergleich mit dem Linux-Treiber und dde_linux. Das konnte schließlich auf eine sehr hohe Zahl an IPC-Vorgängen zwischen Treiber und dem Speicherserver dm_phys zurückgeführt werden. Es wurden sehr oft Speicherseiten angefordert und wieder freigegeben. Der Grund dafür war folgender:

FreeBSD implementiert malloc() mittels Slab-Caches in einer Größe von 16, 32, 64 und so weiter bis 2048 Byte und es werden im beschriebenen Szenario auch im ATA-Treiber und im GEOM je ein Slab-Cache benötigt – insgesamt also zehn Slab-Caches. Die ddekit-Implementierung von Slab-Caches benutzt eine Bibliothek, welche bereits zur Verfügung steht. Diese ermöglicht es für jeden Slab-Cache festzulegen, wie viele wieder frei gewordene Speicherseiten behalten werden, beziehungsweise ab welchem Schwellwert freie Seiten dm_phys wieder zugeführt werden. Um die Speicherbenutzung nicht unnötig zu erhöhen, wurde die Grenze auf Null gesetzt - es sollte also jede frei werdende Seite sofort zurückgegeben werden. Offensichtlich wurden aber bei einem I/O-Auftrag mehrere neue Seiten benötigt, die nach der Bearbeitung wieder freigegeben wurden um beim nächsten Auftrag neu allokiert zu werden. Dieser Vorgang ist mit Thrashing vergleichbar: im engeren Sinn wird damit der Effekt bezeichnet, bei dem ein System mit wenig Arbeitsspeicher sehr oft Speicherseiten auf Festplatte auslagern und andere wieder einlagern muss und dabei viel Rechenzeit verloren geht.

Eine Erhöhung des Schwellwerts auf je eine Seite pro Slab-Cache wäre hier zwar schon ausreichend gewesen, in anderen Szenarien kann man Thrashing aber möglicherweise erst ab zwei, drei oder noch mehr Seiten pro Slab-Cache unterdrücken, was nicht sehr ressourcenschonend ist. Daher entschied ich mich für einen zwischen allen Caches geteilten Seitenpool mit einem globalen Schwellwert. Die Implementierung verwendet zwei einfach verkettete Listen und atomares Compare-Exchange, und funktioniert so ohne die Benutzung von Locks. Desweiteren kann sie auch leicht so erweitert werden, dass der Schwellwert dynamisch angepasst wird.

4.3 dde_fbsd

Die Entwicklung des allgemeinen Teils von dde_fbsd erfolgte mit der Zielsetzung, alle unspezifischen Abhängigkeiten des ATA-Treibers zu erfüllen, die auch bei vielen oder allen anderen Treibern zu erwarten sind. Diese Beschränkung auf den ATA-Treiber ist dabei nicht sehr groß, da sich, wie in Abschnitt 4.3.9 beschrieben, weitere benötigte allgemeine FreeBSD-Komponenten meist einfach zum dde_fbsd hinzufügen lassen. Die klassenspezifischen Abhängigkeiten der Treiber werden generell in eigene Bibliotheken ausgelagert.

In Abschnitt 3.3.3 wurden die Vor- und Nachteile der Wiederverwendung von Legacy-Implementierungen erörtert. Hier sollen nun die Entscheidungen für einzelne FreeBSD-Komponenten erläutert und gegebenenfalls auf Besonderheiten der Implementierung eingegangen werden. In Abbildung 3.4 wurde eine Auswahl von Komponenten und deren Abhängigkeiten untereinander dargestellt. Abbildung 4.1 zeigt für diese, welche Komponenten durch eine eigene Implementierung ersetzt wurden und als Adapter zwischen dem Legacy- und dem Zielsystem dienen.

4.3.1 Synchronisationsmechanismen

Mutexe werden direkt auf ihre ddekit-Pendants abgebildet. Wie Abbildung 4.1 zeigt kann die FreeBSD-Semaphore-Implementierung wiederverwendet werden. Da sie

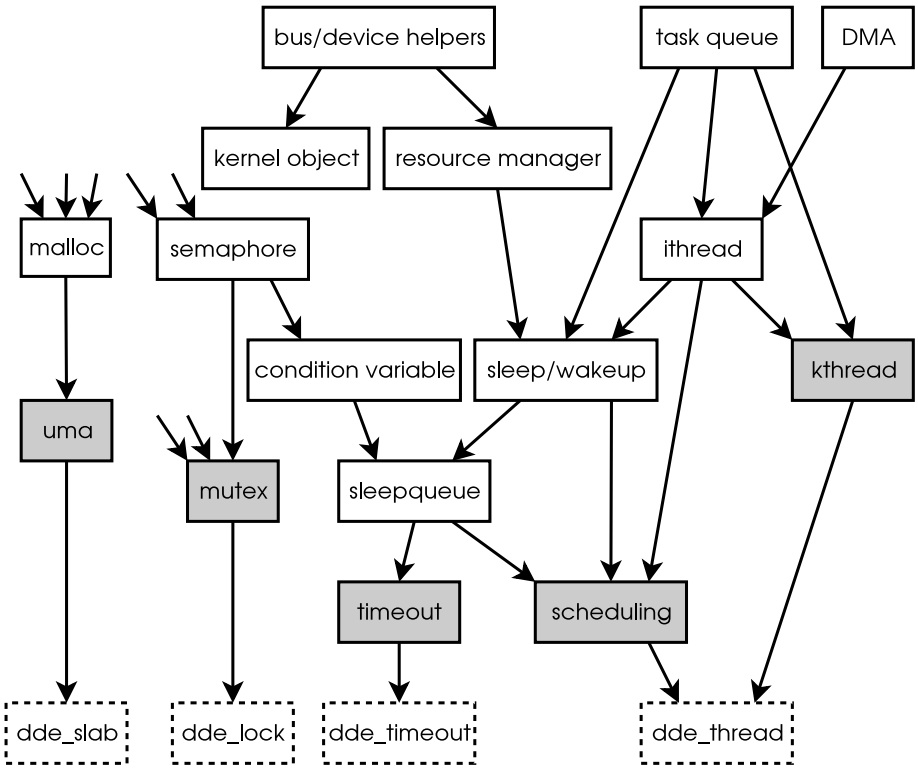

Abbildung 4.1: Abhängigkeitsgraph ausgewählter FreeBSD-Komponenten und welche Schnittstellen zur Abbildung auf das ddekit ausgewählt wurden (grau unterlegt)

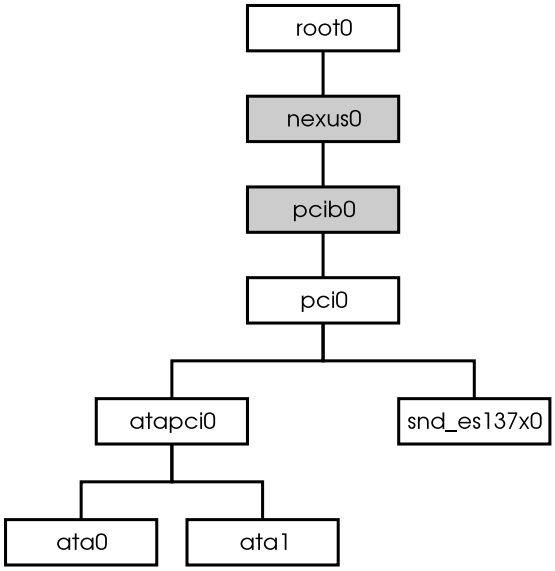

Abbildung 4.2: dde_fbsd Gerätehierarchie mit ATA- und Audiotreiber, die neu zu implementierenden Treiber – Nexus und PCI-Bustreiber – sind grau unterlegt

auf Mutexen basiert, ist möglicherweise eine direkte Abbildung auf ddekit-Semaphore performanter – das kann später als Ansatzpunkt zur Optimierung dienen. Mit den Bedingungsvariablen (condition variables) verhält es sich genauso.

Alle diese Mechanismen benutzen das so genannte „Witness" (engl. Zeuge) – eine Komponente zur Überwachung der Reihenfolge, in der die Synchronisationsmechanismen benutzt werden, um mögliche Deadlocks frühzeitig zu erkennen. Bei einer Reimplementierung sollte dies möglichst erhalten bleiben.

4.3.2 Geräte-/Treiberverwaltung

Die Verwaltung der Geräte und Treiber in FreeBSD (wie in den Abschnitten 2.1.1 und 2.1.2 beschrieben) wird unverändert übernommen. So ist ein problemloses Zusammenarbeiten mehrerer Treiber garantiert. Die Ressourcenverwaltung durch Nexus – also die Zuordnung von I/O-Ports, I/O-Speicher, DMA und Interrupts – muss neu implementiert werden. Sie wird durch ein Nexus, welches die Reservierung und Freigabe der Ressourcen beim ddekit vornimmt, ersetzt. In Abbildung 4.2 findet es sich unter der Bezeichnung nexus0 wieder.

