technische universität
dortmund

Diplomarbeit

Aspektorientierte Entwicklung einer konfigurierbaren x86-Variante der Betriebssystemfamilie CiAO

Björn Bosselmann
9. August 2010

Betreuer:
Prof. Dr.-Ing. Olaf Spinczyk
Dipl.-Inf. Jochen Streicher

Technische Universität Dortmund
Fakultät für Informatik
Lehrstuhl Informatik 12
Arbeitsgruppe Eingebettete Systemsoftware
http://ess.cs.tu-dortmund.de

Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig verfasst, keine anderen als die angegebenen Quellen und Hilfsmittel verwendet sowie Zitate kenntlich gemacht habe.

Dortmund, den 9. August 2010

Björn Bosselmann

Zusammenfassung

Die Diplomarbeit beschäftigt sich mit der Entwicklung einer neuen konfigurierbaren Variante des *CiAO*-Systems für die *x86*-Architektur. Bislang werden auf eingebetteten Systemen häufig Betriebssysteme eingesetzt, die speziell für das jeweilige Einsatzgebiet neu entwickelt wurden. CiAO ist eine Betriebssystemfamilie für eingebettete Systeme, die mit dem Ziel entwickelt wurde, hochgradig konfigurierbar zu sein. Dadurch kann ein Systementwickler ein maßgeschneidertes System erzeugen, ohne gleich ein neues Betriebssystem entwickeln zu müssen. Vor einem ähnlichen Problem stehen ebenfalls Entwickler von *Virtual Appliances*, die ihre Software bereits vorkonfiguriert als *Virtuelle Maschine* ausliefern. Auch dort benötigt man ein maßgeschneidertes Betriebssystem.

Die typischen Betriebssysteme der *x86*-Architektur wurden im Gegensatz dazu als „Allzwecksystem“ entworfen. Sie stellen eine Vielzahl von Funktionen bereit, die unter Umständen nur selten oder nie verwendet werden, aber dennoch Laufzeit und Speicherplatz beanspruchen. Da gerade im Bereich der *Virtual Appliances* die *x86*-Architektur verwendet wird, bietet es sich an, die *CiAO*-Betriebssystemfamilie für diese Architektur anzupassen.

Im Rahmen dieser Arbeit wurde eine konfigurierbare Hardwareabstraktion für *CiAO* entwickelt, so dass das System sowohl *nativ* als auch als *Linux-Gastsystem* im 32-Bit- und 64-Bit-Modus der PC-Architektur ausgeführt werden kann. Die *Aspektorientierte Programmierung* wurde genutzt, um die neuen teils querschneidenden Belange der Architektur umzusetzen. Dabei wurde auch evaluiert, in welchem Umfang die *Aspektorientierte Programmierung* den Entwurf der Hardwareabstraktionsschicht vereinfachen kann. Es hat sich dabei herausgestellt, dass sich insbesondere die völlig neuen Funktionen der *x86*-Architektur leicht in das bestehende System integrieren lassen, obwohl der ursprüngliche Entwurf sie nicht vorgesehen hatte.

Inhaltsverzeichnis

1 Einleitung

1.1 Motivation

Eingebettete Betriebssystem fristen eher ein Schattendasein. Im größeren Kontext eingebettet, sind sie für normale Anwender eher unsichtbar, obwohl sie dennoch sehr wichtige Aufgaben übernehmen, sei es in einer der zahlreichen *CAN*-Controller eines Kraftfahrzeugs oder im heimischen Digital-Satellitenreceiver. In einer eingeschränkten Umgebung wird die eingebettete Systemsoftware häufig auf das Endprodukt zugeschnitten, so dass das System gerade eben noch die Anforderungen erfüllen kann. Bei den typischen PC-Betriebssystemen sieht es hingegen völlig anders aus. Dort interessiert es den Benutzer in der Regel schon, was das System für ihn leisten kann. So werden die konventionellen Betriebssysteme im Gegensatz zu den eingebetteten Vertretern als „Allzwecksystem“ konzipiert, dass heißt, sie müssen in der Lage sein, die meisten gängigen Aufgaben irgendwie zufriedenstellend zu bewältigen. Dazu müssen sie Kompromisse eingehen, die sich negativ auf Laufzeitanforderungen und Speicherplatzbedarf auswirken, wobei das zur Gunsten der Flexibilität für herkömmliche PC-Systeme gerne in Kauf genommen wird.

Auf der anderen Seiten erwecken die *virtuellen Maschinen* (kurz: *VMs*) in der letzten Zeit immer mehr Aufmerksamkeit. So bieten gängige PC-Prozessoren bereits eine Hardwareunterstützung für die *Virtualisierung* an, um das Ausführen von virtuellen Maschinen zu beschleunigen. Daran knüpft auch die Idee der *Virtual Appliances* an. Hierbei geht es darum, Softwareprodukte nicht mehr als Installationsmedium herauszugeben, die der Benutzer selbst installieren muss, sondern als bereits vorinstalliertes System in der Form eines Festspeicherabbildes einer virtuellen Maschine. In gewisser Form ähnelt so eine *VM* wiederum einem eingebetteten System, da sich auch hier der Benutzer primär für die vorinstallierte Software der *VM* interessiert. Also bietet es sich auch hier an, ein konfigurierbares eingebettetes Betriebssystem zu nutzen, dass keine unnötigen Funktionen enthält. Durch die gute Verfügbarkeit der *x86*-Hardware und dazu passende Virtualisierungslösungen empfiehlt es sich dementsprechend, auf diese Architektur aufzusetzen.

1.2 Ziele der Arbeit

Diese Arbeit befasst sich mit der Entwicklung einer konfigurierbaren *x86*-Variante der *CiAO*-Betriebssystemfamilie. Im ersten Schritt soll analysiert werden, welche Belange dabei von Bedeutung sind. Begleitend dazu sollen die einzelnen Merkmale des Systems

festgehalten werden. Anschließend soll ein aspektorientierter Entwurf unter Berücksichtigung der Entwurfsprinzipien *Lose Kopplung*, *Sichtbare Übergänge* und *minimale Erweiterung* entwickelt werden. Dabei soll insbesondere herausgefunden werden, wie sich die teils neuen Belange mit aspektorientierten Sprachmitteln umsetzen lassen, auch wenn *CiAO* bislang keine Unterstützung für einige der neuen *x86*-Funktionen vorsieht. Wenn möglich soll eine beispielhafte Implementierung für diese Funktionen entwickelt und evaluiert werden.

1.3 Gliederung der Arbeit

Im nächsten Kapitel werden zunächst die allgemeinen Grundlagen zur verwendeten Hard- und Software erläutert. Beginnend mit einer kurzen Einführung in die *CiAO*-Betriebssystemfamilie werden die Ziele und Entwicklungsprinzipien sowie die dazu notwendigen Werkzeuge erklärt. Im darauf folgenden Teil wird die weit verbreitete *x86*-Architektur behandelt. Insbesondere werden die Besonderheiten und Unterscheidungsmerkmale zu anderen Architektur betrachtet. Im letzten Teil des Grundlagenkapitels geht es um die Herausforderung der Virtualisierung und mit welchen Mitteln sie realisiert wird.

Im dritten Kapitel wird eine Analyse der zu entwickelnden Hardwareabstaktionsschicht durchgeführt. Dazu werden die notwendigen Belange herausgestellt. Ein wichtiges Augenmerk liegt dabei auf die gegenseitigen Auswirkungen, die die neuen Komponenten mitbringen werden.

Im vierten Kapitel wird die Konzeption und die Realisierung zu den analysierten Komponenten vorgestellt. Falls für diese Komponenten eine neue plattformunabhängige Schnittstelle entwickelt wurde, wird diese kurz erklärt. Wenn möglich wird ergänzend dazu jeweils eine beispielhafte Implementierung der Komponenten präsentiert.

Im fünften Kapitel wird wird die neu entwickelte *CiAO*-Variante evaluiert. Zunächst werden Speicherverbrauch und Laufzeit gemessen. Anschließend wird untersucht, welche Belange des Systems sich aspektorientiert umsetzten ließen.

Der abschließende Ausblick gibt ein kurzes Fazit und geht auf Ansatzpunkte für weitere Entwicklungen ein.

2 Grundlagen

Dieses Kapitel erläutert wichtige Grundlagen, die für das Verständnis der folgenden Kapitel unumgänglich sind. Zunächst wird die *CiAO*-Betriebssystemfamilie näher erläutert. Dazu werden die Konzepte und Entwicklungsprinzipien sowie die dazu benötigten Entwicklungswerkzeuge vorgestellt. Anschließend wird die *x86*-Architektur erläutert, angefangen bei den grundlegenden Systemkomponenten bis hin zu einigen Peripheriegeräten, die im Laufe der Arbeit noch eine Rolle spielen werden. Zum Schluss dieses Kapitels wird noch auf die *Virtualisierung* und dessen Herausforderungen eingegangen.

2.1 *CiAO is Aspect-Oriented*

2.1.1 Ziele

Die *CiAO*-Betriebssystemfamilie wurde vor dem Hintergrund entwickelt, die Idee der *Aspektorientierten Programmierung* vom Design an bis hin zur Implementierung eines *AUTOSAR-OS*-kompatiblen Betriebssystems zu berücksichtigen [1, Kapitel 5]. Dies eröffnete neue Möglichkeiten ein System hochgradig bis ins kleinste Detail konfigurierbar zu gestalten.

2.1.2 Aufbau

Wie in Abbildung 2.1 zu sehen ist, wurde CiAO nach einem einfachen Schichtenmodell entworfen. Von unten nach oben betrachtet handelt es sich dabei im Wesentlichen um folgende Schichten:

hw::dev Die **Treiberschicht** implementiert die hardwareabhängigen Zugriffsroutinen für sämtliche Peripheriegeräte. Diese Schicht ist größtenteils plattformabhängig und muss für jede Zielplattform fast vollständig neu implementiert werden.

hw::hal Die **Hardwareabstraktionsschicht** baut direkt auf die Treiberschicht auf, und stellt nach oben hin eine portable Schnittstelle auf allgemeine Systemkomponenten wie Koroutinen, Timer, etc. zur Verfügung. Auch diese Schicht muss für jede Zielplattform angepasst werden, wobei es sich größtenteils nur um Anpassungen der Adapter-Klassen und um Änderungen in den hardwareabhängigen Datenstrukturen wie z. B. denen des Taskkontextes handelt.

os::krn Die **Kernelschicht** stellt die betriebssystemtyptischen Systemdienste wie den Scheduler, etc. zur Verfügung. Aufgrund der Tatsache, dass diese Schicht im Ideal-

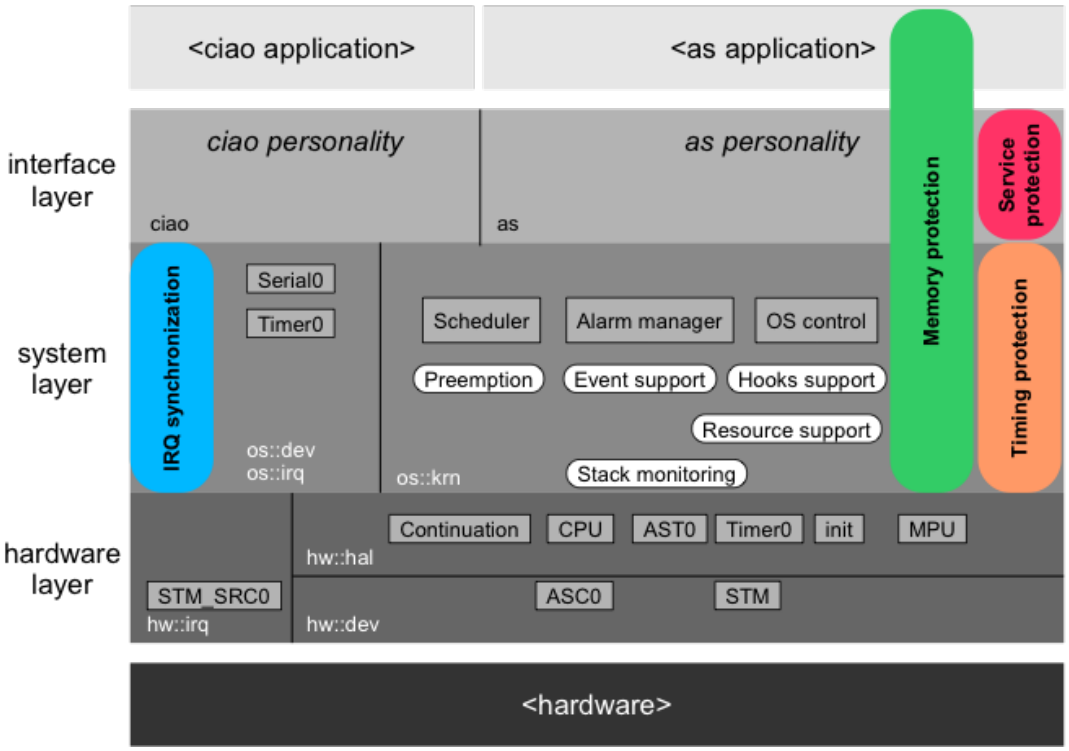

Abbildung 2.1: CiAO Schichtenmodell (Quelle: [2, 03.4, S. 22])

fall ausschließlich Komponenten der Hardwareabstraktionsschicht nutzt, kann sie fast ohne Änderung für jede Zielplattform übernommen werden.

as Bei der ***AUTOSAR-OS*-Schnittstelle** handelt es sich analog zum Verhältnis von `hw::hal` zu `hw::dev` um eine Adapter-Klasse, die AUTOSAR-Aufrufe in native CiAO-Aufrufe umsetzt. Da die Kernelschicht für neue Plattformen nahezu unverändert übernommen werden kann, muss auch die AUTOSAR-Schicht nicht angepasst werden.

2.1.3 verwendete Techniken

Die herkömmlichen Entwicklungswerkzeuge sind bei der Entwicklung eines hochgradig konfigurierbaren Systems überfordert. Zum Erreichen der Konfigurierbarkeit werden zusätzliche Werkzeuge benötigt, mit der die Konfigurationspunkte verwaltet und mit den Implementierungsartefakten verbunden werden können. Im Folgenden wird dazu *pure::variants* vorgestellt, einem Programm zum Erstellen und Verwalten von *Software-Produktlinien*.

2.1.3.1 Software-Produktlinien mit *pure::variants*

Eine *Produktlinie* innerhalb von *pure::variants* (kurz: *p::v*) unterteilt ein Projekt zunächst in einen *Problemraum* und in einen *Lösungsraum* [3].

- Der **Problemraum** spiegelt die einzelnen Konfigurationspunkte und deren Abhängigkeiten wider. Innerhalb von *pure::variants* wird dieser Raum durch ein Merkmalmodell dargestellt, wie in Abbildung 2.2 beispielhaft zu sehen ist.

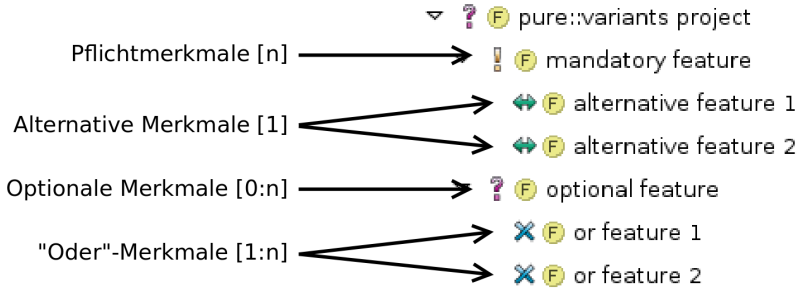

Abbildung 2.2: Featuremodell in pure::variants

- Der **Lösungsraum** beinhaltet die Implementierung aller Komponenten und ist ebenfalls in einer Baumstruktur organisiert, wie in Abbildung 2.3 zu erkennen ist. Zur Implementierung gehören hauptsächlich die Quellcodedateien. In einigen Fällen ist *pure::variants* aber auch in der Lage, selbst Dateien zu generieren. Im einfachsten Fall kann dies eine *Flag*-Datei sein, die Attribute des Modells als Konstanten innerhalb der Implementierung zur Verfügung stellt. Aber es können auch nahezu beliebig komplexe Operationen durchgeführt werden, wie beispielsweise das Ausführen eines *XSLT*-Prozessors oder eines beliebigen externen Programms. Ansonsten werden im Lösungsraum die einzelnen Komponenten mit den Konfigurationsmerkmalen des *Problemraums* verknüpft, so dass die jeweilige Komponente nur dann berücksichtigt wird, wenn das entsprechende Merkmal ausgewählt wurde.

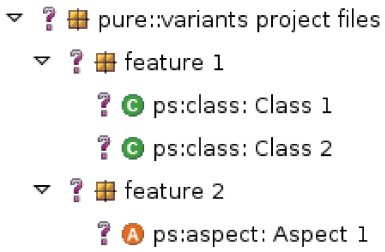

Abbildung 2.3: Familienmodell in pure::variants

2.1.3.2 Aspektorientierte Programmierung

Im Idealfall implementiert ein Artefakt genau eine Anforderung oder ein Belang. Dies ist beim Einsatz konventioneller Programmierparadigmen nicht immer möglich. So gibt

es sogenannte *querschneidende Belange*, die komponentenübergreifend auf viele unterschiedliche Stellen Einfluss nehmen können. Rein imperative Programmiersprachen verlangen, dass jede Operation explizit an jeder betroffenen Stelle im Quellcode ersichtlich sein muss. Da ein querschneidender Belang nun viele Stellen im Kontrollfluss beeinflussen kann, muss jede Stelle entsprechend bearbeitet werden, wie in Abbildung 2.4 gezeigt wird. Insbesondere, wenn die Möglichkeit besteht, Belange wahlweise ein- und auszuschalten, wie es bei Software-Produktlinien der Fall ist, muss man Vorkehrungen treffen, jede dieser Stellen bei der Konfigurierung zu berücksichtigen, was den Quellcode nicht selten unlesbar macht. Um ein einfaches *Tracing* zur Verfolgung von Funktionsaufrufen, in reinem *C* realisieren zu können, muss jede Funktion mit entsprechenden *Tracing-#ifdefs* versehen werden:

Listing 2.1: Konfigurierbares *Tracing* in *C*

```
1  int function() {
2  #ifdef TRACING
3    printf("entering %s\n", __func__);
4  #endif
5    int value = get_value();
6  #ifdef TRACING
7    printf("returning %d from %s\n", value, __func__);
8  #endif
9    return value;
10 }
```

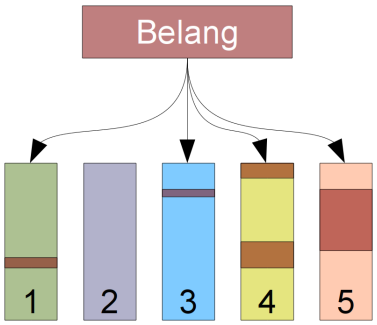

Abbildung 2.4: Querschneidender Belang ohne Aspekte

Die *Aspektorientierte Programmierung* setzt nun genau an diesem Punkt an und erweitert eine normale imperative Programmiersprache um Sprachelemente, die es ermöglichen, den Kontrollfluss gleich mehrerer Komponenten („*quantification*“) ohne dessen vorherige Präparierung („*obliviousness*“) zu beeinflussen [2, 03.3] [4]. Der querschneidende Belang wird wie in Abbildung 2.5 gezeigt in einen *Aspekt-Header* ausgelagert. Erst der sogenannte *Aspektweber* fügt den Aspekt an den entsprechenden Stellen ein, was für den Entwickler allerdings völlig transparent ist.

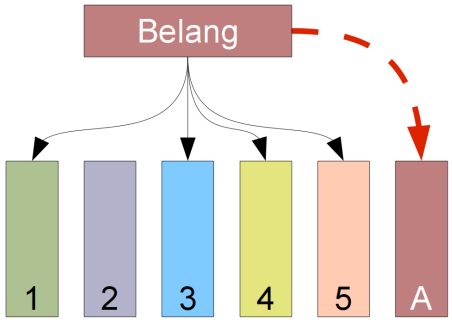

Abbildung 2.5: Querschneidender Belang mit Aspekte

AspectC++

AspectC++ ist eine Spracherweiterung der Programmiersprache C++, die zusätzlich zu allen weiteren Programmierparadigmen von C++ die *aspektorientierte Programmierung* ermöglicht. Dies wird mit Hilfe neu hinzugefügter Schlüsselwörter erreicht. Um das *Tracing*-Beispiel mit Hilfe der *Aspektorientierten Programmierung* zu lösen, reicht nun folgender *Aspekt* aus:

Listing 2.2: Konfigurierbares *Tracing* in *AspectC++*

```
1 aspect Tracing {
2   advice execution("% ...::%(...)") : before() {
3     printf("entering %s\n", tjp->signature());
4   }
5   advice execution("% ...::%(...)") : after() {
6     printf("returning %d from %s\n", *tjp->result(), tjp->signature());
7   }
8 };
```

Im Listing 2.2 wird der *Aspekt* `Tracing` angelegt, der zwei *Advices* enthält. Beide beeinflussen alle Funktionenausführungen (`execution`), die dem *Pointcut* `"% ...::%(...)"` genügen. Dies betrifft also alle Funktionen oder Methoden in allen Namespaces, unabhängig von den Funktionsparametern und dem Typ des Rückgabewertes. Der erste *Advice* wird vor jeder Funktionsausführung (`before()`) ausgeführt, während der zweite *Advice* nach jeder Funktion (`after()`) ausgeführt wird. Statt der Präprozessor-Konstanten `__func__` wird nun die *JoinPoint-API* genutzt, um den Funktionsnamen und den Rückgabewert zu erhalten [5].

Der Aspektweber wird dem eigentlichen Compiler vorgeschaltet und erzeugt wie ein Präprozessor eine Quellcodedatei ohne die aspektorientierten Erweiterungen, bei der sämtliche Aspekte bereits gewebt wurden.

2.1.4 Aspektorientierte Entwurfsprinzipien

Bei der Entwicklung der CiAO-Betriebssystemfamilie wurden drei grundlegende Entwurfsprinzipien berücksichtigt:

Lose Kopplung

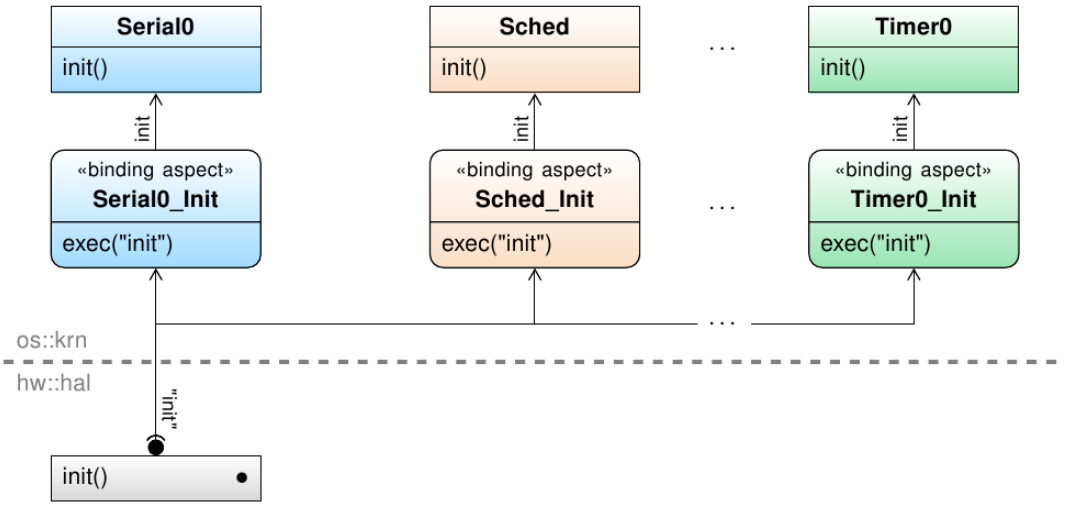

Abbildung 2.6: Lose Kopplung (Quelle: [2, 03.4, S. 27])

Mit der *losen Kopplung* erlaubt man Komponenten, sich quasi selbst in das bestehende System einzubinden, ohne dass man dazu bestehenden Komponenten verändern muss. In Abbildung 2.6 kann man beispielsweise erkennen, dass sich diverse Initialisierungsfunktionen via *Binding-Aspekte* an die globale `init()`-Funktion hängen können, um zusammen mit dieser aufgerufen zu werden. So wird bei jedem Aufruf der `init()`-Funktion aus `hw::hal` jeweils die `init()`-Funktionen der Klassen `Serial0`, `Sched` und `Timer0` aufgerufen.

Der *Binding-Aspekt* `Serial0_Init` könnte beispielsweise so aussehen:

Listing 2.3: Lose Kopplung

```
1 aspect Serial0_Init {
2   advice execution( "void hw::hal::init()" ) : after() {
3     os::krn::Serial0::init();
4   }
5 };
```

Sichtbare Übergänge

Die *sichtbaren Übergänge* ergänzen das Prinzip der losen Kopplung, indem sie definierte Verbindungspunkte für *Policy-Aspekte* in der Form von leeren Funktionen zur Verfügung stellen. Was auf den ersten Blick nach einem Bruch der „obliviousness-Regel“, die in Kapitel 2.1.3.2 beschrieben wurde, aussieht, ist hier notwendig, da die impliziten Verbindungspunkte nicht immer ausreichen. Oft sind davon „Konzepte“ betroffen, die in dieser Form nicht direkt im Quellcode vorkommen, wie beispielsweise das Vorhandensein eines Kernel-Modus, dessen genaue Abgrenzungen nur durch die expliziten Verbindungspunkte `enterKernel()` und `leaveKernel()` zu erkennen sind.

Minimale Erweiterungen

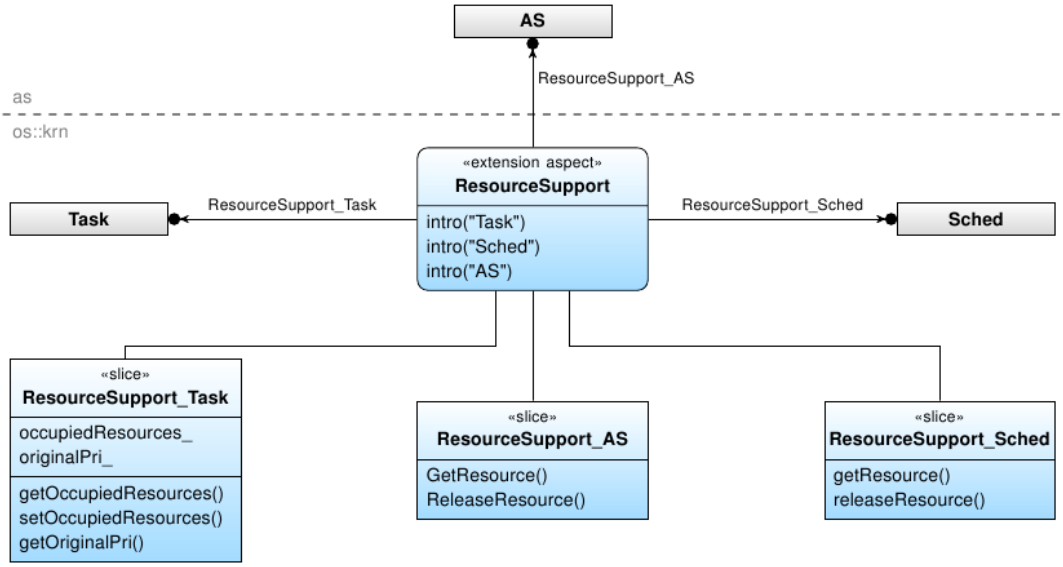

Abbildung 2.7: Minimale Erweiterungen (Quelle: [2, 03.4, S. 31])

Mithilfe der *minimalen Erweiterungen* werden bereits vorhandene Klassen um Attribute und Methoden erweitert. Dies entspricht der konsequenten Verwendung der *Klassen-Slices*. In Abbildung 2.7 wird beispielsweise die aspektorientierte Umsetzung der Ressourcenunterstützung in CiAO dargestellt. So werden die Klassen `Task`, `AS` und `Sched` nachträglich um Methoden erweitert, ohne dass die jeweiligen Klassenrümpfe verändert werden müssen. Beispielsweise erhält die `AS`-Klasse so die zusätzlichen Methoden `GetResource()` und `ReleaseResource()`.

2.2 *x86*-Architektur

Zu der *x86*-Architektur zählt man im weitesten Sinne alle Rechnersysteme, die einen *x86*-Prozessor als Zentraleinheit verwenden. Darunter fallen neben den klassischen *PC-BIOS*-Systemen auch *EFI*-basierte Computer wie sie seit 2005 beispielsweise bei Apple zu finden sind. Im engeren Sinne versteht man unter der klassischen *x86*-Architektur ausschließlich die Reihe der *IBM-PC-kompatiblen* Computer, die in dieser Arbeit betrachtet werden.

Ein *x86*-Prozessor zeigt sich nach außen hin als klassischer *Complex Instruction Set Computer* (kurz: *CISC*). So steht ihm im direkten Vergleich zum Modell der *Reduced Instruction Set Computer* (kurz: *RISC*) ein mächtiger Befehlssatz zur Verfügung, angefangen bei der umfangreichen Adressberechnung bis hin zu Zeichenkettenoperationen [6, Kapitel 8.3].

Seine Daten verwaltet der *x86*-Prozessor im Gegensatz zu vielen anderen Architekturen in zwei völlig unterschiedlichen Adressräumen. Man unterscheidet hier zwischen dem

Speicheradressraum, in dem, wie der Name vermuten lässt, größtenteils Hauptspeicher zu finden ist und dem *I/O-Adressraum*, der hauptsächlich Peripheriegeräte beinhaltet [7, Kapitel 1.4]. Allerdings geht der Trend immer mehr dahin, den Speicheradressraum auch für Geräte zu verwenden. Diese bezeichnet man dann als *memory-mapped*.

Die *x86*-Architektur wurde im Laufe der Zeit mehrfach erweitert, wobei aber ein großer Wert auf Abwärtskompatibilität zum „Ur-PC“, dem *IBM 5150* gelegt wurde. So unterstützen selbst die neusten Prozessoren den Betriebsmodus des damals verwendeten *x86*-Prozessors, dem *Intel 8086* und dem aus Sicht der Software vollständig kompatiblen *Intel 8088* [6, Kapitel 3.8].