Der Zugriff auf den PCI-Bus wird ähnlich realisiert: Es wird ein neuer low-level PCI-Bustreiber implementiert (pcib0 in Abbildung 4.2). Dieser bildet die Aufrufe zum Durchsuchen des PCI-Busses und Lesen/Schreiben der Konfigurationsregister auf die vom ddekit bereitgestellten Funktionen ab. Desweiteren erzeugt er einen *pci*-Kindknoten, dem bei der Autokonfiguration der high-level PCI-Bustreiber zugeordnet wird.

Der low-level PCI-Bustreiber verknüpft sich direkt mit dem Nexus. Auf die Zwischenebene *legacy* wird verzichtet, da sie momentan nicht benötigt wird und später auch einfach eingefügt werden kann, falls sich das als notwendig herausstellt.

4.3.3 Sysinits

Sysinits sind mit ddekit-Initcalls vergleichbar. Sie dienen dazu, verschiedene Module von FreeBSD zu initialisieren. Zusätzlich wird die Reihenfolge der Aufrufe durch zusätzliche Prioritätsparameter des `SYSINIT(..)`-Makros festgelegt. Die Abbildung auf Initcalls geschieht wie folgt: Für jedes Sysinit wird eine Struktur und eine Funktion erzeugt. Die Struktur enthält die Sysinit-Priorität und einen Zeiger auf die aufzurufende Funktion. Die Funktion ruft `bsd_register_sysinit(..)` auf und übergibt einen Zeiger auf die erzeugte Struktur als Parameter. Dadurch wird eine sortierte Liste der Sysinits aufgebaut. Mit Hilfe der Funktion `bsd_call_sysinits()` wird diese Liste abgearbeitet und die Sysinits werden in der gewünschten Reihenfolge ausgeführt.

4.3.4 Timeouts

Die FreeBSD-Implementierung von Timeouts könnte übernommen werden. Sie basiert aber darauf, dass jeden Tick – im Normalfall aller 10 ms – eine Funktion aufgerufen wird, die prüft ob Timeouts eingetreten sind. Das erzeugt eine unnötige CPU-Last, die sich bei mehreren dde_fbsd-Instanzen vervielfacht.

Die neue, eigene Implementierung benutzt einen Mutex zum Schutz der verwendeten Datenstrukturen und eine Bedingungsvariable zum Warten. Die Timeoutbehandlung findet in einem eigenen Thread statt. Nachdem alle aktuellen Timeouts behandelt wurden, legt dieser sich mittels der Bedingungsvariable für die Zeit zum nächsten anstehenden Timeout schlafen. Wird ein Timeout modifiziert oder neu erstellt, so dass dieser jetzt der nächste anstehende ist, wird der Timeoutthread mittels der signal(..)-Methode der Bedingungsvariable aufgeweckt. Der Thread behandelt dann wiederum alle momentanen Timeouts, berechnet die Zeit bis zum nächsten anstehenden Timeout und legt sich erneut schlafen.

4.3.5 Threads und Scheduling

Wie in Abschnitt 3.2.2 erläutert werden FreeBSD-Threads 1:1 auf DDE- und damit auf L4-Threads abgebildet. Aus FreeBSD-Sicht wird somit jeder Thread auf einem eigenen Prozessor ausgeführt. Die Scheduler-Aufrufe, um einen Thread schlafen zu legen oder aufzuwecken, lassen sich komfortabel auf `dde_thread_sleep(..)` beziehungsweise `dde_thread_wakeup(..)` abbilden. Ein Zeiger auf die FreeBSD-Verwaltungsdatenstruktur des aktuellen Threads wird als DDE-Thread-lokales Datum abgelegt.

4.3.6 Speicherverwaltung

Wie in Abschnitt 3.3.3 bereits angedeutet, kann die FreeBSD-Implementierung von malloc übernommen werden. Sie basiert nur auf UMA – FreeBSDs Implementierung von Slab-Caches. Diese werden neu implementiert und auf ddekit-Slab-Caches abgebildet.

4.3.7 Sonstiges

Die Registrierung von Interrupt-Handlern wird auf das ddekit abgebildet. Dabei wird zusätzlich die Registrierung mehrerer Handler für einen Interrupt unterstützt, die dann hintereinander aufgerufen werden. Für `(s-)printf(..)` wird die FreeBSD-Implementierung leicht modifiziert übernommen, da sie zusätzliche, in FreeBSD verwendete Formatcodes zur Verfügung stellt. Der vollständig bearbeitete String wird dann zur Ausgabe an das ddekit weitergereicht.

4.3.8 Dummy-Implementierungen

Einige wenige Komponenten wurden (vorerst) nicht vollständig implementiert, da sie im ATA-Treiber-Szenario nicht verwendet werden und somit auch nicht die Möglichkeit besteht, sie anhand dessen zu testen, oder weil sie nicht benötigt werden. Das betrifft zum einen die Verwaltung von Geräten mittels devfs, also die Bereitstellung der logischen Geräte im Dateisystem unter /dev. Dies wird sicherlich sehr früh bei der Erweiterung auf andere Geräteklassen benötigt. Desweiteren wurde auf die Schnittstelle zum Lesen und Verändern von Kernparametern zur Laufzeit (sysctl) und das Sammeln von zufälligen Daten zur Zufallszahlengenerierung verzichtet.

4.3.9 Weitere Komponenten

Auf Basis der bis hierhin dargestellten grundlegenden Komponenten wurden alle weiteren ohne oder mit nur sehr geringen Änderungen aus FreeBSD übernommen. Das betrifft unter anderem die Task-Queues, Sleep-Queues und DMA-Hilfsroutinen. Aus dieser Erfahrung lässt sich abschätzen, dass sich auch später zusätzlich notwendig werdende Komponenten ohne Schwierigkeiten übernehmen lassen sollten.

Kapitel 5

Beispielanwendung: ATA-Treiber-Server

Als Beispiel zur Benutzbarkeit von ddekit und dde_fbsd, und als Möglichkeit zur Leistungsbewertung im Vergleich zu dde_linux in Abschnitt 6.2 ist als erste Anwendung ein ATA-Treiber-Server entstanden.

5.1 Entwurf

Der Aufbau von l4ata wird in Abbildung 5.1 dargestellt. Zwei Komponenten sind neu zu entwickeln: einerseits die blockgerätespezifische FreeBSD-DDE-Bibliothek libdad und andererseits der eigentliche Treiber-Server l4ata.

In der libdad werden die klassenspezifischen Abhängigkeiten des FreeBSD-ATA-Treibers zusammengefasst und es wird eine einfache, FreeBSD-unspezifische Schnittstelle zum Zugriff auf die Blockgeräte angeboten. Mit Hilfe der libdad kann der ATA-Treiber zu einer Applikation hinzugelinkt werden, die dann direkt darauf zugreifen kann. So auch bei l4ata, das als bddf-Bustreiber agiert und die Abbildung der beiden Schnittstellen aufeinander vornimmt. Die Anbindung an das bddf geschieht über die von diesem bereitgestellte Client-Bibliothek. Die Kommunikation soll über IPC erfolgen, aber durch Austauschen der Client-Bibliothek sollte der Treiber auch direkt zum bddf hinzugelinkt werden können.

In den nächsten beiden Abschnitten wird auf die Implementierung von libdad und l4ata eingegangen.

5.2 libdad – Zugriff auf Blockgeräte

Die libdad bietet Zugriff auf die Blockgeräte über die dad-Schnittstelle an. Diese beinhaltet nur zwei Funktionen: `dad_set_announce_disk(*fun)` zum Setzen der Funktion, die aufgerufen werden soll, wenn ein neues Blockgerät gefunden wurde, und desweiteren `dad_put_request(*request)` zum Absetzen der Aufträge.

Der ATA-Treiber erstellt für jedes gefundene Gerät ein Geom (siehe Abschnitt 2.1.3). Für CD-Laufwerke werden Instanzen der GEOM-Klasse *acd* erstellt, die im Treiber selbst implementiert ist. Für Festplatten erstellt er indirekt durch Aufrufe von `disk_`

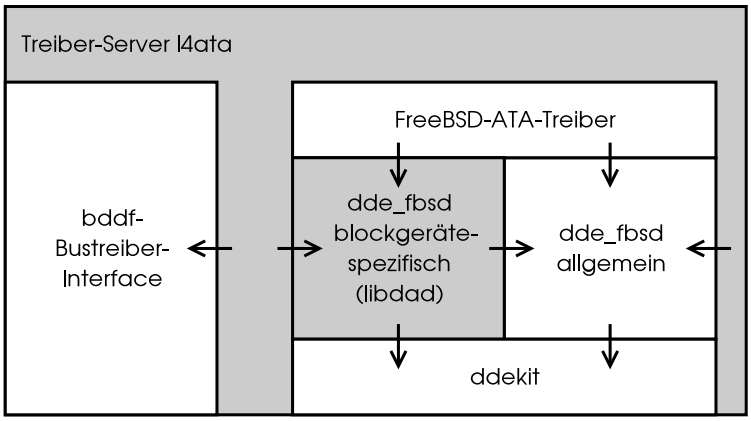

Abbildung 5.1: Aufbau des ATA-Treiber-Servers l4ata: Verknüpfung der spezifischen dde_fbsd-Bibliothek und dem bddf-Bustreiber-Interface. Die zu entwickelnden Komponenten sind grau unterlegt.