Da die zahlreichen Erweiterungen, insbesondere in Hinblick auf die Datenwortbreite, zu Inkompatibilitäten geführt hätten, wurden zur Laufzeit umschaltbare Betriebsmodi eingeführt, die im Folgenden behandelt werden.

2.2.1 Betriebsmodi

Der **16-Bit Real-Mode** ist der ursprüngliche 16-Bit Betriebsmodus des *x86*. Selbst aktuelle *x86*-Prozessoren stehen in diesem Modus nur insgesamt 20 Adressbits und somit ein MB Speicherraum zur Verfügung. Wegen der reinen 16-Bit Architektur setzt sich eine lineare 20-Bit-Adresse aus einem mit 16 multipliziert Segmentregisterwert addiert mit einem 16-Bit-Offset zusammen [6, Kapitel 6.5]. Genutzt wird dieser Modus ausschließlich während des Bootvorgangs, der im Kapitel 2.2.4 genauer beschrieben wird.

Der **16-Bit Protected-Mode** ist eine Erweiterung, die ausschließlich auf dem *286er* genutzt wurde. Anders als im Real-Mode referenzieren die Segmentregister nun sogenannte *Segmentdeskriptoren*, die in *globalen* (*GDT*) oder *lokalen* (*LDT*) *Deskriptor Tabellen* untergebracht sind. Neben der obligatorischen Segmentstartadresse bieten die Segmentdeskriptoren noch die Möglichkeit, Segmentlängen und Zugriffsberechtigungen anzugeben. Damit bietet der Protected-Mode erstmals Mechanismen, um effektiv den Zugriff auf Speicherbereiche zu beschränken.

Der **32-Bit Protected-Mode** wurde mit dem *386er* eingeführt und ist bis heute der verbreitetste Modus, wird aber nach und nach durch den *Long-Mode* ersetzt. Im Vergleich zum *16-Bit Protected-Mode* gehört zu den wesentlichen Änderungen die Erweiterung der meisten Register, Operationen und Speicherzugriffe auf eine Breite von 32 Bit. Zusammen mit dem 32-Bit Protected-Mode wurde ebenfalls eine seitenbasierte Speicherverwaltung eingeführt, die in Kapitel 2.2.2 näher erläutert wird. Die Tatsache, dass die Adresslänge identisch ist mit der Registerlänge war die Geburtsstunde des *flachen Speichermodells* [6, Kapitel 6.7].

Der **64-Bit Long-Mode** ist die 64-Bit Erweiterung der *x86*-Architektur. Neben der erneuten Erweiterung der 32-Bit-Register auf 64-Bit Länge, wurden 8 weitere Allzweckregister hinzugefügt. Die seitenbasierte Speicherverwaltung wurde ebenfalls auf die größere Adressbreite angepasst, wohingegen die segmentierte Speicherverwaltung größtenteils entfernt wurde.

2.2.2 Speicherverwaltung

Wie im vorangehenden Kapitel bereits erwähnt wurde, unterstützt die *x86*-Architektur abhängig vom Betriebsmodus eine *segmentbasierte* oder eine *seitenbasierte* Speicherverwaltung.

Die **segmentbasierte Speicherverwaltung** organisiert den Speicher als Reihe von Segmenten, die sich in Startadresse und, seit der Einführung des Protected-Mode, in Länge und Zugriffsart unterscheiden können. Sämtliche Speicherzugriffe beziehen sich immer genau auf ein Segment, auf dessen Startadresse ein Offset addiert wird. Da diese Art der Speicherverwaltung im 64-Bit Modus entfernt wurde[1] [8, Kapitel 3.4.2.1], wird sie hier nicht weiter betrachtet.

Die **seitenbasierte Speicherverwaltung**, auch *Paging* genannt, geht einen Schritt weiter als die segmentbasierte Speicherverwaltung und bildet blockweise virtuelle Speicheradressen auf physikalische Speicheradressen ab. Ein Speicherzugriff auf eine bestimmte Speicheradresse wird demnach nicht mehr direkt ausgeführt, sondern es findet vorher eine „Adressumrechnung“ von der virtuellen Adresse aus dem Programm in eine physikalische Speicheradresse statt. Für die Umrechnung wird in der Regel eine Tabelle verwendet, in der die einzelnen Speicherblöcke abgebildet werden. Diese enthält zusätzlich noch Informationen über die zugelassene Zugriffsart, um unkontrollierte Speicherzugriffe einzuschränken. Auf dem *x86* verwendet man keine einzelne große Tabelle, die alle Einträge beherbergt, da dieser Ansatz zu unflexibel wäre. Stattdessen wird von der *Memory Management Unit* (kurz: *MMU*) eine hierarchische Tabelle genutzt. Die einzelnen Stufen dieser Hierarchie verarbeiten jeweils nur einen Teil der virtuellen Speicheradresse. So entscheidet die erste Stufe der Seitentabelle anhand der 9 oder 10 höchstwertigen Bits der virtuellen Adresse, welche zweite Stufe zu verwenden ist. Diese ermittelt anhand der nächsten 9 oder 10 Bits, welche dritte Stufe zum Zuge kommt, usw., bis die gesamte Adresse verarbeitet wurde. Nicht benötigte virtuelle Speicherbereiche können in der Tabelle ausgelassen werden, um die Seitentabelle klein zu halten. Je nach verwendetem Tabellenformat und Seitengröße könnte die Tabelle sonst riesige Ausmaße annehmen. Die hierarchische Struktur erlaubt es zudem, die Seitentabelle bis zu einem gewissen Grad zu fragmentieren, was die Seitenverwaltung erheblich vereinfacht.

Aufgrund der wachsenden Speichergrößen wurde das Format der hierarchischen Seitentabellen mehrfach angepasst. So unterscheidet man zwischen dem *32-Bit*-, *PAE*- und *64-Bit*-Paging.

32-Bit-Paging

Das **32-Bit**-Paging wurde zusammen mit dem 32-Bit Protected Mode eingeführt und unterstützt ursprünglich nur eine Seitengröße von 4 KB. Die gesamte Tabellenhierarchie wird in Abbildung 2.8 genauer dargestellt.

Später wurde die *Page Size Extension* (kurz: *PSE*) eingeführt, die zusätzlich zu den 4-KB-Seiten solche mit 4 MB Größe unterstützt. Dazu referenziert das *Page Directory* direkt physikalischen Speicher.

[1] Die Segmente sind noch vorhanden, erstrecken sich aber immer über den gesamten Speicher

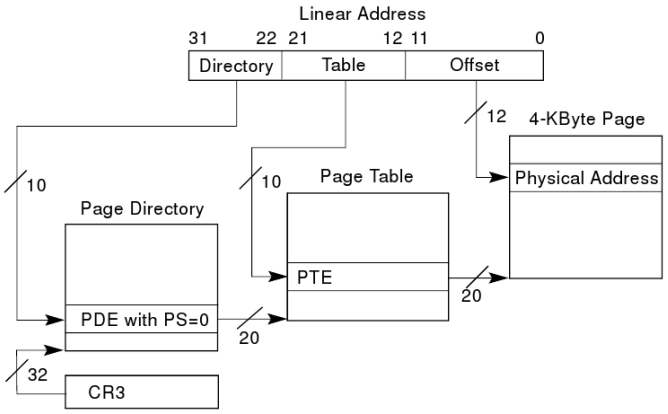

Abbildung 2.8: 32-Bit Paging (Quelle: [9, 4-10])

PAE-Paging

Mit der Einführung des **PAE**-Paging reagierte man auf die Notwendigkeit, mehr als 4 GB Speicher adressieren zu können. Dazu wurden physikalische Adressen mit bis zu 52-Bit Länge eingeführt, wohingegen die virtuelle Adresse weiterhin 32 Bit lang war. Um die Größe einer Tabellenstufe konstant zu halten, halbiert sich aufgrund der längeren Einträge die Anzahl der Verweise pro Tabelle auf 512. Zum Ausgleich wird der hierarchischen Seitentabelle eine 4 Einträge fassende Tabelle (*PDPT*) vorgeschaltet, wie in Abbildung 2.9 zu erkennen ist.

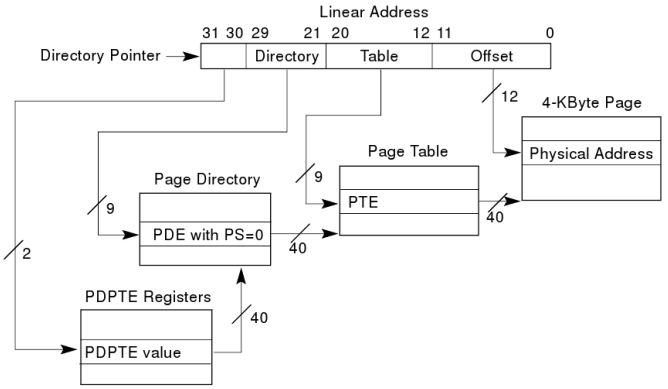

Abbildung 2.9: PAE-Bit Paging (Quelle: [9, 4-18])

64-Bit-Paging

Parallel zum Long-Mode wurde das **64-Bit**-Paging eingeführt, um nun auch virtuelle Adressen mit bis zu 48 Bit Länge umsetzen zu können. Hierzu wurden im Vergleich zum 32-Bit-Paging zwei weitere Stufen in die Tabellenhierarchie eingefügt, wie man in Abbildung 2.10 sehen kann. Spätestens hier wird deutlich, wie groß der Vorteil des Aus-

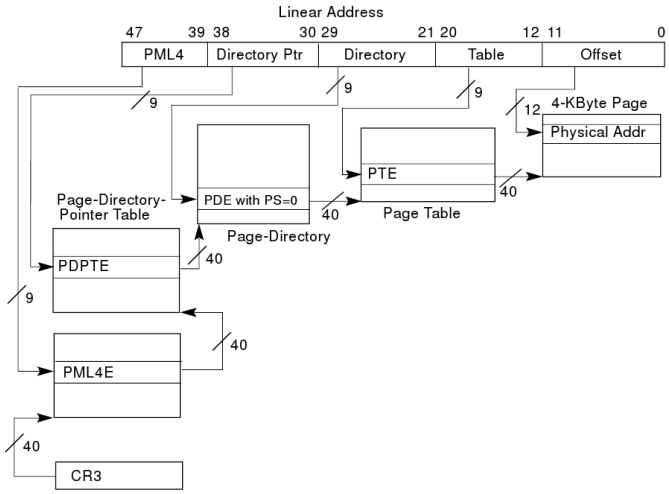

Abbildung 2.10: 64-Bit Paging (Quelle: [9, 4-25])

lassens von Tabellenstufen nicht benötigter Speicherbereiche ist. Als Beispielrechnung dient hier eine 64-Bit-Seitentabelle mit ausschließlich 4-KB-Seiten, die den vollständigen 48-Bit Adressraum abdeckt:

$$\overbrace{4KB}^{PML4} + \overbrace{512^1 \cdot 4KB}^{PDPT} + \overbrace{512^2 \cdot 4KB}^{PD} + \overbrace{512^3 \cdot 4KB}^{PT} = 537.921.540KB \approx 513GB$$

Eine Seitentabelle dieser Größenordnung würde jeden Rahmen sprengen.

2.2.3 Unterbrechungsbehandlung

Für die Vorsortierung bzw. Priorisierung eintreffenden Unterbrechungen (*Interrupts*) verwendete die klassische *x86*-Architektur ein bis zwei Bausteine des Typs *Intel 8259*, der im Kontext des PCs oft einfach nur *Programmable Interrupt Controller* (kurz: *PIC*) genannt wird, um bis zu 15 unterschiedliche externe Unterbrechungen bearbeiten zu können.

Im Laufe der Zeit wurde dieser *PIC* durch die *Advanced Programmable Interrupt Controller*-Architektur (kurz: *APIC*) ersetzt, die neben 256 verschiedenen externen Unterbrechungen zusätzlich eine Verteilung der Unterbrechungen auf mehrere Prozessoren unterstützt [9, Kapitel 10]. Falls diese Zusatzfunktionen nicht benötigt werden, besteht mit der neuen Architektur aber nach wie vor die Möglichkeit, den klassischen *PIC* zu nutzen, der vom *APIC* bei Bedarf vollständig emuliert wird. Der *APIC* wird hier nur der Vollständigkeit halber erwähnt und im Folgenden nicht weiter betrachtet.

2.2.4 Bootvorgang

BIOS

Das *Basic Input Output System* (kurz: *BIOS*) dient in erster Linie dazu, die verbaute Hardware nach dem Einschalten des Rechners zu testen und zu initialisieren. Nach diesem *Power on Self Test* (*POST*) wird der erste Datenblock vom Bootmedium in den Speicher geladen und die Ausführung im 16-Bit Real-Mode dort fortgesetzt.

Weiterhin bietet das BIOS auch nach dem Booten eine standardisierte Schnittstelle zu grundlegenden Geräten wie Tastatur, Monitor, Festspeichermedien, etc. an. Ältere Betriebssysteme (z. B. *DOS*) nutzten fast ausschließlich diese Hardwareabstraktionsschicht, um auf die Hardware zuzugreifen. Modernere Systeme bringen dazu in der Regel eigene, teils leistungsfähigere Gerätetreiber mit, so dass die BIOS-Routinen nur noch von sogenannten *Bootloadern* wie beispielsweise dem *GRUB* zum Laden des Betriebssystemkerns genutzt werden (vgl. [6, Kapitel 1.1.5]).

GRUB

Der *GRUB* (*Grand Unified Bootloader*) ist ein Repräsentant eines modernen Bootloaders mit vielen Zusatzfunktionen. So bietet er neben dem Laden eines Kernels die Möglichkeit weitere Informationen über das System (z. B. Speichergröße) sowie Kommandozeilenparameter an das zu ladende Betriebssystem zu übergeben. Außerdem kann er bei Bedarf die Umschaltung in den 32-Bit Protected-Mode übernehmen, so dass ein Betriebssystem für diese Aufgabe keinen 16-Bit-Stub benötigt.

2.2.5 Grundlegende Geräte

2.2.5.1 Timer

Ein *Timer* ist ein simpler Baustein, der in regelmäßigen Abständen eine Unterbrechung auslösen kann. Dieser wird innerhalb eines Betriebssystem in der Regel als Zeitbasis - beispielsweise für den *Scheduler* - genutzt.

Auf modernen *x86*-Systemen stehen gleich mehrere Timer-Module zur Verfügung:

- Der **Programmable Interval Timer** (kurz: *PIT*) ist der Standardtimer des IBM-PCs und steht aus Kompatibilitätsgründen auf jedem Rechner zur Verfügung. Dieser beinhaltet einen 16-Bit Zähler, der mit $1,193\overline{18}$ MHz herunterzählt und beim Erreichen der 0 den *IRQ 0* auslöst. Durch einen modifizierbaren Startwert lässt sich der Timer auf unterschiedliche Unterbrechungsintervalle einstellen.

- Die **Real-Time Clock** (kurz: *RTC*) gehört wie der *PIT* zu den „klassischen“ Systemkomponenten [6, Kapitel 19.1]. In erster Linie wird die *RTC* dazu genutzt, die aktuelle Zeit des PCs auch im ausgeschalteten Zustand weiterzuzählen. Nebenbei befindet sich im Speicher der *RTC* noch ausreichend Speicherplatz, um die *BIOS*-Einstellungen aufzunehmen. Dennoch lässt sich diese Echtzeituhr auch als

Timer verwenden. Im Normalfall zählt die Uhr im Sekundentakt, aber durch Umprogrammierung des eingebauten Taktteilers, kann sie den *IRQ 8* mit einer Rate von 2 Hz bis 8192 Hz auslösen.

- Der **APIC-Timer** ist ein Bestandteil des neueren *APIC*s [9, Kapitel 10.5.4]. Dabei handelt es sich um einen 32-Bit Timer, der mit Bus-Takt heruntergezählt wird. Neben dem Setzen eines Startwertes verfügt der *APIC-Timer* wie die *RTC* über einen programmierbare Taktteiler, um das Unterbrechungsintervall verändern zu können. Anders als beim *PIT* oder bei der *RTC* lässt sich die *IRQ*-Nummer beim *APIC-Timer* frei wählen.
- Der **High Precision Event Timer** ist ein weiterer Timer, der auf neueren PCs verbaut wird [10]. Dieser Baustein verwendet einen 64-Bit Zähler, der mit mindestens 10 MHz heraufgezählt wird. 3 bis 32 Komparatoren stehen zur Verfügung, um einmalige oder periodische Unterbrechungen mit frei wählbarer *IRQ*-Nummer auszulösen.

2.2.5.2 COM-Port

Beim *COM-Port* des *x86* handelt es sich um eine *RS-232*- bzw. *EIA-232*-Schnittstelle, an der typischerweise Kommunikationsgeräte wie Modems oder Terminals angeschlossen werden. Insbesondere ein Terminal ist dafür geeignet, Textmeldungen auszugeben oder Benutzereingaben zu empfangen.

Wie alle klassischen Geräte der *x86*-Architektur werden die *COM-Ports* über den *I/O*-Adressraum programmiert. Darüber sind mehrere Konfigurationsregister zu erreichen, mit dessen Hilfe sämtliche *UART*-Parameter wie Baudrate, Parität oder Anzahl Datenbits und Stoppbits eingestellt werden können. Ein weiteres Pufferregister dient zum byteweisen Empfangen und Senden von Daten über die Schnittstelle.

2.2.5.3 Color Graphics Adapter

Der *Color Graphics Adapter* (kurz: *CGA*) aus dem Jahr 1981 war die erste echte Grafikkarte für den IBM-PC [7, Kapitel 4.2.2] und etablierte dementsprechend einen Grafikstandard, der bis heute seine Berechtigung hat. So wird die Art der Ansteuerung selbst bei aktuellen Grafikadaptern aus Kompatibilitätsgründen weiterhin unterstützt. Insbesondere die reinen Textmodi werden bis heute genutzt, da jeder grafikfähige PC sie beherrscht.

Die Ansteuerung eines *CGA*-kompatiblen Grafikadapters erfolgt sowohl über den Speicher- als auch den *I/O*-Adressraum. So befindet sich im Bereich von `0xB8000` bis `0xBFFFF` ein 32 KB großes Speicherfenster, um auf den Grafikspeicher zugreifen zu können. Während in den verschiedenen Grafikmodi Pixelgrafiken verwendet werden, bedient man sich in den Textmodi eines festen Zeichensatzes, um Text darzustellen. Die einzelnen Zeichen lassen sich zusätzlich noch mit einem Attribut versehen, um auf 16 verschiedene Farben für Vorder- und Hintergrund sowie verschiedene Auszeichnungsmerkmale zugreifen zu können. Neben diesem Speicherfenster, verwenden alle *CGA*-Karten zwei Register

im *I/O*-Adressraum. Der Wert im ersten Register an `0x3D4` dient als Index, um die internen Steuerregister der Grafikkarte adressieren zu können. Das zweite Register an `0x3D5` wird genutzt, um schließlich Daten in bzw. aus diese Steuerregister schreiben bzw. lesen zu können.

2.2.5.4 AT-Keyboard

Die Tastatur ist das grundlegendste Eingabemedium, dass auf PCs zur Verfügung steht. Da es sich dabei ebenfalls um ein Gerät der „klassischen“ *x86*-Architektur handelt, wird dessen Schnittstelle notfalls von der Firmware emuliert, wenn die Tastatur beispielsweise per *USB* angebunden wurde.

Die Ansteuerung des Tastaturcontrollers läuft wie bei *CGA*-Grafikkarten über zwei Steuerregister, die im *I/O*-Adressraum erreichbar sind. Ein Register übernimmt auch hier die Funktion des Indexregisters, während über das andere Register Daten byteweise ausgetauscht werden. Darüber lassen sich zum einen die *Scancodes* der jeweils gedrückten Tasten auslesen. Zum anderen kann man der Tastatur aber auch Befehle zusenden, die neben Test- und Diagnosefunktionen auch die Steuerung der eingebauten LEDs umfassen.

2.2.6 Peripherie

2.2.6.1 Intel PRO/1000

Bei der *Intel PRO/1000* (kurz: *e1000*) handelt es sich um eine von Intel hergestellte Produktlinie von 1-GBit-Ethernet-Netzwerkkarten. Aufgrund guter Treiberunterstüzung in den aktuellen Betriebssystemen, stellen die meisten Virtualisierungslösungen entsprechende virtuelle Netzwerkadapter zur Verfügung. So bieten beispielsweise *QEMU* und *VirtualBox* jeweils die Emulation einer *Intel PRO/1000 MT Desktop* basierend auf dem *82540EM*-Chipsatz [11][12] an, die im Folgenden genauer betrachtet wird, wobei andere *PRO/1000*-Netzwerkkarten sehr ähnlich aufgebaut sind, sich aber im Detail unterscheiden können.

Zu den Funktionen einer *PRO/1000* gehört unter anderem die *Busmaster*-Fähigkeit, die es der Netzwerkkarte erlaubt, unabhängig von der Zentraleinheit eingehende Pakete in den Hauptspeicher zu schreiben, bzw. ausgehende Pakete vom Speicher zu lesen. Als Besonderheit unterstützt der Adapter zur weiteren Entlastung des Prozessors das *Checksum-Offloading* bei *IP*-, *UDP*- und *TCP*-Paketen, so dass sämtliche Prüfsummen eines Pakets vom Chipsatz der Netzwerkkarte berechnet werden können. Bei TCP-Paketen wird zusätzlich noch das *Segement-Offloading* unterstützt, um größere Datenmengen automatisch in mehrere einzelne TCP-Paketen aufzuteilen.

Ansteuerung

Alle internen Konfigurationsregister der *PRO/1000* werden sowohl im I/O- als auch im Speicheradressraum eingeblendet (*memory-mapped*), dessen Adressen wie bei allen PCI-

Geräten im entsprechenden PCI-Konfigurationsraum zu finden sind. Darüber lassen sich sämtliche Funktionen der Netzwerkkarte steuern. Insbesondere findet man darin auch die jeweiligen Anfangs- und Endzeiger der Sende- und Empfangsringpuffer für Deskriptoren.

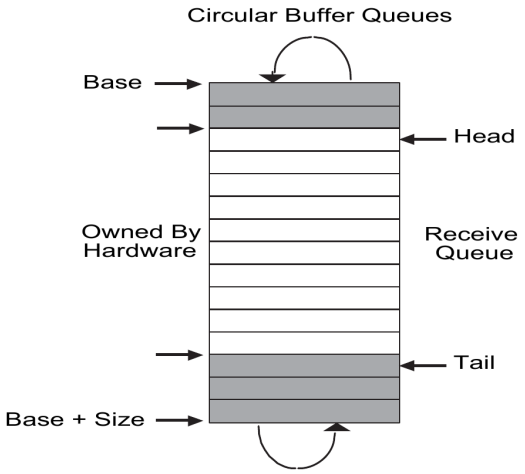

Abbildung 2.11: Empfangs-Deskriptorring[12]

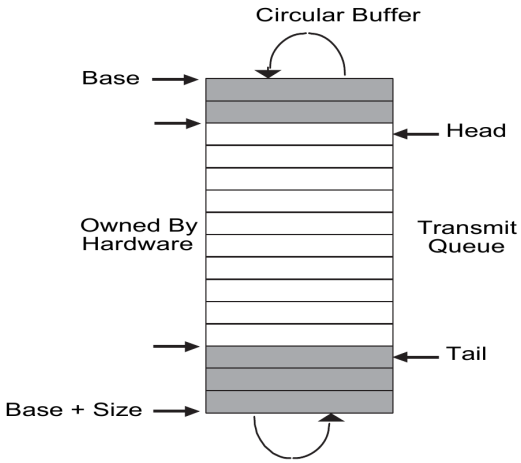

Abbildung 2.12: Sende-Deskriptorring[12]

Die Deskriptoren beinhalten Informationen über den genauen Aufbau der zu verarbeitenden Pakete. Bei den Sendedeskriptoren unterscheidet man zwischen dem *Legacy Transmit Descriptor Format*, das neben den Status- und Befehlsbits nur noch einen Zeiger auf den Paketanfang und die Paketgröße enthält, und dem *TCP/IP Context Format*, das zusätzlich noch Informationen über die Protokollart (*IPv4* oder *IPv6*, *UDP* oder *TCP*) und ggf. die Positionen der jeweiligen Prüfsummen im Paket enthalten kann, um die oben genannten *Offloading*-Funktionen realisieren zu können. Die Empfangsdeskriptoren, die ihrerseits neben den Statusbits nur einen Zeiger auf einen freien Speicherbereich und dessen Größe benötigen, kennen diese Unterscheidung nicht, da der Netzwerkadapter anhand des empfangenen Pakets selbst entscheiden kann, um was für ein Paket es sich handelt. Die Statusbits zeigen dabei an, ob eine Prüfsummenberechnung stattgefunden hat und ob diese erfolgreich war.

2.2.6.2 ATA

Unter *Advanced Technology Attachment* (kurz: *ATA*) versteht man das Kommunikationsprotokoll zwischen dem Controller und *IDE*- (auch *P-ATA* genannt) bzw. *S-ATA*-Datenträger [6, Kapitel 26.4 bis 26.6]. Mit der Festlegung des *ATA*-Protokolls für *IDE*-Geräte ging auch die Standardisierung der „klassischen“ Anbindung der *IDE*-Controller einher. Aus der Sicht eines Betriebssystems reicht also in der Regel ein Treiber aus, um die meisten *IDE*-Controller ansteuern zu können. Selbst viele *S-ATA*-Systeme lassen sich so umkonfigurieren, dass sie sich softwareseitig als *IDE*-Controller ansprechen lassen.

Ansteuerung

Aufgrund der Standardisierung sind die I/O-Adressen der Befehls- und Kontrollregister (auch *Taskfile* genannt) der meisten fest verbauten *ATA*-Controller auf *0x1F0* und *0x3F4* für den primären Kanal und *0x170* und *0x374* für den sekundären Kanal festgelegt. Im Falle von nachgerüsteten *ATA*-Controllern sowie *DMA*-fähigen Controllern kann man die entsprechenden I/O-Adressen dem PCI-Konfigurationsraum des Controllers entnehmen.

PIO

Der *Programmed Input/Output*-Modus (kurz: *PIO*) ist der älteste Zugriffsmodus für *ATA*-Geräte bei dem sämtliche Befehle und Nutzdaten über den I/O-Adressraum gehen. Im einfachsten Fall läuft ein Lese- oder Schreibvorgang im *PIO*-Modus folgendermaßen ab:

1. Die Sektoradresse muss im jeweiligen Format (s. u.) ins Befehlsregister geschrieben werden.
2. Das Lese- oder Schreibkommando muss im Befehlsregister gesetzt werden.
3. 16-Bit Datenblöcke werden wiederholt aus bzw. in das Datenregister gelesen bzw. geschrieben, bis der gesamter Sektor vollständig abgearbeitet wurde.

DMA

Der *Direct Memory Access*-Modus (kurz: *DMA*) ist ein erweiterter Zugriffsmodus und nutzt die Busmaster-Fähigkeit modernerer *ATA*-Controller aus, um Daten zwischen Hauptspeicher und Datenträger auszutauschen. Der größte Nachteil des *PIO*-Modus ist die geringe Übertragungsgeschwindigkeit, die dadurch begründet ist, dass die Zentraleinheit reihenweise 16-Bit-Blöcke explizit lesen und schreiben muss. Ein *DMA*-fähiger *ATA*-Controller nimmt dem Prozessor diese zeitaufwändige Prozedur ab und kümmert sich selbst um den Datenaustausch. Im Falle einer *DMA*-Übertragung sieht eine Transaktion folgendermaßen aus:

1. Ein *Physical Region Descriptor* mit einem Zeiger auf die zu verarbeiteten Daten muss angelegt und im *Busmaster*-Register registriert werden.
2. Die zugehörige Sektoradresse muss im jeweiligen Format ins Befehlsregister geschrieben werden.
3. Ein *DMA*-Lese- oder Schreibbefehl muss abgesetzt werden.
4. Der Controller signalisiert die vollständige Abarbeitung durch eine Unterbrechung, während die Statusregister evtl. vorgekommene Fehler anzeigen.

Adressierungsarten

Die ursprüngliche *ATA-1*-Spezifikation aus dem Jahr 1989 unterstützte nur die klassische Adressierungsweise mit dem **CHS** 3-Tupel, bestehend aus einer Zylindernummer (**C**: *Cylinder*), einer Lese-/Schreibkopfnummer (**H**: *Heads*) und einer Sektornummer (**S**: *Sectors*). Damit können fast 8 GB adressiert werden, was für damalige Verhältnisse mehr als ausreichend war [6, Kapitel 26.5.1]:

$$\overbrace{255}^{Kopf} \times \overbrace{1024}^{Zylinder} \times \overbrace{63}^{Sektor} \times \overbrace{512 Bytes}^{Sektorgröße} = 8.556.380.160 Bytes \approx 7,84 GB$$

Während die Adressbestandteile ursprünglich der physikalischen Eigenschaften einer Festplatte folgten, nahmen sie später nur noch die Rolle einer virtuellen Adresse mit einer beispielsweise theoretisch möglichen aber realitätsfernen Anzahl von maximal 255 Lese-/Schreibköpfe an. Somit folgte ab *ATA-2* der logische Schritt, die 512-Byte Datenblöcke eines Festspeichers einfach der Reihe nach zu nummerieren, da eine Umrechnung der Adresse sowieso vonnöten war.

Das *Logical Block Addressing* (kurz: **LBA**) erweiterte den maximal adressierbaren Bereich auf 128 GB. Dazu wurden die ehemaligen *CHS*-Register erweitert und bilden zusammen eine 28-Bit Sektoradresse mit der bei 512-Byte Sektorgröße insgesamt 128 GB Speicher adressiert werden kann.

Eine erneute Erweiterung der Adressierungsregister im *ATA-6*-Standard auf 48 Bits (**LBA48**) vergrößerte den adressierbaren Bereich auf nun 128 PB [13].

2.3 Virtualisierung

Definitionsgemäß ist es die primäre Aufgabe eines modernen Betriebssystems, die gesamten Betriebsmittel eines Computers zu verwalten und den Anwendungen entsprechende Schnittstellen zur Verfügung zu stellen [14, Kapitel 1]. Deshalb ist es nicht ohne Weiteres möglich, mehrere Betriebssysteme parallel auf einem einzelnen Computer auszuführen. Dabei wäre es gerade bei der Entwicklung eines Betriebssystems wünschenswert, es unter der Kontrolle eines bereits laufenden Systems (auch *Host*-System genannt), ausführen zu können, um so die gegebenen Debugmöglichkeiten ausnutzen zu können. Auch die zu Beginn angesprochenen *Virtual Appliances* bringen ihr eigenens Betriebssystem mit, dass unverändert vom *Host*-System ausgeführt werden soll.