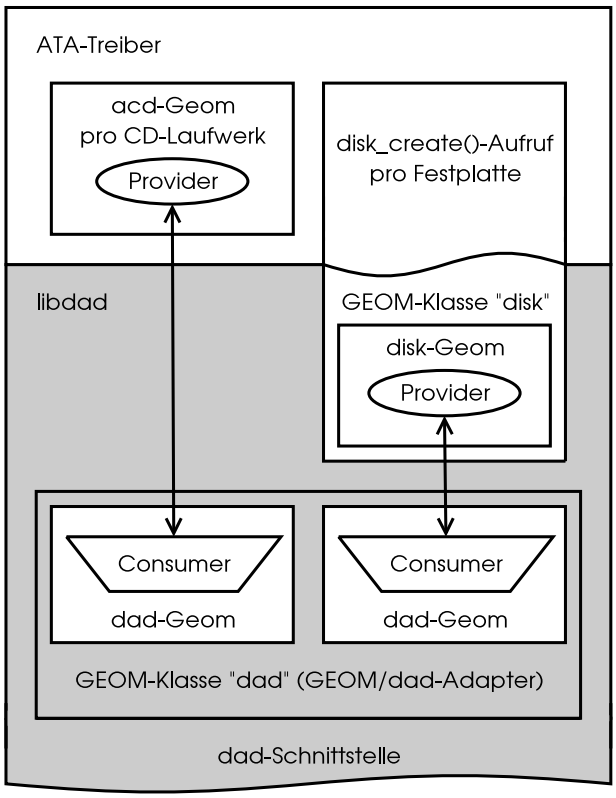

Abbildung 5.2: Aufbau der libdad: Im einfachen Fall wird über eine neue GEOM-Klasse auf den ATA-Treiber zugegriffen.

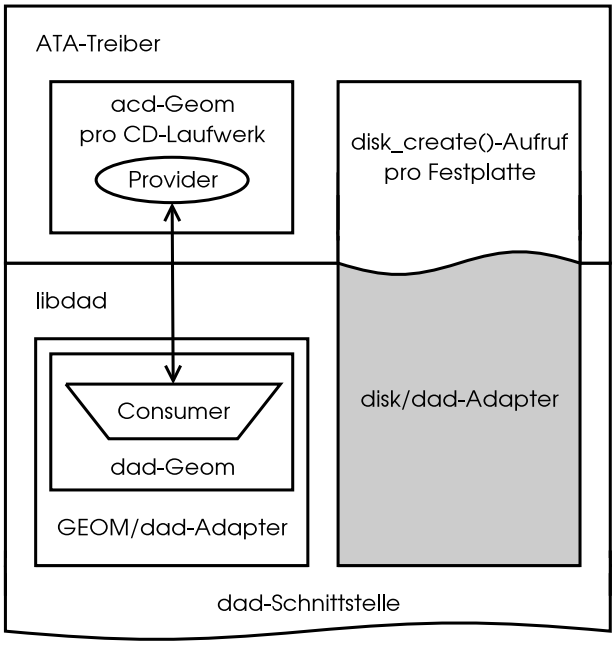

Abbildung 5.3: „Abkürzung“ für Festplattenzugriffe

`create(..)` Instanzen der Klasse *disk*. Abbildung 5.2 zeigt den Aufbau der libdad: Um auf die vom Treiber erstellten Geoms zugreifen zu können, wird in libdad eine eigene GEOM-Klasse implementiert. Bei Erstellung eines neuen Providers wird die `taste(..)`-Funktion jeder GEOM-Klasse aufgerufen, so dass diese überprüfen können, ob sie sich damit verbinden wollen. Die GEOM-Klasse *dad* erstellt für jeden Provider ein Geom und einen Consumer, und verknüpft diese miteinander.

Danach wird die mittels `dad_set_announce_disk(..)` registrierte Funktion aufgerufen und ihr die Daten des neu gefundenen Gerätes in einer Struktur vom Typ `dad_disk_t` übergeben. In der Struktur ist ebenfalls der Consumer eingetragen, mit Hilfe dessen `dad_put_request(*request)` I/O-Aufträge bedienen kann. Der Parameter `request` ist ein Zeiger auf eine Struktur, die den auszuführenden Auftrag beschreibt. Sie beinhaltet unter anderem auch die virtuelle und die physische Adresse des Quell-/Zielpuffers und einen Zeiger auf eine Funktion, die bei Beendigung des Auftrags aufgerufen werden soll.

`dad_put_request(*request)` nimmt nur eine sehr kurze Behandlung der Aufträge vor: es setzt die entsprechenden Einträge in der DDE-Seitentabelle, so dass für DMA darauf zugegriffen werden kann, erstellt und initialisiert eine neue bio (Block-I/O) - Struktur und übergibt diese zur Bearbeitung an das GEOM. Bei Beendigung des Auftrags werden die Einträge in der Seitentabelle wieder gelöscht, die bio-Struktur freigegeben und die Benachrichtigungsfunktion des Auftrags aufgerufen.

5.2.1 „Abkürzung" für Festplatten

Das GEOM wird in der libdad nur benutzt, um über gefundene Blockgeräte informiert zu werden und um somit auf sie zugreifen zu können. Zu seinem eigentlichen Zweck, zum Beispiel zur Partitionierung oder RAID-Behandlung von Festplatten, wird es nicht benötigt, da dies im bddf-Server erfolgt (Partitionierung) oder erfolgen sollte (RAID). Die Auftragsbehandlung wird allerdings durch das GEOM verlangsamt, da das Weiterleiten der Aufträge durch die Geom-Hierarchie in einem eigenen Thread erfolgt, was zusätzliche Thread-Umschaltungen nach sich zieht.

Am einfachsten lässt sich das GEOM für Festplatten umgehen, indem nicht die GEOM-Klasse disk benutzt wird, welche die Funktion `disk_create(..)` beinhaltet, sondern `disk_create(..)` selbst implementiert wird. Dort kann dann die Abbildung auf die dad-Schnittstelle direkt stattfinden. Abbildung 5.3 zeigt den neuen Aufbau.

Zum Zugriff auf CD- und andere Laufwerke wird vorerst die GEOM-Klasse dad beibehalten. Um auch bei der Auftragsbearbeitung für diese Blockgeräte das GEOM zu umgehen, kann die Funktion zur Anmeldung eines neuen GEOM-Providers `g_new_providerf(..)` entsprechend modifiziert werden. Ihr stehen alle zum Zugriff auf die Laufwerke nötigen Informationen zur Verfügung. Der Eingriff in die Funktion kann auch so stattfinden, dass die Funktionalität des GEOMs nicht davon beeinträchtigt wird.

5.3 l4ata – ein ATA-Server für bddf

Die Aufgabe des in Abbildung 5.1 dargestellten l4ata-Servers ist die Abbildung der bddf-Bustreiber- und der dad-Schnittstelle aufeinander. Er registriert neu gefundene Geräte beim bddf-Server und wartet in einer Schleife auf neue Aufträge von diesem.

Da bddf-Aufträge eine Scatter/Gather-Liste – also mehrere Puffer für einen Auftrag – beinhalten, das dad-Interface aber nur einen Puffer pro Auftrag vorsieht, müssen die bddf-Aufträge gegebenenfalls aufgebrochen und auf mehrere dad-Aufträge aufgeteilt werden. Dies resultiert daraus, dass weder GEOM, noch der ATA-Treiber Scatter/Gather anbieten. Die Auftragsschleife von l4ata erstellt also einen dad-Auftrag pro Scatter/Gather-Element.

Desweiteren wird geprüft, ob bei der Ausrichtung, dem Alignment, des übergebenen Puffers DMA möglich ist. Ist dies nicht der Fall, wird ein passender Shadow-Puffer erstellt. Dabei werden zusammenhängende Bereiche erkannt und zu einem dad-Auftrag zusammengefasst. Bei Schreib-Aufträgen werden die entsprechenden Daten in den Shadow-Puffer kopiert, bevor sie an die libdad weitergereicht werden.

Bei Beendigung eines dad-Auftrags wird überprüft ob damit alle Teil-Aufträge des zugehörigen bddf-Auftrags beendet sind und gegebenenfalls das bddf informiert. Bei Lese-Aufträgen in einen Shadow-Puffer muss dieser davor noch kopiert werden.