Die *Virtualisierung* löst das Problem, indem sie den weiteren Betriebssystemen (im Folgenden auch *Gast*-Systeme genannt), kontrolliert Zugriff auf Hardwarekomponenten gestattet. Dies ist auch die Abgrenzung zu der *Emulation*, bei der Hardwarekomponenten vollständig in Software nachgebaut werden und man deswegen mit Geschwindigkeitseinbußen rechnen muss.

2.3.1 Herausforderungen

Das parallele Ausführen mehrerer Betriebssysteme ist insofern problematisch, da konventionelle Betriebssysteme nicht damit rechnen, dass andere Instanzen ebenfalls direkte

Hardwarezugriffe tätigen könnten. Ein anderes Problem ist der Einsatz von *privilegierten* Instruktionen, die ein natives Host-System ohne Weiteres nutzen kann, ein Gast-System allerdings nach Möglichkeit vermeiden sollte.

Beide Probleme werden im einfachsten Fall damit gelöst, dass die entsprechenden Hardwarezugriffe und betroffene Instruktionen abgefangen und in Software emuliert werden. Eine höhere Ausführungsgeschwindigkeit erreicht man aber durch den Einsatz einer evtl. vorhandenen prozessorseitigen Virtualisierungsunterstützung oder der Paravirtualisierung.

2.3.2 Hardwaregestützte Virtualisierung

Um die Arbeit der Virtualisierungsprogramme zu unterstützen, wurden neuere *x86*-Prozessoren um einen Betriebsmodus erweitert, der einem Gastsystem teilweise die Ausführung von privilegierten Instruktionen erlaubt. Insbesondere betrifft dies die Befehle der Speicherverwaltung, der Deskriptortabellenverwaltung und der Syscall-Ausführung [15, Sektion 6], die nun ohne Eingriff der Virtualisierungssoftware ausgeführt werden können.

QEMU

Mit *QEMU* steht ein vollständiger *x86*-Emulator zur Verfügung, der neben dem Prozessor auch diverse Peripheriegeräte emulieren kann. Um die Ausführung des Gastsystems zu beschleunigen, bedient sich *QEMU* der *dynamischen Übersetzung* von Binärcode (*binary-translation*) [16].

KVM

Die *Kernel-based Virtual Machine* (kurz: *KVM*) erweitert den Ansatz von *QEMU* um eine vollwertige hardwaregestützte Virtualisierung. Wie der Name bereits suggeriert, wird anders als bei der reinen Softwarelösung von QEMU ein Kernelmodul benötigt, um die Hardwarebeschleunigung nutzen zu können.

2.3.3 Paravirtualisierung

Die Virtualisierung von Peripheriegeräten ist insofern schwierig, da kaum Geräte dafür ausgelegt sind, von mehreren Betriebssystemen zeitgleich angesprochen zu werden. Eine reine Emulation von realen Hardwarekomponenten ist ebenfalls problematisch, da das Gastsystem wegen der Annahme, mit realer Hardware zu kommunizieren, unnötig viele privilegierte Instruktionen für den Hardwarezugriff (insbesondere `in`- und `out`-Instruktionen) nutzen wird. Eine Lösung dieses Problems wäre die *Paravirtualisierung*. Konkret bedeutet diese Art der Virtualisierung, dass sich das Gastsystem nun „bewusst“ ist, dass es virtualisiert wird, und sich entsprechend darauf einstellen kann. So kann die Virtualisierungssoftware einfachere Schnittstellen zur Verfügung stellen, die das Gastsystem statt der echten Gerätetreiber nutzen kann. Dies könnte beispielsweise ein

Speicherbereich sein, auf den sowohl Host- als auch die Gastsysteme zugreifen können. Zusammen mit *KVM* wurde das Paravirtualisierungsframework *Virtio* entwickelt, das nun im Folgenden beschrieben wird.

Virtio ist eine ursprünglich für Linux, später aber auch auf andere Systeme portierte Schnittstelle für paravirtualisierte Peripheriegeräte [17]. Es existieren unter anderem Schnittstellen für „konventionelle“ Geräte wie Festspeicher oder Ethernet-Netzwerke aber auch für recht exotische Geräte, die man nur in virtualisierten Umgebungen findet, wie *Memory-Ballooning*, einer Technik, die es der Virtualisierungssoftware ermöglicht, die Größe des Hauptspeichers des Gastes zur Laufzeit zu ändern.

V-Ringe

Der Datenaustausch zwischen Host- und Gastsystem erfolgt bei Virtio größtenteils mithilfe von zwei Ringpuffern, die zusammen einen *V-Ring* bilden und im Adressraum des Gastes abgelegt und vom Host auf Anforderung verarbeitet werden. Sämtliche Puf-

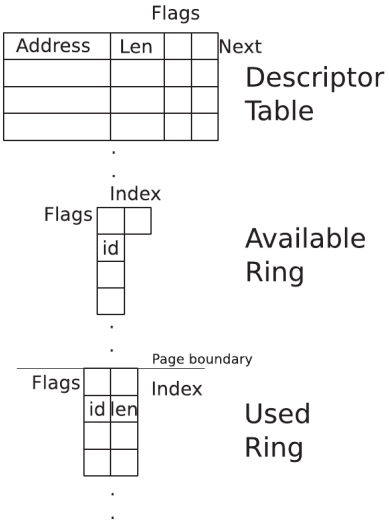

Abbildung 2.13: Virtio Ringpuffer (Quelle: [17, Kapitel 4])

fer, die ein Gast benutzen möchte, werden zunächst in der Deskriptortabelle des V-Rings registriert und anschließend mit Daten gefüllt. Dabei kann es sich um Befehle oder um Nutzdaten für ein Virtio-Gerät handeln. Sobald ein Puffer zur Bearbeitung vom Host präpariert wurde, wird dessen Deskriptornummer im *Available-Ring* eingetragen. Der Host wartet allerdings mit der Abarbeitung, bis er explizit vom Gast - beispielsweise über die *Virtio-PCI-Schnittstelle* - dazu aufgefordert wird, um ein dauerhaftes Pollen des Available-Rings zu vermeiden (siehe Abbildung 2.13). Sobald der Host einen Puffer vollständig verarbeitet hat, wird er in den *Used-Ring* eingefügt und kann vom Gast erneut verwendet werden. Die Verwendung von zwei Ringpuffern hat dabei den Vorteil, dass der Host die Pakete in beliebiger Reihenfolge bearbeiten kann, sofern es einen Sinn ergibt. So könnte beispielsweise ein Virtio-Festspeicher zeitlich spätere Leseanforderungen vorziehen, falls sich diese bereits im Cache des Hosts befinden.

PCI-Schnittstelle

Wie vorher bereits erwähnt wurde, reichen die sich im Speicher befindenden Datenstrukturen nicht aus, um eine effiziente Kommunikation zwischen Host und Gast zu gewährleisten. Das liegt hauptsächlich darin begründet, dass ein einfacher Speicherzugriff des Gastes in der Regel kein Verlassen der Virtualisierungsumgebung und damit eine Eingriffsmöglichkeit seitens des Hostes bewirkt. Solche Umschaltvorgänge zwischen Gast und Host haben relativ hohe Kontextwechselkosten zur Folge und daher gilt es, diese weitestgehend bei der Virtualisierung zu vermeiden. Im oben beschriebenen Fall des V-Rings ist ein Eingriff des Hosts aber durchaus erwünscht, weshalb hier ausnahmsweise privilegierte Instruktionen genutzt werden können. Da praktisch alle handelsüblichen Betriebssysteme bereits eine Unterstützung des PCI-Busses beinhalten, bietet es sich natürlich an, sämtliche Virtio-Geräte als virtuelle PCI-Geräte einzubinden [17, Kapitel 6]. Der Großteil der zu übertragenen Daten wird nach wie vor über den V-Ring abgewickelt. Ausschließlich Steueranweisungen, wie eine Verarbeitungsaufforderung seitens des Gastes, werden über den virtuellen PCI-Bus abgehandelt.

Virtio-Geräte

- **Virtio-Netzwerkadapter**

  Bei dem Virtio-Netzwerkadapter handelt es sich um eine stark vereinfachte virtuelle Netzwerkkarte, die ganz auf die Bedürfnisse von Virtualisierungslösungen zugeschnitten ist. Es werden insgesamt zwei V-Ringe verwendet, jeweils einen für den Paketempfang und dem Paketversand. Jede Operation benutzt mindestens zwei Deskriptoren. Der erste Deskriptor dient immer der Konfiguration. Hier lassen sich ähnlich zum *TCP/IP Context Descriptor* der *PRO/1000*, die in Kapitel 2.2.6.1 beschrieben wurden, die Parameter für die *Offloading*-Features einstellen, die der Virtio-Netzwerkadapter ebenfalls zur Verfügung stellt. Alle folgenden Deskriptoren referenzieren die zu übertragenden Nutzdaten.

- **Virtio-Festspeicher**

  Das Virtio-Festspeichermodul bietet einem Gastsystem eine einfache Schnittstelle für virtuelle Festplatten an. Ähnlich wie bei dem Virtio-Netzwerkadapter ist es auch Ziel dieses Moduls, ein häufiges Eingreifen des Hostsystems zu vermeiden. Die Schnittstelle nutzt dazu einen V-Ring, der sowohl Lese- als auch Schreiboperationen aufnimmt. Eine Operation wird immer mit einem Befehlsdeskriptor eingeleitet. Dieser beinhaltet zunächst ein Feld über die Art der Operation. Dabei kann es sich einerseits um einen einfachen Lese- oder Schreibbefehl handeln. Andererseits definiert Virtio aber auch die Möglichkeit, generische *SCSI*-Kommandos anzugeben, was beispielsweise dazu genutzt werden kann, um virtuelle CD-ROMs auszuwerfen [16, Kapitel 5.1]. Weiterhin enthält der einleitende Befehlsdeskriptor eine Sektoradresse, auf die sich die auszuführende Operation beziehen soll, und ein Prioritätsfeld, um dem Host Hinweise über die Dringlichkeit einer Operation zu geben. Nach dem Befehlsdeskriptor folgen im V-Ring ein oder mehrere Puffer,

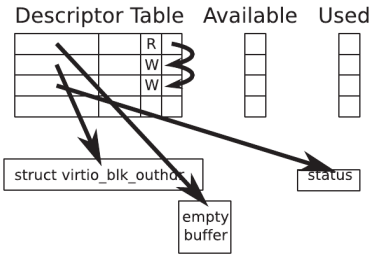

Abbildung 2.14: Virtio Block Device (Quelle: [17, Kapitel 5.1])

die die Nutzdaten enthalten. In Abbildung 2.14 ist der V-Ring für das Virtio-Festspeichermodul abgebildet.

3 Analyse

Das folgende Kapitel beschäftigt sich mit einer Analyse der neu zu entwickelnden Variante der Betriebssystemfamilie *CiAO*. Hierbei werden die einzelnen Komponenten betrachtet, die im Rahmen dieser Arbeit entstehen werden. Jede Komponente wird auf mögliche Merkmale hin untersucht, die in der Implementierung maßgeblich die Konfigurationsmöglichkeiten beeinflussen werden. Weiterhin werden auch die Auswirkungen der einzelnen Belange auf das restliche System analysiert, um ausreichend Anhaltspunkte für den aspektorientierten Entwurf zu gewinnen.

Das Kapitel unterteilt sich in drei Abschnitte. Zunächst wird die grundlegende Systemarchitektur der neuen *CiAO*-Variante betrachtet. Anschließend werden die architekturellen Besonderheiten der *x86*-Architektur dahingehend untersucht, wie die neuen Funktionen, wie beispielsweise das *Paging*, unter *CiAO* nutzbar gemacht werden können. Zum Schluss werden die Treiber zu den Peripheriegeräten aus Kapitel 2.2.6 analysiert.

3.1 Grundlegende Architektur

In Abbildung 3.1 wird das Merkmalmodell der *x86*-Variante von *CiAO* dargestellt. Der Benutzer des Systems sollte unabhängig von der sonstigen Konfiguration die gewünschte *x86*-Unterarchitektur (*32-Bit* oder *64-Bit*) auswählen können. Zusätzlich hat der Anwender grundsätzlich die Wahl zwischen einem *nativen* System und einem *Gastsystem*. Die jeweiligen konfigurierbaren Treiber beider Systeme werden später genauer analysiert. Im Folgenden werden zunächst die Herausforderungen der verschiedenen Arten einer Umsetzung von *CiAO* auf die *x86*-Architekturen diskutiert. Dabei werden insbesondere die *native* und die *Gast*-Umsetzung miteinander verglichen. Im Anschluss werden die Unterschiede des *64-Bit*-Systems zur herkömmlichen Variante behandelt.

3.1.1 CiAO als natives System

Ein Betriebssystem dient im Normalfall als direkter Mittler zwischen Anwendungen und den zur Verfügung stehenden Betriebsmitteln [14, Kapitel 1]. Demzufolge wird ein Betriebssystem, abgesehen von der Firmware der Plattform, ohne weitere Softwareschichten direkt auf der Hardware ausgeführt, was in Abbildung 3.2 dargestellt wird. Zur Applikation hin bietet das Betriebssystem eine standardisierte Schnittstelle an, die von sämtlichen Eigenheiten der verwendeten Hardware abstrahiert. Die *native CiAO*-Variante soll genau diese Aufgabe übernehmen. Sie soll in der Lage sein, ohne die Hilfe einer zugrundeliegenden Systemsoftware, auf einem *IBM-kompatiblen PC* ausgeführt zu werden. Einzig allein ein externer Bootloader wie *GRUB* (siehe Kapitel 2.2.4) darf benutzt werden, um

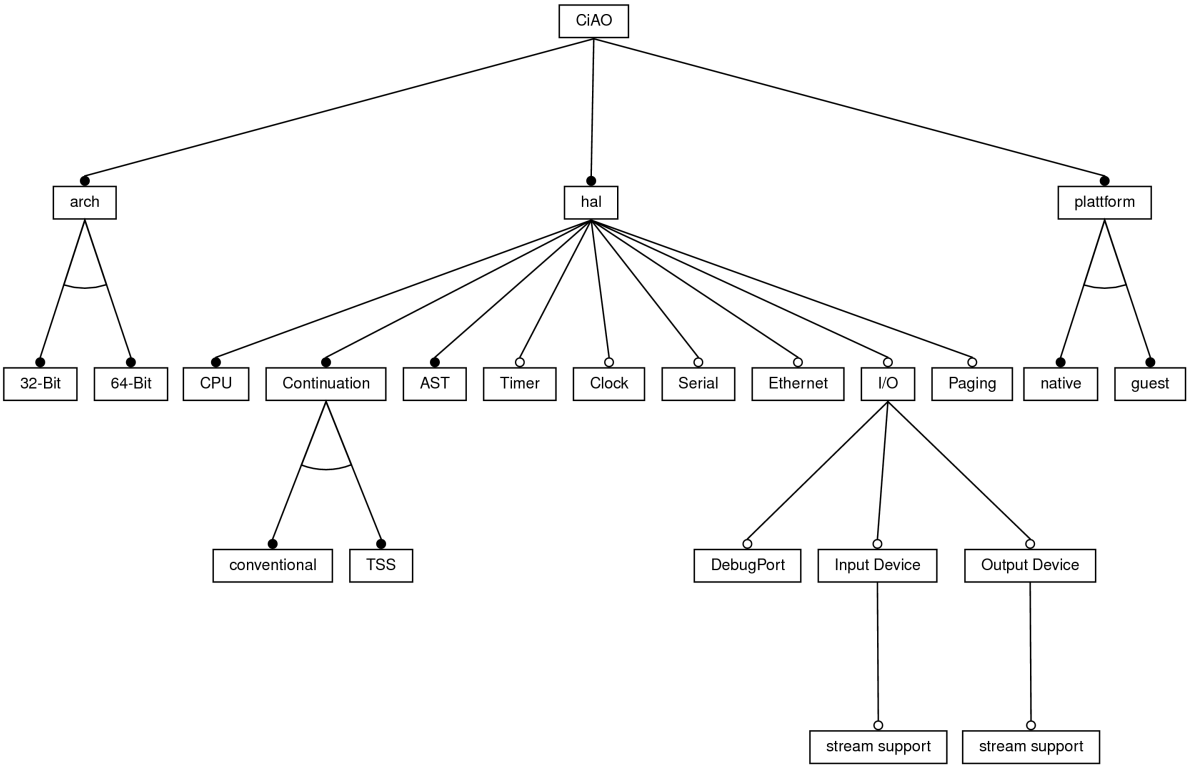

Abbildung 3.1: *x86*-Variante von *CiAO*

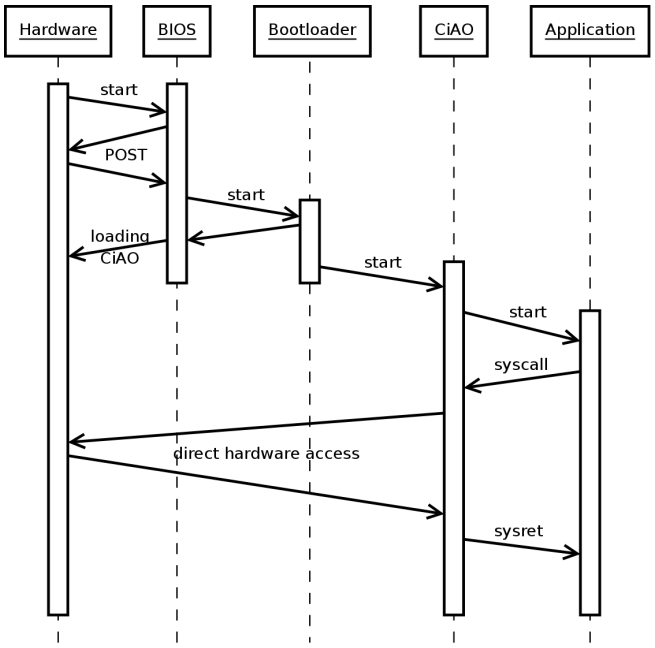

Abbildung 3.2: Ablaufmodell der nativen *CiAO*-Variante

eine komfortable Möglichkeit zur Verfügung zu stellen, einen Systemkern zu laden, da die Firmware dazu selbst nicht in der Lage ist.

Das *native System* in der 32-Bit Ausführung dient als Grundlage für alle weiteren Ausprägungen des Systems. Sämtliche Auswirkungen der der im Folgenden beschriebenen Untervarianten beziehen sich immer auf dieses System.

Merkmalmodell

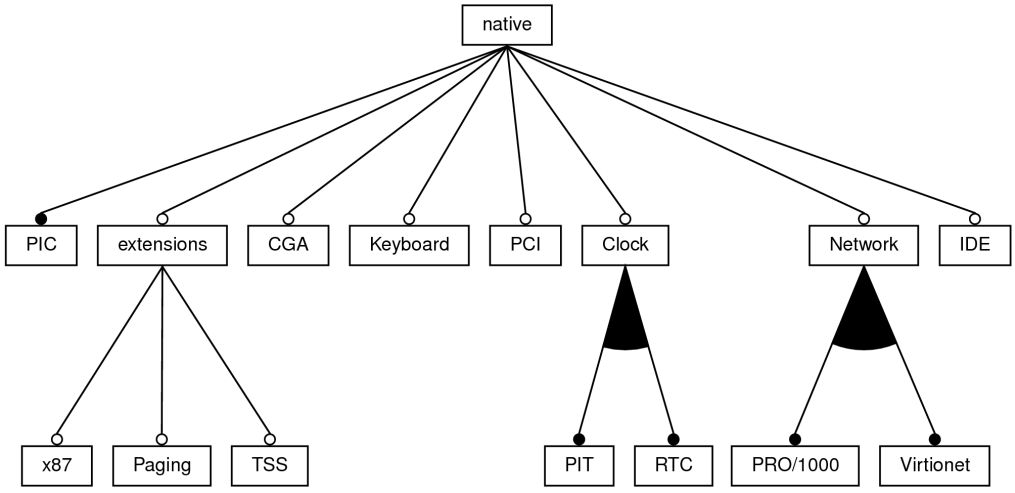

Abbildung 3.3: Merkmalmodell der nativen *CiAO*-Variante

Das in Abbildung 3.3 dargestellte Merkmalmodell zeigt alle Gerätetreiber der nativen *CiAO*-Variante, wobei die Untermerkmale der einzelnen Treiber später genauer betrachtet werden.

3.1.2 CiAO als Linux-Gastsystem

Im Gegensatz zu einem nativen System wird ein *Gastsystem* als normale Applikation auf einem *Hostbetriebssystem* ausgeführt. So erreicht man ähnlich wie bei der Virtualisierung, die in Kapitel 2.3 beschrieben wurde, dass ein zweites Betriebssystem als Gast unter einem Hostsystem ausgeführt werden kann. Im Unterschied zur reinen Virtualisierung benötigt das *Linux-Gastsystem* keine spezielle Virtualisierungssoftware, auch *Hypervisor* genannt, die für die Hardwarezuteilung zuständig ist. Statt dessen nutzt ein *Linux-Gastsystem* die normale *POSIX-API*, die von Linux zur Verfügung gestellt wird. Um die Kompatibilität zum nativen System zu wahren, muss die Hardwareabstraktionsschicht zur Gastanwendung hin dieselben Funktionen anbieten. Diese werden aber anders als beim nativen System über das Hostsystem abgewickelt, dass die alleinige Kontrolle über die Hardware ausübt. In Abbildung 3.4 erkennt man im Vergleich zum vorher beschriebenen *nativen* System diese zusätzliche „Schicht“ des Hostsystems.

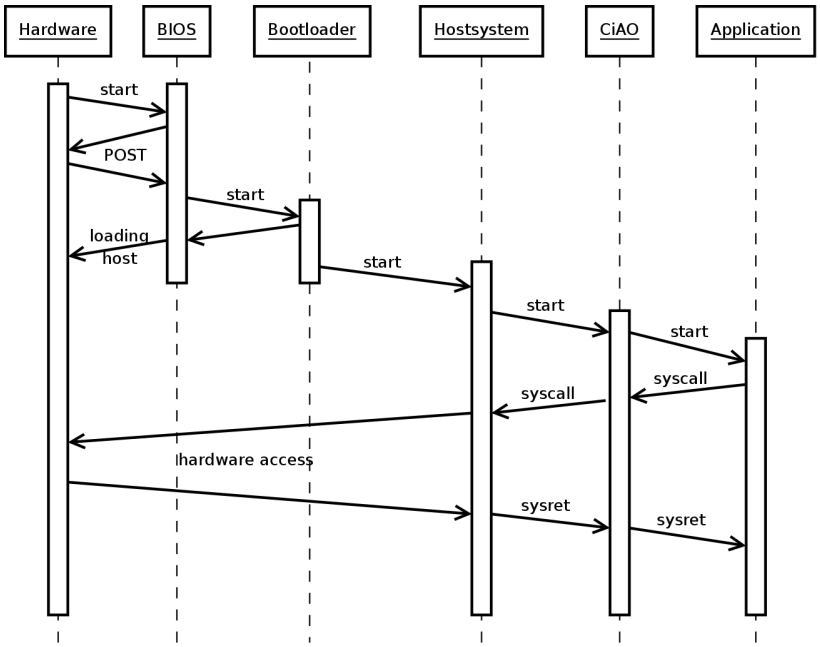

Abbildung 3.4: Ablaufmodell der *CiAO*-Gast-Variante

Vergleich zum nativen System

Wenn CiAO als Gast unter einem Hostsystem ausgeführt wird, unterliegt es wie jede andere normale Applikation auch den geltenden Zugriffsbeschränkungen für den *Userspace*. So werden sämtliche direkte Hardwarezugriffe, wie sie von den herkömmlichen Gerätetreibern vorgenommen werden, effektiv unterbunden. Innerhalb des *CiAO*-Gastsystems werden daher sogenannte *Syscall-Wrapper* die Aufgabe der Gerätetreiber übernehmen. Diese bieten den Gastapplikationen eine definierte Schnittstelle zu den Funktionen des Hostsystems an. So wird beispielsweise ein Textausgabetreiber, der auf dem nativen System direkt die Grafikhardware ansteuert, unter dem Gastsystem ausschließlich Bibliotheksfunktionen des Hostsystems für den Zugriff auf die Standardausgabe nutzen.

Merkmalmodell

In Abbildung 3.5 werden die *Syscall-Wrapper* der Gast-Variante von *CiAO* dargestellt.

Auswirkungen

Initialisierung **(guest1)** Beim Start des nativen Systems sorgt die Initialisierungsroutine dafür, dass alle wichtigen Systemkomponenten initialisiert sind, bevor die Kontrolle an das eigentliche Betriebssystem übergeben wird. Das Gastsystem kann davon ausgehen, dass diese Komponenten vom Hostsystem bereits korrekt initialisiert wurden und daher kein weiterer Eingriff notwendig ist. So kann die Initialisierungs-

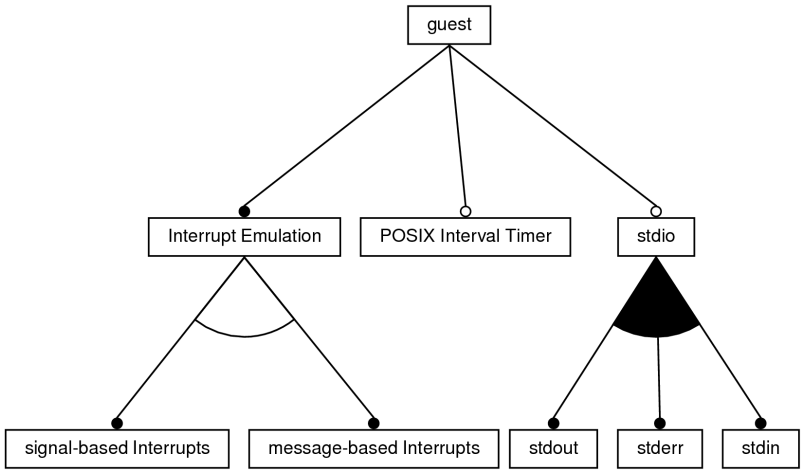

Abbildung 3.5: Merkmalmodell der *CiAO*-Gast-Variante

routine des Gastsystems direkt den Gastsystemkernel starten. Eventuell müssen vorher noch die verwendeten dynamischen Bibliotheken initialisiert werden.

Treiber (guest2) Da jeder Hardwarezugriff über die Hostsystem-API abgebildet werden muss, benötigt die *CiAO*-Gastimplementierung einen vollständig Satz an eigenen Gerätetreibern. Die Implementierung dieser Treiber fällt in der Regel deutlich einfacher aus, da das Hostsystem für die meisten Peripheriegeräte bereits eine erprobte Schnittstelle anbietet.

3.1.3 64-Bit-Unterstützung

Die *64-Bit-Unterstützung* soll sämtliche Neuerungen des *Long-Modes*, der in Kapitel 2.2.1 beschrieben wurde, unter *CiAO* nutzbar machen. Als eigenständiger Betriebsmodus ist der *64-Bit-Modus* nur bedingt mit dem *32-Bit-Modus* kompatibel. Dieser Umstand ist aber nicht weiter problematisch, da der Compiler die Aufgabe übernimmt, die passenden Anweisungen für die jeweilige Architektur zu generieren. Da die zugrundeliegende Hardware nach wie vor identisch ist, können die Routinen zur Ansteuerung in den meisten Fällen unverändert übernommen werden.

Auswirkungen

Initialisierung (lm1) Bevor *CiAO* überhaupt nativ gestartet werden kann, muss die Initialisierungsroutine zunächst den *Long-Mode* aktivieren. Dazu ist wiederum das frühzeitige Aktivieren des *64-Bit-Pagings* innerhalb der Startroutinen notwendig.

Diese Veränderungen entfallen bei der Konfigurierung eines *Linux-Gastsystems*, da der Host selbst den benötigten Modus vor Ausführung des Gastsystems aktiviert.

Kontextwechsel (lm2) Da sich die Anzahl der Register, die Registerlänge und die Aufrufkonvention grundlegend verändert haben, benötigt *CiAO* für die 64-Bit-Umsetzung

eigene Kontextwechselroutinen und jeweils passende Datenstrukturen die den gesicherten Kontext aufnehmen können.

Da ein Kontextwechsel über das *TSS* im 64-Bit-Modus nicht mehr unterstützt wird, entfallen die dazu zugehörigen Merkmale.

Paging (lm3) Aufgrund des neuen Pagingformates im 64-Bit Betriebsmodus müssen die Datenstrukturen und Zugriffsfunktionen der Seitentabelle entsprechend angepasst werden, da im *Long-Mode* ausschließlich das *64-Bit-Paging* genutzt werden kann.

3.2 Besonderheiten der Architektur

Die *x86*-Architektur weist einige Besonderheiten auf, die in *CiAO* bis jetzt nicht berücksichtigt wurden. Diese werden im Folgenden genauer betrachtet, insbesondere unter dem Gesichtspunkt, wie sich solche neuen Komponenten in das bestehende System integrieren lassen.

3.2.1 *x87* Gleitkommazahleinheit

Die Gleitkommazahleinheit der *x86*-Architektur war ursprünglich als mathematischer Coprozessor ausgelegt, der bei Bedarf nachgerüstet werden konnte. Der somit von der Zentraleinheit getrennte Baustein besaß deshalb einen eigenen Befehls- und Registersatz.

Seit dem Erscheinen des Pentium-Prozessors ist die Gleitkommazahleinheit allerdings fester Bestandteil der Zentraleinheit. Aus Kompatibilitätsgründen wurde die getrennte Architektur aus Sicht der Software beibehalten, die sich insbesondere bei der Initialisierungssequenz und dem Sichern des internen Zustandes widerspiegelt.

Auswirkungen

Initialisierung (fpu1) Um die Gleitkommazahleinheit benutzen zu können, muss sie vor der ersten Verwendung initialisiert werden. Da sämtliche Komponenten des Betriebssystem ohne Gleitkommazahlberechnungen auskommen, kann dies bei der Initialisierung der einzelnen Tasks geschehen.

Taskkontext (fpu2) Weiterhin muss bei jedem Kontextwechsel der vollständige Zustand der Einheit gesichert und wiederhergestellt werden. Dazu ist eine Erweiterung der Taskkontextstruktur innerhalb der *Continuation*-Klasse notwendig.