Kapitel 6

Bewertung

6.1 Aufwandsbewertung

Der Entwicklungsaufwand lässt sich mit Hilfe der Größe der einzelnen Pakete abschätzen. Abbildung 6.1 zeigt den Umfang des ddekits, dde_fbsds und von dde_linux anhand der Anzahl ihrer Quelltextzeilen (lines of code, LOC). Es existieren zwei Versionen von dde_linux: Eine für Linux 2.4 (*dde_linux*) und eine Version für Linux 2.6 (*dde_linux26*). Hier soll nur dde_linux betrachtet werden, da in dde_linux26 nicht alle ATA-spezifischen von den allgemeinen Komponenten getrennt sind.

Auffällig ist der Umfang der für dde_fbsd wiederverwendeten FreeBSD-Komponenten. Das folgt daraus, dass der Implementierungsaufwand möglichst gering gehalten werden soll, indem immer – falls möglich und sinnvoll – die FreeBSD-Implementierung einer Komponente wiederverwendet wurde. Außerdem wurde nicht versucht die Komponenten so klein und einfach wie möglich zu halten, wie das sicher bei einer Reimplementierung der Fall gewesen wäre.

Stattdessen sind die meisten FreeBSD-Komponenten ohne oder mit nur wenigen Änderungen übernommen worden. Diese sollten dadurch leichter auf eine neue FreeBSD-Version aktualisiert werden können. Desweiteren ist es durch die hohe Wiederverwendung möglicherweise einfacher das DDE um zusätzliche FreeBSD-Komponenten zu erweitern.

Ein weiterer Grund für den Umfang des wiederverwendeten FreeBSD-Quelltextes könnte eine hohe Komplexität der FreeBSD-Implementierungen sein – so sind zum Beispiel in FreeBSD die Timeout-Implementierung 350 Zeilen, meine eigene Reimplementierung dagegen 100 Zeilen lang. Dies alles hat zur Folge, dass das dde_fbsd mit

Paket	Umfang (gerundet)
ddekit	780 LOC
dde_fbsd	17.400 LOC davon neu: 1.400 LOC + wiederverwendet: 16.000 LOC
dde_linux	2.600 LOC

Abbildung 6.1: Umfang der einzelnen Pakete in Quelltextzeilen (lines of code, LOC)

	Rechnersystem Celeron 1700	Rechnersystem Celeron 900
Mainboard	Asus P4PE (Intel 845 PE)	Asus TUSL2-C (Intel 815EP)
Prozessor	Intel Celeron 1700 MHz	Intel Celeron 900 MHz
Arbeitsspeicher	256 MB	256 MB
Festplatte	IBM IC35L040 40 GB, 7200 rpm UDMA 5 (100 MB/s) 2048 kB Cache	Maxtor 2B020H1 20 GB, 5400 rpm UDMA 5 (100 MB/s) 2048 kB Cache

Abbildung 6.2: Konfiguration der Testrechner

einem Umfang von 17.400 Quelltextzeilen deutlich größer ist als dde_linux mit 2.600 LOC.

Betrachtet man dagegen nur die Größe des neu entwickelten Quelltexts, so ist dde_fbsd selbst inklusive ddekit mit 2.180 LOC noch deutlich (-16%) kleiner als dde_linux – ohne das wiederverwendbare ddekit sogar nur circa halb so groß. Daraus lässt sich die Vermutung ableiten, dass der Ansatz dieser Arbeit einen geringeren Aufwand fordert als der von dde_linux. Für einen genauen Vergleich müssten beide Ansätze auf das selbe Legacy-System angewendet werden. Interessant wäre auch der Vergleich von dde_linux mit einer auf dem ddekit basierenden Version dessen.

Das ddekit ist mit 780 Zeilen Quelltext recht klein. Für eine Portierung sind alle Komponenten außer den Bedingungsvariablen relevant. Somit bleiben 700 LOC, welche an das neue Zielsystem angepasst werden müssen. Der Aufwand dafür ist als sehr gering einzuschätzen. In Abschnitt 6.3.3 wird noch einmal auf die Portierbarkeit des ddekits eingegangen.

6.2 Leistungsbewertung

Zur Leistungsbewertung wurde der entwickelte ATA-Treiberserver l4ata mit dem auf dde_linux26 basierenden bddf_ide verglichen. Zur Einordnung der Ergebnisse wurden die gleichen Tests unter FreeBSD und Linux wiederholt. Im nächsten Abschnitt werden Aufbau und Ablauf der Messungen erläutert. Danach folgt die Darstellung der Messergebnisse, die anschließend in Abschnitt 6.2.5 interpretiert und bewertet werden.

6.2.1 Konfiguration und Ablauf

Alle Tests werden auf zwei unterschiedlichen Rechnern durchgeführt, deren Konfiguration in Abbildung 6.2 wiedergegeben ist. Gemessen wird der Datendurchsatz und die CPU-Auslastung bei Verwendung unterschiedlicher Blockgrößen. Die Zugriffe auf die Festplatten erfolgen lesend auf die ersten Sektoren. Damit können alle Anfragen außer der ersten aus dem platten-internen Cache übertragen werden. Die

gemessenen Zeiten beinhalten also keine Kopfbewegungen der Festplatte, die für die Bewertung der Treiberperformance auch irrelevant sind.

Es werden jeweils Blöcke von 1, 2, 4, 8 ... 4096 Sektoren gelesen. Die Sektorgröße beträgt 512 Byte. Die Lesevorgänge zwischen den Zeitmesspunkten wurden mehrere Male wiederholt: Bei den Blockgrößen von 1 bis 16 Sektoren sind dies 16384 Wiederholungen. Danach wird die Anzahl der Wiederholungen jedes Mal halbiert, also 8192 Wiederholungen bei 32 Sektoren bis 64 Wiederholungen bei 4096. Dadurch befinden sich die gemessen Zeiten immer zwischen einer und acht Sekunden – es wird also nicht zu kurz gemessen, was eine geringere Genauigkeit zur Folge hätte, aber auch nicht unnötig lang. Jede dieser Messungen wird 100 mal wiederholt.

Zur Zeitmessung wird der Timestamp-Counter, zur Messung der CPU-Auslastung zusätzlich der Performance-Counter des jeweiligen Prozessors benutzt. Zur Umrechnung der Zählerstände in Millisekunden wird bei einem Rechner jeweils immer die gleiche Konstante benutzt. Damit werden Abweichungen durch unterschiedliche Kalibrierungsergebnisse vermieden.

Unter FreeBSD und Linux läuft die Testanwendung als Nutzer-Prozess. Die Systeme werden zu den Messungen im Einbenutzermodus gestartet und die Testanwendung ist neben der Shell der einzige Prozess. Auf die Festplatte wird durch die speziellen Dateien `/dev/ad2` unter FreeBSD und `/dev/hdc` unter Linux zugegriffen. Diese werden mit den Flags `O_SYNC` und `O_DIRECT` geöffnet, so dass die zu lesenden Daten jedes Mal neu und per DMA direkt in den von der Anwendung bereitgestellten Puffer geschrieben werden. Um nicht während der Messungen konkurrierende Festplattenzugriffe hervorzurufen, werden die Messwerte während des Durchlaufs gesammelt und erst zum Schluss ausgegeben. Der FreeBSD- und der Linuxkern wurden gepatcht, damit die Instruktionen zum Auslesen des Timestamp- und des Performance-Counters (rdtsc und rdpmc) auch im Nutzerland ausgeführt werden können. Es wurde jeweils die Version verwendet, auf der das zugehörige DDE basiert: FreeBSD 5.4 und Linux 2.6.6.

Unter DROPS erfolgt die Ausgabe der Messwerte über die serielle Schnittstelle. Die Kerne der drei Systeme und alle DROPS-Anwendungen wurden durch Verwendung der entsprechenden Compiler-Schalter auf den jeweils verwendeten Prozessor optimiert. FreeBSD und Linux wurden sowohl in der Uniprozessor-, als auch in der SMP-Version getestet.

6.2.2 Messergebnisse

Die Messergebnisse sind in den Abbildungen 6.3, 6.4 und 6.5 wiedergegeben. Für Abbildung 6.5 wurden die Messungen des Celeron 900-Systems noch einmal mit der IBM-Festplatte des Celeron 1700-Systems wiederholt.

Das obere Diagramm zeigt jeweils den Datendurchsatz und die CPU-Auslastung unter dde_fbsd, FreeBSD, dde_linux26 und Linux. An jedem Punkt wird jeweils der Durchschnitt der gemessenen Werte dargestellt. Zusätzlich ist durch zwei kurze waagerechte Linien die Standardabweichung abgetragen. Da diese nie größer als 0,39 MB/s und 0,37 % ist, fallen die beiden Linien meist zusammen.