***TSS*-Kontextwechsel (fpu3)** Falls der *TSS*-Taskwechsel genutzt wird, muss nach dem Wiederherstellen des Kontextes zwingend das *task switched*-Bit gelöscht werden, andernfalls löst die nächste Gleitkommazahloperation eine Trap aus. Diese Trap dient der Optimierung, um das Wiederherstellen des Kontextes der Gleitkommazahleinheit so lange hinauszuzögern, bis sie erneut benutzt wird. Dies spart unter Umständen das Laden und Speichern der Gleitkommazahlregister, falls diese über

einen längeren Zeitraum nicht genutzt werden. Dieser Ansatz wird in dieser CiAO-Variante allerdings nicht weiter berücksichtigt, da bereits zum Konfigurationszeitpunkt entschieden wird, ob Gleitkommazahloperationen benötigt werden.

3.2.2 TSS-Taskwechsel

Die *x86*-Architektur besitzt neben der konventionellen Möglichkeit des Kontextwechsels, bei dem manuell sämtliche relevante Registerinhalte gesichert werden müssen, eine spezielle hardwaregestützte Form der Zustandssicherung und -wiederherstellung. Anstatt jedes Register mit einzelnen `mov`-Instruktionen in den Speicher zu schreiben, kann der *x86* im 32-Bit Modus automatisch alle Allzweck- und Segmentregister, das Statusregister, den Instruktionszeiger und den Zeiger auf die Seitentabelle sichern und wiederherstellen [9, Kapitel 7]. Die dazu notwendigen Informationen werden in einem *Task-State-Segment* abgelegt.

Auswirkungen

Taskkontext (tss1) Die jeweiligen Taskzustände werden beim *TSS*-Taskwechsel in spezielle Segmente geschrieben. Diese *Task-State-Segmente* ersetzen die konventionellen Kontextstrukturen der *Continuation*-Klasse.

Kontextwechsel (tss2) Beim Kontextwechsel müssen nun nicht mehr manuell alle relevanten Register gesichert und wiederhergestellt werden. Die zuständigen Funktionen müssen stattdessen mit Hilfe eines *Task-Gates* einen hardwaregestützten Taskwechsel anstoßen [9, Kapitel 7.2.5].

Deskriptortabelle (tss3) Jedes *Task-State-Segment* muss vor Verwendung in die *Globale Deskriptor Tabelle* eingetragen werden. Dies kann während der Initialisierung eines Tasks geschehen.

3.2.3 seitenbasierte Speicherverwaltung

Die *seitenbasierte Speicherverwaltung*, auch *Paging* genannt, wurde in Kapitel 2.2.2 bereits beschrieben. *CiAO* bietet bislang keine Möglichkeit, Seitenzuordnungen zu verwalten. Diese müssen im Rahmen der *Paging*-Schnittstelle ergänzt werden.

Konfigurationspunkte

Im 64-Bit-Modus muss zwingend das *64-Bit-Paging* verwendet werden, so dass es in diesem Fall keine Auswahlmöglichkeit für den Benutzer geben wird. Hier kann höchstens auswählt werden, ob die *Paging*-Schnittstelle hinzukonfiguriert werden soll, um die Seitentabelle zur Laufzeit modifizieren zu können.

Im 32-Bit-Modus sollte die Verwendung der seitenbasierten Speicherverwaltung hingegen rein optional sein. Bei Verwendung sollten dem Benutzer sowohl das *32-Bit-Paging* als auch das *PAE-Paging* als gleichwertige Alternativen zur Verfügung stehen (mmu1).

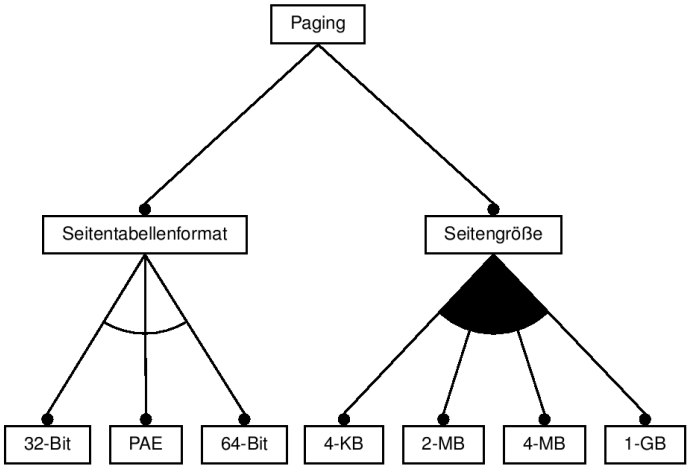

Abbildung 3.6: Merkmalmodell des Pagings

Je nach verwendetem Tabellenformat stehen schließlich noch verschiedene Seitengrößen zur Verfügung. So unterstützen alle Formate die 4-KB-Seitengröße. Das *32-Bit-Paging* bietet zusätzlich noch 4-MB-Seiten an. Die *PAE*- und *64-Bit*-Formate erlauben stattdessen 2-MB-Seiten. Dem *64-Bit-Paging* stehen schließlich mit zusätzlich 1-GB-Seiten gleich drei verschiedene Seitengrößen zur Verfügung. Der Anwender soll zwischen den verfügbaren Formaten frei wählen können.

In Abbildung 3.6 werden diese Konfigurationsmöglichkeiten nochmal als Merkmalmodell dargestellt.

Auswirkungen

Initialisierung (mmu2) Während der Systeminitialisierung muss mindestens eine Seitentabelle angelegt werden, die weiterhin Zugriff auf die Funktionen und die Datenstrukturen des Systemkerns zulässt. Ohne besondere Vorkehrungen ist der *CiAO*-Kern nicht relozierbar. Dementsprechend darf das Aktivieren der Paging-Einheit die virtuelle Position des Systems im Speicher nicht verändern.

Speicherzugriffe (mmu3) Sobald die Paging-Einheit aktiviert wurde, unterliegen sämtliche Speicherzugriffe der Adressumrechnung über die Seitentabelle. Unzulässige Zugriffe auf gesperrte oder nicht zugewiesene Speicherbereiche lösen eine *Pagefault-Exception* (kurz: *#PF*) aus, die bei Bedarf durch einem von *CiAO* zur Verfügung gestellten *Protection-Hook* behandelt werden kann [19, Kapitel 2.1.2].

3.2.3.1 seitenbasierter Speicherschutz

Mit Hilfe der *seitenbasierten Speicherverwaltung* sollen sich unzulässige Speicherzugriffe unterbinden lassen.

Auswirkungen

Initialisierung (mp1) Bei der Initialisierung eines Tasks, muss jeweils eine neue Seitentabelle aufgebaut werden, die die Zugriffsberechtigungen der Applikation berücksichtigen.

Kontextwechsel (mp2) Um für jeden Task oder jede Applikation unterschiedliche Berechtigungen realisieren zu können, werden jeweils taskeigene Seitentabellen benötigt. Diese sind Teil der jeweiligen Taskkontexte und müssen entsprechend gesichert und wiederhergestellt werden.

Systemaufrufe (mp3) Die Systemaufrufe benötigen in der Regel vollen Systemzugriff, um ihre jeweilige Aufgabe erfüllen zu können. Da die Seitentabellen der Applikationen normalerweise keine Speicherzugriffe auf kernelinterne Strukturen zulassen, muss vor jedem Systemaufruf auf eine kerneleigene Seitentabelle umgeschaltet werden. Nach der Ausführung des Systemaufrufes muss die ursprüngliche Seitentabelle wiederhergestellt werden.

Unterbrechungsbehandlung (mp4) Sobald eine Unterbrechung, eine Exception oder eine Trap auftritt, wird die aktuelle Programmausführung unterbrochen, um die zuständigen Behandlungsroutinen auszuführen. Dabei beginnt die Ausführung zunächst mit der vor dem Auftreten der Unterbrechung verwendeten Seitentabelle. Unter Umständen muss hier ebenfalls eine Umschaltung zur Kernelseitentabelle erfolgen.

3.3 Treiber

Die *Gerätetreiber* gehören zu den unverzichtbaren Bestandteilen jedes Betriebssystems. Nur mit dessen Hilfe können Peripheriegeräte ordnungsgemäß angesteuert werden. Im Folgenden werden insbesondere die Treiber mit Konfigurationsmöglichkeiten seitens des Benutzers erläutert.

3.3.1 AT-Keyboard

Als Möglichkeit zur Texteingabe unterstützen alle PCs den Anschluss einer Tastatur, die in Kapitel 2.2.5.4 beschrieben wurde.

Konfigurationspunkte

Der wichtigste Konfigurationspunkt des Tastaturtreibers ist das zu verwendende Layout, dass sich je nach Sprache unterscheiden kann (kb1). Ansonsten sollte der Treiber noch optionale Ansteuerfunktionen für die LEDs bereit stellen, sofern sie benötigt werden. Diese Punkte werden in Abbildung 3.7 dargestellt.

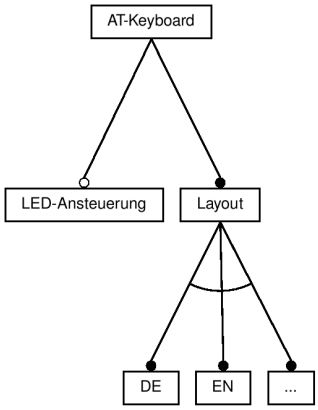

Abbildung 3.7: Merkmalmodell des Tastaturtreibers

Auswirkungen

Initialisierung (kb2) Die AT-Tastatur benötigt keine zwingende Initialisierungsroutinen, da das *PC-BIOS* bereits eine ausreichende Initialisierung vornimmt. Allerdings bietet es sich an, ein Reset durchzuführen, damit sich die Tastatur in einem fest definierten Zustand befindet.

3.3.2 *Intel PRO/1000 MT Desktop*

Die auf dem *Intel 82640EM* basierende *Intel PRO/1000 MT Desktop* ist eine 1000-MBit-Ethernet-Netzwerkkarte, dessen Aufbau bereits im Kapitel 2.2.6.1 beschrieben wurde.

Konfigurationspunkte

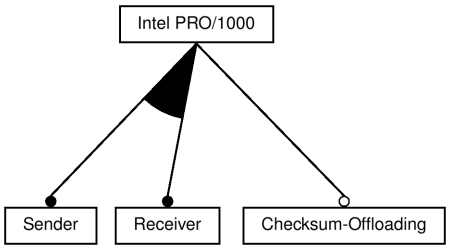

Abbildung 3.8: Merkmalmodell des *Intel PRO/1000*-Treibers

Die *Intel PRO/1000* Netzwerkkartenproduktlinie bietet neben den einfachen Sende- und Empfangsfunktionen (e1) die Möglichkeit, aufwändige Berechnungen optional vom Netzwerkadapter durchführen zu lassen (e2). Abbildung 3.8 zeigt das zugehörige Merkmalmodell.

Auswirkungen

Initialisierung (e3) Bevor die Netzwerkkarte genutzt werden kann, ist eine einmalige Initialisierungssequenz notwendig. Diese führt einen Reset der Netzwerkhardware aus, um sämtliche internen Register auf die definierten Anfangswerte zu setzen.

Paging (e4) Da die internen Register der Netzwerkkarte in den Speicheradressraum abgebildet werden, muss unter Umständen die Seitentabelle um einen Eintrag auf den entsprechenden Speicherbereich erweitert werden.

3.3.3 IDE-Treiber

Der *IDE-Treiber* dient zur Ansteuerung der in Kapitel 2.2.6.2 beschriebenen *P-ATA*-Festplatten. Als Basis für zukünftige Dateisystemtreiber, muss dieser Treiber Lese- und Schreiboperationen auf beliebige Sektoren des Mediums bereitstellen.

Konfigurationspunkte

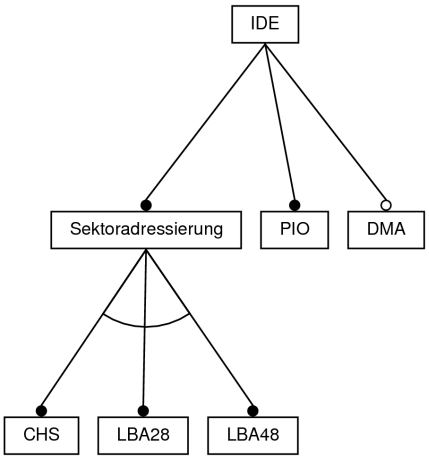

Abbildung 3.9: Merkmalmodell des *IDE*-Treibers

Moderne *IDE*-Controller unterstützen alle drei Adressierungsarten, die im Grundlagenkapitel bereits beschrieben wurden. Da eine Adressierungsweise ausreicht, um ein Medium ansteuern zu können, bietet es sich an, eine Auswahlmöglichkeit zum Konfigurationszeitpunkt anzubieten (ide1). Neben der *PIO*-Datenübertragung, die zu Initialisierungszwecken benötigt wird und daher immer vorhanden sein muss, soll auch eine optionale *DMA*-Erweiterung angeboten werden (ide2).

Auswirkungen

Initialisierung (ide3) Der Initialisierungszeitpunkt kann genutzt werden, um durch die einzelnen verfügbaren IDE-Kanäle zu iterieren. So erfährt der Treiber, welche Lauf-

werke angeschlossen wurden. Jedes IDE-Laufwerk unterstützt ein *Identifikationskommando*, um weitere Informationen wie beispielsweise die Speicherkapazität zu ermitteln [13].

4 Konzeption und Realisierung

In diesem Kapitel geht es um den Entwurf der *x86*-Variante von *CiAO*. Im ersten Abschnitt werden zunächst die neu entwickelten Schnittstellen der Hardwareabstaktionsschicht erläutert. Anschließend werden die grundlegenden Anpassungen beschrieben, die notwendig sind, um *CiAO* an eine neue Plattform anzupassen. Dies betrifft hauptsächlich die bereits vorhandenen *HAL*-Schnittstellen, die innerhalb des Kerns genutzt werden. Das dritte Unterkapitel beschäftigt sich anschließend mit der Realisierung der *Gast*-Variante von *CiAO*, die unter einem konventionellen *x86*-Linux-System lauffähig ist. Dabei werden auch einige einfache Gastsystem-Treiber vorgestellt, um die konzeptionellen Unterschiede zwischen „echten“ Gerätetreibern und Gastsystem-Treibern deutlich zu machen. Im nächsten Unterkapitel wird die Unterstützung für die 64-Bit-Unterarchitektur des *x86* innerhalb von *CiAO* vorgestellt. Dabei werden insbesondere die Unterschiede zwischen dem 32-Bit- und dem 64-Bit-Betriebsmodus hervorgehoben. Das letzte Unterkapitel geht schließlich auf den seitenbasierten Speicherschutz ein, der mit Hilfe der neu entworfenen *Paging*-Schnittstelle entwickelt wurde.