Die unteren Diagramme enthalten jeweils drei Graphen: Ein Graph (A) zeigt den Datendurchsatz unter dde_fbsd relativ zu dem unter FreeBSD, also wie nah die Leistung des DDEs an die des Legacy-Systems heran reicht. Der zweite Graph (B) zeigt

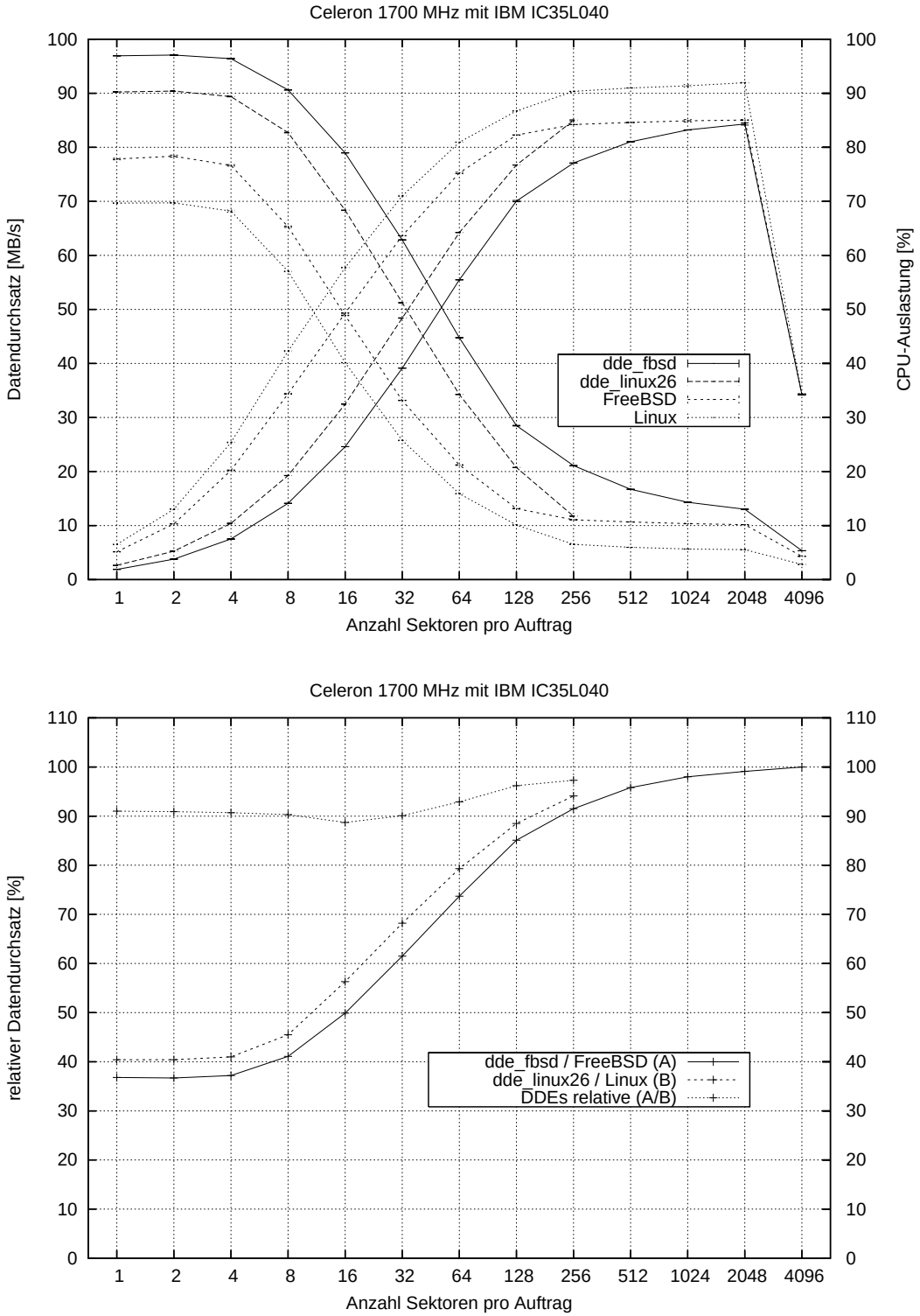

Abbildung 6.3: Leistungsvergleich auf dem Celeron 1700-System. Oben: Vergleich zwischen dde_fbsd, FreeBSD, dde_linux und Linux; die steigenden Kurven stellen den Datendurchsatz, die fallenden die CPU-Auslastung dar. Unten: Datendurchsatz der DDEs relativ zu ihren nativen Legacy-Systemen; Graph A: dde_fbsd zu FreeBSD; Graph B: dde_linux26 zu Linux; Graph A/B: A dividiert durch B – Verhältnis der relativen Leistung.

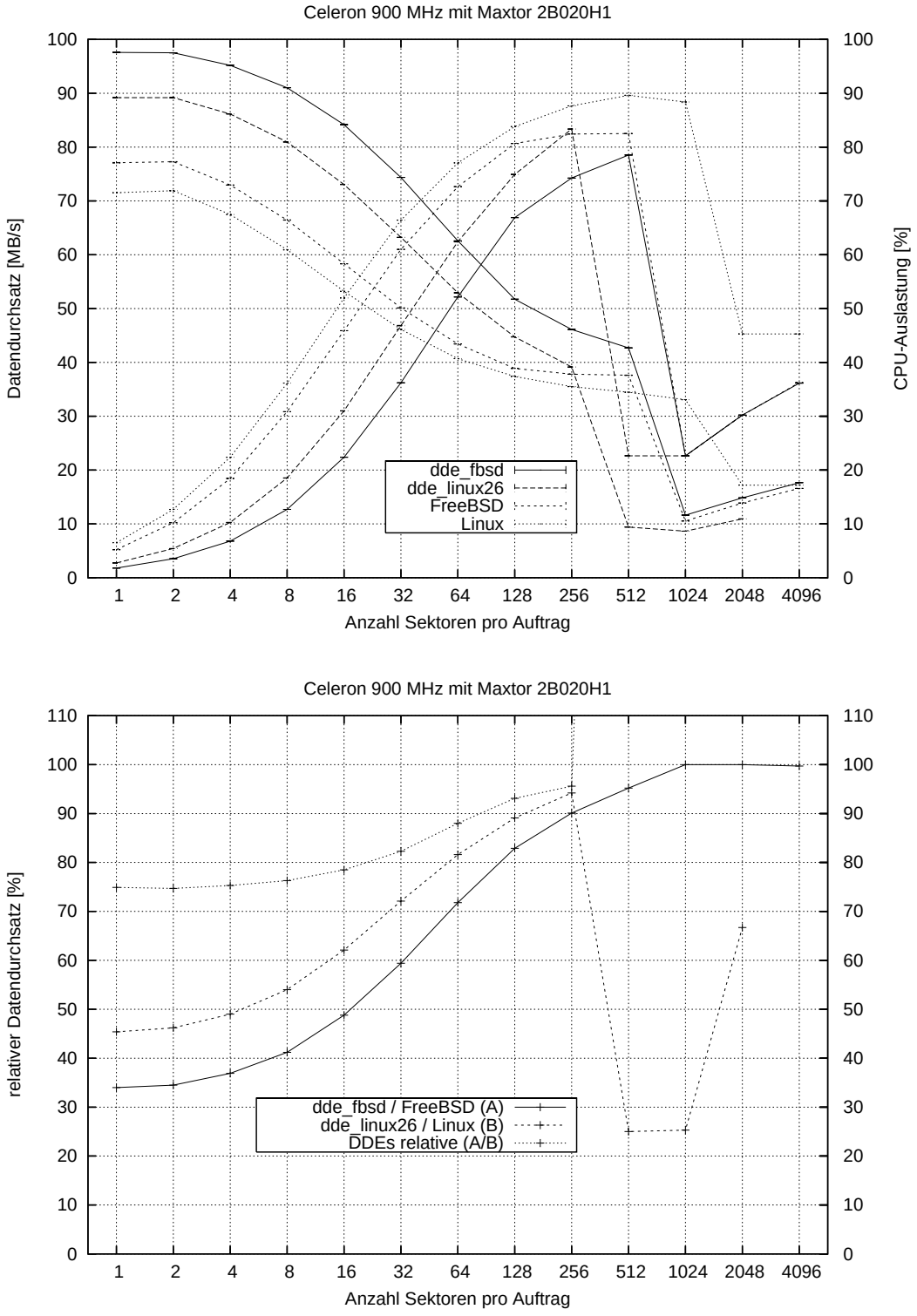

Abbildung 6.4: Leistungsvergleich auf dem Celeron 900-System. Besonders auffällig ist das Einbrechen des Datendurchsatzes bei unterschiedlichen Blockgrößen bei dde_linux26 und dem nativen Linux.

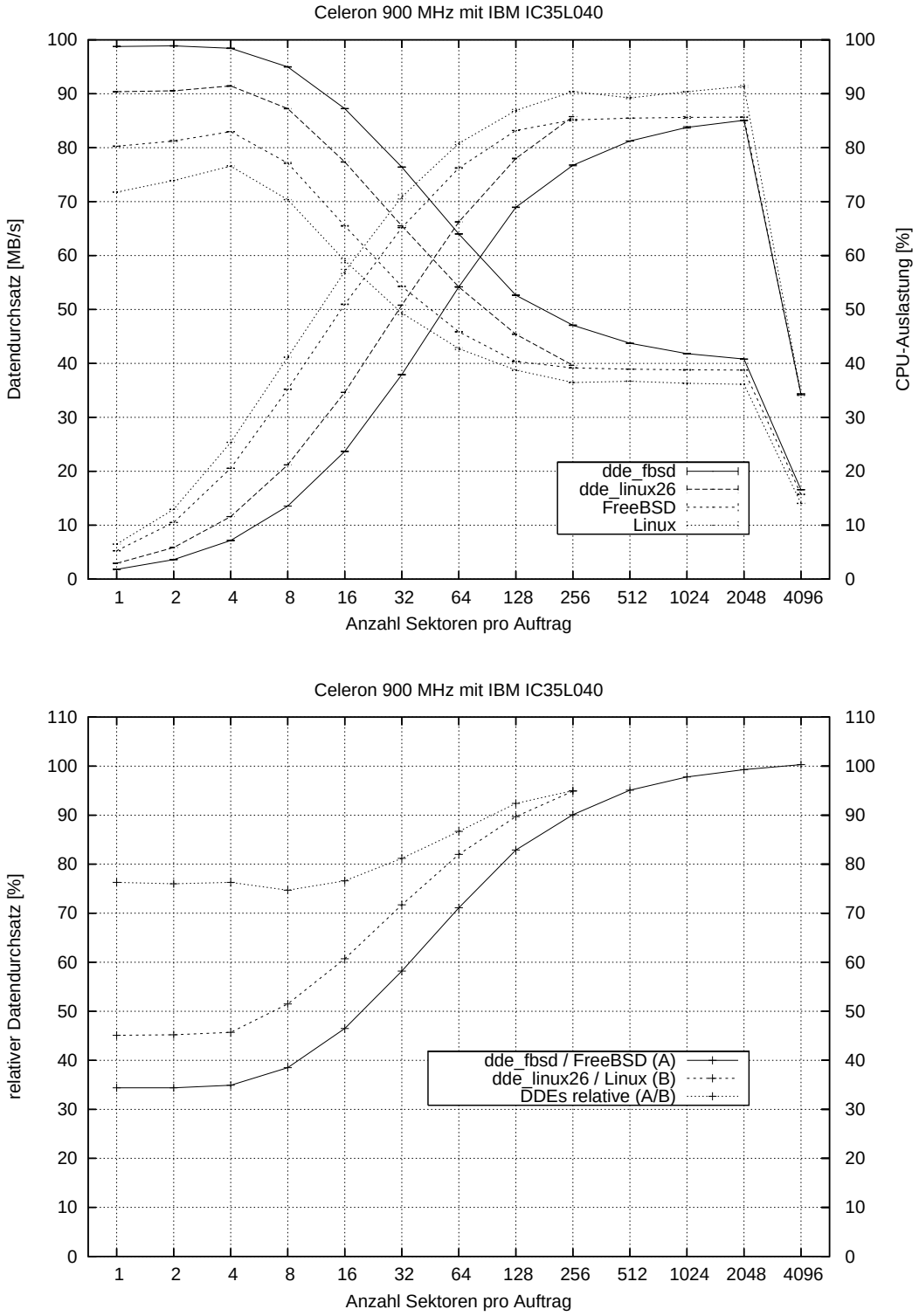

Abbildung 6.5: Leistungsvergleich auf dem Celeron 900-System, mit der IBM IC35L040 Festplatte des Celeron 1700-Systems wiederholt.

den entsprechenden Verlauf für dde_linux26 relativ zu Linux. Der dritte Graph (A/B) zeigt den Quotienten A geteilt durch B, vergleicht also die relative dde_fbsd- mit der relativen dde_linux26-Leistung.

6.2.3 Interpretation des Kurvenverlaufs für die IBM-Festplatte

Der Verlauf der Graphen für die IBM-Festplatte, also Abbildungen 6.3 und 6.5, lässt sich sehr gut erklären: Bei niedrigen Blockgrößen ist der Rechenaufwand zum Erstellen und Bearbeiten der Aufträge gegenüber der Auftragsbearbeitungsdauer der Festplatte sehr groß. Dadurch kann kein hoher Datendurchsatz erreicht werden und die CPU-Auslastung ist sehr hoch. Mit höheren Blockgrößen sinkt das Verhältnis Rechenaufwand zu I/O-Aufwand, da mit einem einzelnen Auftrag mehr Daten übertragen werden. Entsprechend steigt der Datendurchsatz an, wohingegen die CPU-Auslastung sinkt. Bei hohen Blockgrößen nähert sich der Datendurchsatz einem maximalen Wert.

Alle Graphen für die IBM-Festplatte, die sich auf dde_linux26 beziehen, brechen bei einer Blockgröße von 256 Sektoren ab, da der Linux-IDE-Treiber für diese nur DMA-Aufträge über bis zu 256 Sektoren unterstützt. Unter Linux werden größere Aufträge in mehrere kleinere geteilt, was sich am deutlichsten in Abbildung 6.5 oben in einem Knick der Kurvenverläufe bei einer Blockgröße von 256 Sektoren wiederspiegelt. Im bddf_ide, dem dde_linux26-IDE-Treiberserver, fehlt die Unterstützung für größere Aufträge.

Da bei beiden verwendeten Festplatten spätestens nach der Übertragung von 16 Sektoren ein Interrupt ausgelöst wird, nähert sich die CPU-Auslastung im weiteren Verlauf nicht der Null-Prozent-Marke sondern einem vom Aufwand der Interruptbehandlung und Auftragsvor- und -nachbereitung abhängigen Wert darüber. Entsprechend ist auch der erreichbare Datendurchsatz von System zu System unterschiedlich.

Bei einer Blockgröße von 4096 Sektoren, also 2048 kB, bricht der Datendurchsatz auf allen Systemen ein. Die IBM-Festplatte verfügt zwar über 2048 kB Cache, diese werden jedoch offensichtlich nicht vollständig benutzt, so dass bei dieser Auftragsgröße bereits von der Festplatte gelesen werden muss. Die CPU-Auslastung sinkt ab diesem Punkt ebenfalls, da sich der Rechenaufwand kaum, die Bearbeitungszeit durch die Festplatte aber stark erhöht.

6.2.4 Interpretation des Kurvenverlaufs für die Maxtor-Festplatte

Die Graphen in Abbildung 6.4, welche sich auf die Maxtor-Festplatte beziehen, lassen sich nicht so leicht nachvollziehen. Der Kurvenverlauf für dde_fbsd und FreeBSD verhält sich ähnlich dem der IBM-Festplatte, wie er im vorherigen Abschnitt beschrieben ist. Nur scheint hier nicht der gesamte Cache von 2048 kB, sondern weniger als 512 kB (1024 Sektoren) ausgenutzt zu werden, da der Datendurchsatz an dieser Stelle bereits einbricht. Unter Linux dagegen bricht der Durchsatz erst bei einer Blockgröße von 2048 Sektoren (1024kB) ein – bei dde_linux jedoch schon bei 512 Sektoren (256kB). Die benutzte Cachegröße ist hier offensichtlich unterschiedlich. Daher lassen sich nur die Werte bis zu einer Blockgröße von 256 Sektoren miteinander verglei-

chen. Für das Verständnis der Messwerte bei höheren Blockgrößen ist eine detaillierte Kenntnis der Funktionsweise des Caches nötig.

6.2.5 Bewertung

In diesem Abschnitt werden die Leistungsdaten der DDEs im Vergleich mit den entsprechenden Legacy-Systemen und untereinander bewertet, und es werden Ansatzpunkte für Optimierungen erörtert.

Leistung der DDEs

Die Bewertung der DDE-Leistung lässt sich am besten anhand der Diagramme in Abbildung 6.5 vornehmen. Der Kurvenverlauf ist trotz der sehr unterschiedlichen Prozessortaktung sehr ähnlich. Bei niedrigen Blockgrößen bis acht Sektoren wird nur circa 40% des Datendurchsatzes des jeweiligen Legacy-Systems erreicht. Mit höheren Blockgrößen steigt er gleichmäßig an, bis bei 256 Sektoren pro Auftrag 90% des Legacy-Durchsatzes erreicht werden. Bei Verwendung noch größerer Blöcke nähert sich der Durchsatz weiter dem des Legacy-Systems, bis mit dde_fbsd bei einer Blockgröße von 4096 Sektoren 100,0% (Celeron 1700) und 100,3% (Celeron 900) des FreeBSD-Wertes erreicht werden.

Da mindestens nach jeweils 16 Sektoren immer wieder ein Interrupt ausgelöst wird, wird der Rechenaufwand und damit der Datendurchsatz bei hohen Blockgrößen überwiegend durch die Interrupt-Behandlung, bei niedrigen Blockgrößen dagegen durch die Erstellung, Vor- und Nachbearbeitung der Aufträge beeinflusst. Da bei hohen Blockgrößen ungefähr der Legacy-Durchsatz erreicht wird, lässt sich schlussfolgern, dass die Interruptbehandlung sehr effizient erfolgt. Dies deutet darauf hin, dass auch die beteiligten Komponenten – dde_linux26, dde_fbsd, ddekit und l4io effizient arbeiten.

Die niedrigen relativen Durchsatz-Werte bei kleinen Aufträgen zeigen einen deutlichen Mehraufwand bei der Auftragsvor- und -nachbereitung an. Da die DDEs nun vermutlich effizient arbeiten, ist wahrscheinlich zum großen Teil das bddf dafür verantwortlich. Es beinhaltet aber mit der Unterstützung von Echtzeitaufträgen aber auch eine größere Funktionalität als Linux oder FreeBSD.

Desweiteren sollte bedacht werden, dass die Messmethode so gewählt wurde, dass die unterschiedliche Leistung sehr stark hervortritt. Bei normaler Nutzung wird durch die Verteilung der Zugriffe über größere Bereiche der Festplatte nicht nur aus dem Cache gelesen. Die notwendigen Bewegungen zur Positionierung des Schreib-/Lesekopfs über dem gewünschten Sektor erhöhen die Bearbeitungsdauer durch die Festplatte erheblich. Dadurch sinkt der Einfluss der Bearbeitungsdauer durch bddf und Treiber, und die daraus resultierenden Ergebnisse der verschiedenen Betriebssysteme sollten viel näher beieinander liegen.

Leistung von dde_fbsd/ddekit im Vergleich zu dde_linux

Der Verlauf des Graphen A/B in den beiden Diagrammen in Abbildung 6.5 vergleicht die Leistungen der beiden DDEs relativ zu den zugehörigen Legacy-Systemen mit-

einander. Hier erreicht dde_fbsd mit ddekit 75% bis 95% der relativen Leistung von dde_linux26. Bei dde_fbsd und/oder dem ddekit sollte es also noch einige Optimierungsmöglichkeiten geben.

Die bei der Auftragsbearbeitung benutzen Komponenten des ddekits sind die Semaphoren, Mutexe und Slabs. Zusätzlich werden die sleep- und die wakeup-Funktion für Threads verwendet, die auf Bedingungsvariablen beruhen und damit von der Effizienz von Semaphoren und Mutexen abhängen. Es wird ebenfalls häufig auf thread-lokalen Speicher zugegriffen und bei jedem Auftrag werden die internen Seitentabellen bearbeitet. Letzteres sollte sich aber nicht spürbar auf die Leistung auswirken, da es nur wenige schnelle Instruktionen beinhaltet. Die anderen ddekit-Funktionen wie PCI-Konfiguration, Ressourcenverwaltung, Thread-Erstellung und Allokation größerer Speicherbereiche werden nur in der Initialisierungsphase des Treibers benutzt und wirken sich somit nicht auf die Leistung der Auftragsbearbeitung aus.

Innerhalb von dde_fbsd fällt die häufige Verwendung des curthread-Zeigers auf den aktuellen Thread auf. Dabei wird jedes Mal auf den vom ddekit bereitgestellten thread-lokalen Speicher zugegriffen. Die Verwendung von curthread kann an einigen Stellen – zum Beispiel in der Mutex-Implementierung – noch vermieden werden. Die Synchronisationsprimitive wie Semaphoren und Mutexe sind vermutlich gute Ansatzpunkte für genauere Untersuchungen, da sie häufig benutzt werden. Weitere Optimierungsmöglichkeiten finden sich sicherlich bei Messung der Leistung einzelner Komponenten.

Die Funktionen in l4ata und libdad zur Abbildung der bddf- auf die dad-Schnittstelle und der dad- auf die FreeBSD-Schnittstelle sind ebenfalls noch nicht bezüglich ihres CPU-Aufwands untersucht worden. Ein erster einfacher Ansatzpunkt wäre die Verwendung von Slab-Caches anstelle von malloc und free in l4ata.

6.3 Übertragbarkeit des Ansatzes

In den nächsten drei Abschnitten soll die Übertragbarkeit des Device-Driver-Environment- und des ddekit-Ansatzes betrachtet werden. Das beinhaltet die Verwendbarkeit für andere Geräteklassen als IDE und die Übertragbarkeit auf weitere Legacy- und andere Zielsysteme.

6.3.1 Andere Geräteklassen

Das dde_fbsd ist grundsätzlich für alle FreeBSD-Gerätetreiber verwendbar. Um eine weitere Geräteklasse zu unterstützen, müssen ihre Abhängigkeiten untersucht und erfüllt werden, falls sie nicht schon durch das dde_fbsd abgedeckt sind. Auf Grundlage der bereits im dde_fbsd vorhandenen Komponenten sollten die benötigten FreeBSD-Komponenten meist ohne oder mit nur wenigen Änderungen übernommen werden können. Die Abhängigkeiten von PCI-Audiotreibern konnten beispielsweise innerhalb von weniger als zwei Stunden erfüllt werden, indem die benötigten Komponenten einfach unverändert von FreeBSD übernommen wurden.

Außerdem ist es meist nötig, einen geräteklassen-spezifischen Adapter zu implementieren, der die entsprechende FreeBSD-Funktionalität DROPS-Anwendungen zur

Verfügung stellt. Der Adapter und die zusätzlichen Komponenten bilden dann eine neue spezifische dde_fbsd-Bibliothek, entsprechend dem Beispiel der libdad.

Auf diese Weise können nicht nur weitere Gerätetreiber, sondern auch Subsysteme unterstützt werden. Der Aufwand dafür wächst mit der Anzahl und Komplexität ihrer Abhängigkeiten. So konnte das in Abschnitt 2.1.3 beschriebene GEOM einfach übernommen werden, wohingegen die Unterstützung für ein Dateisystem oder Netzwerkprotokoll sicher bedeutend aufwendiger ist.

6.3.2 Andere Legacy-Systeme

Der Device-Driver-Environment-Ansatz lässt sich auch auf andere Legacy-Systeme anwenden. Bei dde_linux wurde ja bereits ähnlich vorgegangen. Um die in Abschnitt 3.1.1 erwähnten Vorteile einer gemeinsamen Grundlage nutzen zu können, sollte dde_linux auch auf das ddekit portiert werden.

Der Aufwand zur Unterstützung eines weiteren Legacy-Systems wird sehr unterschiedlich hoch sein. Bei Systemen wie FreeBSD, bei denen der Quelltext vorliegt und frei weiterverwendet werden darf, ist der Aufwand sicher am geringsten – am höchsten dagegen bei Systemen, zu deren Quelltext man keinen Zugriff hat, wie das bei Microsoft Windows der Fall ist. Dort muss man rein anhand der Dokumentation für Treiberentwickler vorgehen. In jedem Falle sollte die Verwendung des ddekits den Implementierungsaufwand verringern.

Treiber in Binärform

Ein anderer Aspekt ist die Frage, ob die gewünschten Treiber nur kompiliert, in Binärform vorliegen, oder ob auch der Quelltext zur Verfügung steht und verwendet werden darf. Ist letzteres nicht der Fall, so muss der Treiber zur Laufzeit, also dynamisch, zum DDE und zur Anwendung hinzu gelinkt werden. Dies ist auch die Vorgehensweise von NDISulator mit Windows-Treibern.

Dabei sind die Möglichkeiten des DDEs etwas eingeschränkt, da `#define`s oder Inline-Funktionen im kompilierten Treiber nicht mehr angepasst werden können. Dies sollte aber auch nie zwingend notwendig sein, kann jedoch sicher in einigen Fällen den Implementierungsaufwand verringern.

6.3.3 Andere Zielsysteme

Die Übertragbarkeit auf andere Zielsysteme ist mit der Portierbarkeit des ddekits gegeben. Die Dienste, die das ddekit bereitstellt, müssen auf dem Zielsystem zur Verfügung stehen, beziehungsweise implementierbar sein. Das beinhaltet, wie in Abschnitt 3.1.3 beschrieben, unter anderem Threads, Mutexe, Semaphoren, Speicherallokation/-freigabe, Interruptbehandlung und Initcalls.

PC-Betriebssysteme

Mit Ausnahme der Initcalls ist die vom ddekit benötigte Funktionalität sicher auf jedem PC-Betriebssystem vorhanden. Auf DROPS werden Initcalls durch Verwendung

eines zusätzlichen Abschnitts der ELF-Binärdatei unterstützt. Sollte dies auf dem Zielsystem nicht möglich sein, bleibt die Möglichkeit, einen Preprocessing-Schritt einzuführen, der die Initcalls in einem Array ablegt und damit zugreifbar macht.

Die Abbildung der ddekit-Dienste sollte also auf den verbreiteten PC-Betriebssystemen keine Probleme aufwerfen. Im Normalfall sollte der Portierungsaufwand für das ddekit eher gering sein, da es, wie in Abschnitt 6.1 beschrieben, sehr klein ist. Die größten Schwierigkeiten sind sicher eher praktischer Natur, wie die Verwendung eines anderen C-Compilers oder die Einbindung in das Build-System.

Hypervisor

Ein anderes Anwendungsgebiet des ddekits ist die Verwendung im Zusammenhang mit einem Hypervisor wie zum Beispiel Xen (XEN). Dabei handelt es sich um einen Virtual-Machine-Monitor, auf dem mehrere paravirtualisierte Betriebssysteme gleichzeitig ausgeführt werden können. Die Systeme laufen dabei in eigenen virtuellen Maschinen, so genannten Domains. Ab Version 2, die in (FHN+04) beschrieben wird, befinden sich die Gerätetreiber nicht mehr in Xen selbst, sondern werden auch in mit den nötigen Rechten ausgestatteten virtuellen Maschinen – Treiber-Domains – ausgeführt. Da in einer solchen Treiber-Domain die vom ddekit benötigte Funktionalität wie Threads und Speicherverwaltung nicht zur Verfügung steht, sondern auf die virtuelle Maschine abgebildet werden muss, ist der Portierungsaufwand hier deutlich höher einzuschätzen.

Kombinationsmöglichkeiten

Durch die Abstraktion des ddekits, vervielfachen sich die Anwendungsmöglichkeiten: mit jeder Portierung des ddekits auf ein neues Zielsystem, stehen auch dort sofort alle bis dahin unterstützten Legacy-Treiber zur Verfügung; und jede ddekit-basierte Implementierung eines DDEs für ein weiteres Legacy-System macht dieses sofort auf allen unterstützten Zielsystemen verfügbar.

Kapitel 7

Ausblick

Der Großteil der Entwicklung von dde_fbsd ist abgeschlossen. Jetzt sollte es sich im praktischen Einsatz bewähren, wobei es aber sicher immer wieder erweitert werden muss. Desweiteren wird dabei sicherlich die nachträgliche Implementierung einiger der in Abschnitt 4.3.8 aufgezählten noch unvollständig implementierten Komponenten notwendig sein.

Während der Entwicklung von dde_fbsd für FreeBSD 5.4 wurde FreeBSD 6 fertig gestellt. Die entsprechende Ankündigung verspricht unter anderem eine höhere Leistung beim Zugriff auf Blockgeräte. Es wäre interessant zu sehen, wie sich die Verbesserungen auf die Leistung eines dde_fbsd für die Version 6 auswirken. Dabei könnte der Aufwand zur Aktualisierung des DDE genauer untersucht werden.

Einen weiteren interessanten Anwendungsfall stellt NDISulator dar. Mit seiner Portierung mit Hilfe von dde_fbsd würden sogar Windows-Treiber für Funknetzwerkkarten unter DROPS zur Verfügung stehen. Eine andere Aufgabe wäre die Verbesserung der Leistung von dde_fbsd. Bei genauerer Untersuchung durch Messung der Leistung einzelner Komponenten finden sich sicher weitere Optimierungsmöglichkeiten.

Das Umfeld des ddekits bietet viele Möglichkeiten für Untersuchungen und Weiterentwicklungen. So wäre ein Leistungsvergleich zwischen dem jetzigen dde_linux und einer auf dem ddekit basierenden Version sehr interessant. Von sehr großem Nutzen dürfte ein Windows-DDE sein. Damit hätte man die Möglichkeit auf den sehr großen Bestand an Windowstreibern zuzugreifen. Die Entwicklung eines Gerätetreiber-Frameworks als Rahmen zur Erstellung nativer Treiber ist ebenfalls ein interessantes Einsatzgebiet für das ddekit.

DROPS fehlt momentan noch eine Treiber- und Geräteverwaltung. Mit dieser sollte es möglich sein, dass Treiber unterschiedlicher Quellen zusammenarbeiten – zum Beispiel ein USB-Gerätetreiber auf Basis von dde_fbsd und ein USB-Bustreiber auf Basis von dde_linux. Dazu müssen die notwendigen Schnittstellen gefunden und definiert werden und die Treiber-/Geräteverwaltung muss das Stapeln von Treibern unterstützen. Darüber hinaus sollte auch Hotplugging und Energiemanagement unterstützt werden. Das ddekit muss dann entsprechend erweitert werden und diese Funktionalität seinen Nutzern komfortabel zur Verfügung stellen.

Desweiteren sollten die in Abschnitt 3.4 genannten Möglichkeiten zur Fehlerisolation ausgenutzt werden. Dies beinhaltet neben einer restriktiven Rechtevergabe an die Treiberserver die Beschränkung der Zugriffsmöglichkeiten mittels DMA.

Kapitel 8

Zusammenfassung

Die Zielstellung der Entwicklung eines DDEs für FreeBSD ist erreicht. Das entstandene dde_fbsd ist nicht auf die Unterstützung von Gerätetreibern beschränkt, sondern lässt sich auch auf andere FreeBSD-Subsysteme anwenden. Beides wurde anhand des entwickelten bddf-Bustreibers l4ata gezeigt, der den FreeBSD-ATA-Treiber und das GEOM-Subsystem enthält. Ein interessanter Bestandteil der Arbeit ist das unabhängig von der Aufgabenstellung entstandene ddekit. Es bietet die nötige Grundfunktionalität zur Implementierung eines DDEs, eines nativen Treibers oder eines Treiberframeworks und sollte helfen den Entwicklungsaufwand dafür zu verringern.

Der Umfang des neu entwickelten Quelltexts für dde_fbsd beträgt nur 55% des Umfangs des vergleichbaren dde_linux und ist damit deutlich geringer. Dies wurde vor allem durch eine häufige Wiederverwendung von FreeBSD-Quelltext erreicht. Desweiteren lässt sich das dde_fbsd sehr leicht erweitern. So konnten bei seiner Entwicklung viele FreeBSD-Komponenten gänzlich ohne Anpassungen übernommen werden. Der Entwicklungsaufwand für dde_fbsd (ohne das ddekit einzuschließen) ist also als geringer als der von dde_linux einzuschätzen.

Die Auswertung der Leistung der bddf-Bustreiber – l4ata und das auf dde_linux26 aufbauende bddf_ide – zeigt, dass bei Festplattenzugriffen je nach verwendeter Blockgröße zwischen 35% und 100% des unter FreeBSD respektive Linux gemessenen Datendurchsatzes erreicht wird. Aus den Ergebnissen lässt sich ebenfalls ableiten, dass der Verlust gegenüber dem jeweiligen nativen Legacy-System vermutlich durch den bddf-Server verursacht wird. Vergleicht man diese relativen Leistungszahlen der beiden DDEs miteinander, sieht man, dass der auf dde_fbsd basierende Treiber zwischen 75% und 97% der Leistung des auf dde_linux26 basierenden Treibers erreicht. Die Unterschiede zwischen den verschiedenen DDEs untereinander und zu den Legacy-Systemen sollten jedoch im normalen Betrieb eher gering ausfallen, da die Gestaltung des Tests sie besonders herausstellt.

Literaturverzeichnis

(Cla05) Dietrich Clauß, „Investigation of Mechanisms to Support User-Level Thread Packages on Top of the L4-Fiasco Microkernel", 02/2005

(DROPS) `http://tudos.com/drops/`

(FBSD) `http://www.freebsd.org/`

(FHN+04) Keir Fraser, Steven Hand, Rolf Neugebauer, Ian Pratt, Andrew Warfield, Mark Williamson, „Safe Hardware Access with the Xen Virtual Machine Monitor", 10/2004

(Fiasco) `http://tudos.org/fiasco/`

(Hel01) Christian Helmuth, „Generische Portierung von Linux-Gerätetreibern auf die DROPS-Architektur", 07/2001

(HLM+03) Hermann Härtig, Jork Löser, Frank Mehnert, Lars Reuther, Martin Pohlack, Alexander Warg, „An I/O Architecture for Microkernel-Based Operating Systems", 07/2003

(IA32) IA-32 Intel Architecture Software Developer's Manual Volume 1: Basic Architecture

(L4Env) `http://tudos.org/l4env/`

(L4Spec) `http://tudos.org/L4/lnx86-21.ps.gz`

(LeHe03) Ben Leslie, Gernot Heiser, „Towards Untrusted Device Drivers", 03/2003

(LU04) Joshua LeVasseur, Volkmar Uhlig, „A Sledgehammer Approach to Reuse of Legacy Device Drivers", 09/2004

(LUSG04) Joshua LeVasseur, Volkmar Uhlig, Jan Stoess, Stefan Götz, „Unmodified Device Driver Reuse and Improved System Dependability via Virtual Machines", 12/2004

(Men04) Marek Menzer, „Entwicklung eines Blockgeräte-Frameworks für DROPS", 08/2004

(NDISu) `http://cvsweb.freebsd.org/src/sys/compat/ndis/`

(NdisW) `http://ndiswrapper.sf.net/`

(XEN) `http://xen.sf.net/`