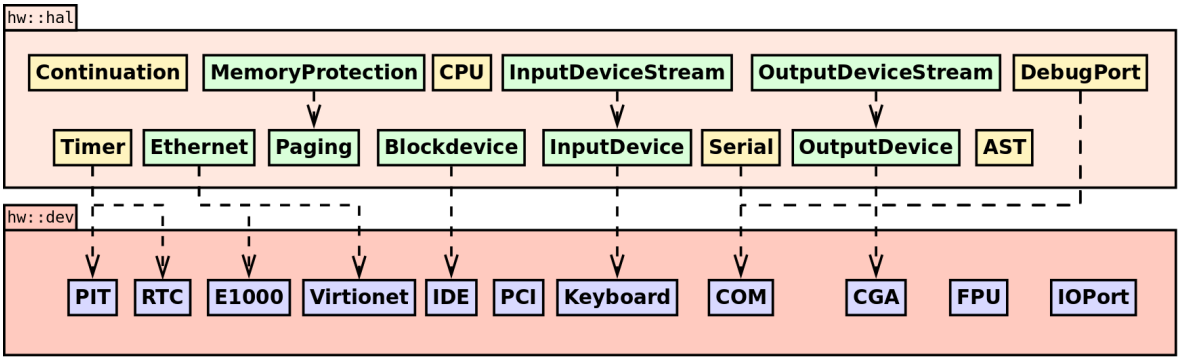

Abbildung 4.1: Überblick über die *CiAO*-Hardwareschichten

4.1 Schnittstellen

Da *CiAO* ursprünglich für die *TriCore*-Architektur entwickelt wurde, fehlen innerhalb der *HAL* teilweise Schnittstellen zu Peripheriegeräten, die innerhalb der *x86*-Architektur gebräuchlich sind. Die neuen Schnittstellen, die in Abbildung 4.1 grün eingefärbt sind, wurden im Rahmen dieser Arbeit entwickelt. Ergänzend dazu wird ebenfalls eine konfigurierbare Beispielimplementierung, aufbauend auf den jeweiligen Gerätetreiber, vorgestellt. Die bereits vorhandenen in der Abbildung gelb eingefärbten Schnittstellen wurden

nach Möglichkeit ebenfalls umgesetzt, sofern die entsprechende Hardware auf der *x86*-Architektur verfügbar war.

4.1.1 Input- und OutputDevice(Stream)

Die Möglichkeit der formatierten Ein- und Ausgabe sind auf eingebetteten Systemen oft sehr beschränkt. Auf PC-Architekturen gehören sie aber mindestens zum Normalfall eines Systems. *CiAO* bietet zumindest für die formatierte Ausgabe einen *DebugPort* in der Form eines minimalen `printf`s an. Dieser ist für einen normalen Programmablauf aber nicht zu empfehlen, da die Zeichenausgabe zum einen ungepuffert und zum anderen unter Sperrung sämtlicher Unterbrechungen erfolgt. Ebenfalls sind mit dem *DebugPort* keine Eingaben möglich.

Da Konzepte wie *Pufferung* oder *Integer-Text-Konvertierung* im Gegensatz zu der eigentlichen Zeichenein- und ausgabe nicht architekturabhängig sind, bietet es sich an, die Ein- und Ausgabekomponenten in einen plattformabhängigen Teil und in einen plattformunabhängigen Teil aufzuteilen:

InputDevice und **OutputDevice** stellen dabei allgemeine Schnittstellen für die ungepufferte Ein- und Ausgabe zur Verfügung, ohne allerdings Unterbrechungen zu sperren. Diese müssen wegen der ihrer hardwarenähe plattformspezifisch angepasst werden.

Ein *OutputDevice* stellt folgende Methoden zur Verfügung:

`clearScreen()`	löscht den Bildschirminhalt
`setCursorPosition(x, y)`	verschiebt den Cursor nach (x,y)
`putCharacter(c)`	Gibt ein Zeichen aus
`putString(str, size)`	Gibt eine Zeichenkette aus
`setAttribute(attribute)`	Setzt das Attribut
`resetAttribute()`	Setzt das Attribut zurück

Ein *InputDevice* stellt ergänzend dazu folgende Methoden zur Verfügung:

`echo(on)`	De/Aktiviert das Echo eingegebener Zeichen
`peekCharacter()`	Gibt aktuelles Zeichen zurück, ohne es zu aus Puffer löschen
`getCharacter()`	Gibt aktuelles Zeichen zurück und löscht es aus Puffer
`charactersAvailable()`	Gibt die Anzahl der Zeichen im Puffer zurück

Diese *HAL*-Funktionen können im einfachsten Fall direkt auf die jeweiligen Gerätetreiber für Textausgaben und -eingaben abgebildet werden. Es handelt sich also um reine Adapterklassen. Die `printf()`-Funktion des *DebugPort*s lässt sich auf ähnliche Art und Weise entweder auf das *OutputDevice* oder ebenfalls direkt auf die Gerätetreiber abbilden.

InputDeviceStream und **OutputDeviceStream** ergänzen die Klassen um jeweils einen eigenen Puffer und der Möglichkeit Zahlen formatiert ein- und auszugeben, wobei die Schnittstelle bewusst an den aus der C++-Standardbibliothek bekannten `std::istream`- und `std::ostream`-Klassen angelehnt und als `<<`- und `>>`-Operatoren implementiert wurde. Um die zusätzlichen Funktionen des *OutputDevice*s nutzbar zu machen, werden neben den bekannten `bin`, `oct`, `dec`, `hex` und `endl` zusätzlich folgende Manipulatoren bereitgestellt:

`setpos(x, y)`	Verschiebt den Cursor nach (x,y)
`attrib(a)`	Setzt Attribute
`clear`	Löscht den Bildschirm

4.1.2 Ethernet

Die meisten PCs werden heutzutage per *Ethernet* vernetzt. Allein deswegen ergibt es bereits einen Sinn, eine allgemeine Schnittstelle zum Empfangen und Versenden von *IEEE 802.3 Ethernet Frames* zu entwerfen.

Standardfunktionen:

`getAddress()`	gibt die MAC-Adresse zurück
`setAddress(addr)`	setzt die MAC-Adresse
`send(frame, size)`	sendet Ethernet-Frame
`hasBeenSent(frame)`	`true`, falls Frame erfolgreich gesendet
`receive(frame, size)`	empfängt Ethernet-Frame
`nextData()`	gibt aktuell empfangenen Ethernet-Frame zurück
`nextFree()`	gibt empfangenes Ethernet-Frame frei

Offloading-Schnittstelle:

`send(pkt, size, tcp, ip_hdl)`	Offloading-Sendefunktion (s. u.)
`nextDataOk()`	`true`, falls Prüfsummen erfolgreich geprüft wurden

Die Standardfunktionen implementieren die grundlegenden Sende- und Empfangsfunktionen eines Netzwerkadapters. Die *Offloading*-Funktionen dienen dazu, einen Teil der Prüfsummen eines *TCP/IP*-Stacks von der Netzwerkhardware berechnen zu lassen, falls diese solche Operationen unterstützt.

Implementierung

Die *Ethernet*-Schnittstelle wird beispielsweise von dem Treiber der *Intel PRO/1000* implementiert, der in Abbildung 4.2 dargestellt wird. Die `E1000`-Klasse implementiert zunächst nur Initialisierungsroutinen und stellt selbst keine Kommunikationsfunktionen bereit. Die Sende- und Empfangsfunktionen können jeweils unabhängig voneinander per Slice in den Treiber hinein konfiguriert werden. So stellt der `E1000_Transmitter`-Aspekt alle Sendefunktionen und der `E1000_Receiver`-Aspekt alle Empfangsfunktionen bereit. Obwohl jeder Netzwerkadapter diese grundlegenden Operationen unterstützt, kann es unter Umständen sinnvoll sein, auf nicht benötigte Sende- und Empfangseinheiten zu verzichten, da sie jeweils verhältnismäßig große Deskriptorringe und Puffer verwenden.

Die optionalen *Offloading*-Aspekte beeinflussen die *Offloading*-Funktionen, die standardmäßig keine Prüfsummen berechnen und diese Aufgabe dem Benutzer überlassen, und aktivieren die hardwareunterstützte Berechnung der *TCP/IP*-Prüfsummen. Der `E1000_Transmitter_Offloading`-Aspekt modifiziert die `send()`-Funktion so, dass sie die *TCP/IP Context Deskriptoren*, die in Kapitel 2.2.6.1 bereits beschrieben wurden, nutzt, um das Offloading umzusetzen. Im Falle der Empfangsfunktionen reicht es aus, wenn der `E1000_Receiver_Offloading`-Aspekt die `init()`-Funktionen anpasst, dass

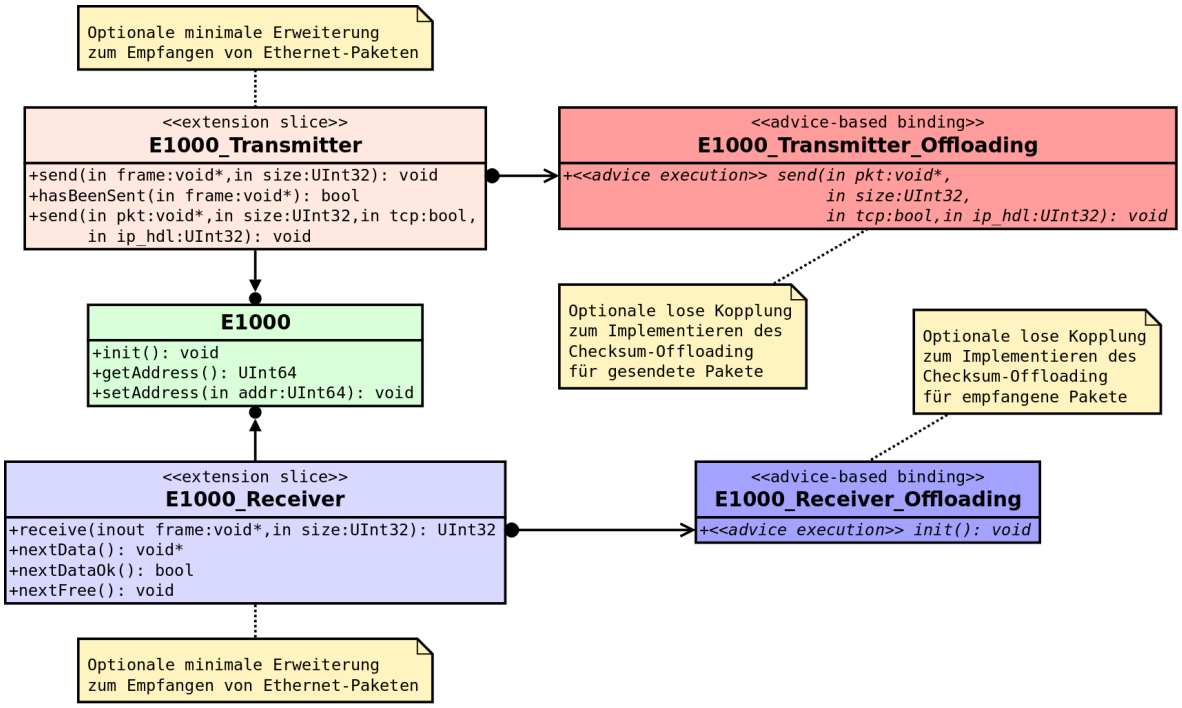

Abbildung 4.2: PRO/1000 Treiber

sie bei der Initialisierung des Adapters die automatische Überprüfung der Prüfsummen aktiviert.

4.1.3 Blockdevice

Unter einem *Blockdevice* lassen sich alle Speichermedien zusammenfassen, die in einer konstanten Anzahl Datenblöcke gleicher Größe organisiert sind. Dazu zählen insbesondere Festplatten und Flashspeicher, aber auch CD-ROMs und ähnliche Speichermedien.

`getDeviceInfo()`	Gibt Informationen zum Blockdevice zurück
`readblock(data, address)`	Liest einen Sektor ein
`writeblock(data, address)`	Schreibt einen Sektor
`readblockDMA(data, address)`	Liest einen Sektor im DMA-Modus ein
`writeblockDMA(data, address)`	Schreibt einen Sektor im DMA-Modus

Die einfachen `read`- und `writeblock`-Funktionen stehen immer zur Verfügung. Dabei handelt es sich um *blockierende* Funktionen, die erst zurückkehren, wenn die gewünschte Operation abgeschlossen ist. Die optionalen *DMA*-Funktionen lassen sich bei Bedarf nach dem Prinzip der *minimalen Erweiterung* hinzufügen und laufen grundsätzlich im Hintergrund ab. Diese Funktionen kehren sofort zurück und melden den Abschluss eines Vorgangs per Unterbrechung.

Implementierung

Eine beispielhafte Implementierung eines *Blockdevice*s ist mit dem *IDE*-Treiber gegeben.

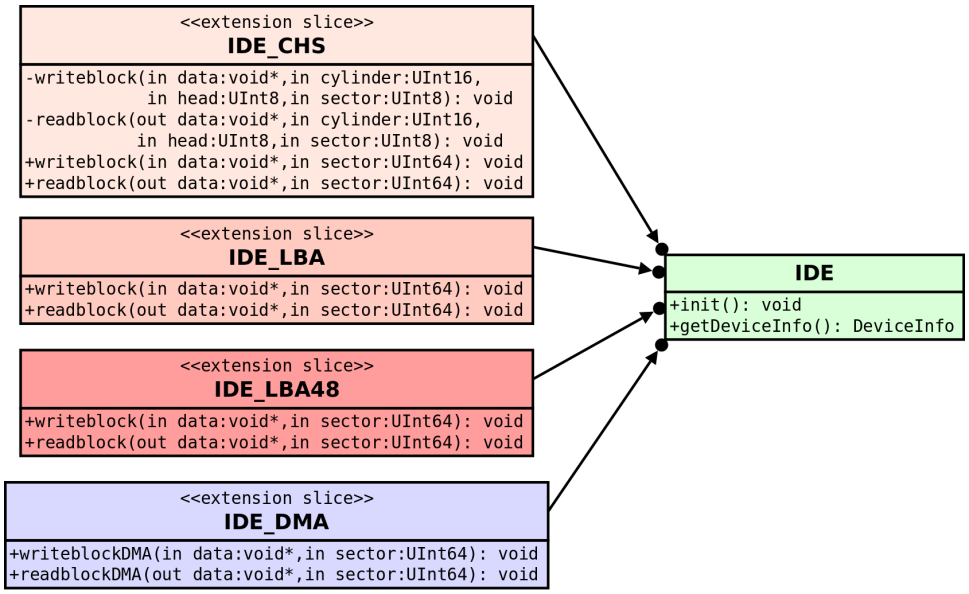

Abbildung 4.3: IDE-Treiber

Analog zu dem *PRO/1000*-Treiber, stellt die `IDE`-Basisklasse nur Initialisierungsroutinen sowie Funktionen zur Statusabfrage zur Verfügung. Mit Hilfe von *pure::variants* muss man genau einen der Aspekte `IDE_CHS`, `IDE_LBA` und `IDE_LBA48` auswählen, die jeweils die Lese- und Schreiboperationen der unterschiedlichen Adressierungsarten, die in Kapitel 2.2.6.2 erläutert wurden, per *Klassen-Slice* hinzufügen. Unabhängig davon kann der `IDE_DMA`-Aspekt ausgewählt werden, um die oben beschriebenen *DMA*-Funktionen bereit zu stellen. Diese ersetzen aufgrund der unterschiedlichen Semantik nicht die *PIO*-Funktionen, sondern werden ebenfalls per *Klassen-Slice* hinzugefügt.

4.1.4 Paging

Die Schnittstelle der *seitenbasierten Speicherverwaltung*, auch *Paging* genannt, ist zwar keine typische Treiber-Schnittstelle wie *Ethernet* oder *Blockdevice*, aber wegen ihrer zentralen Rolle im Speichermanagement spricht einiges dafür, eine Hardwareabstraktion dafür zu entwerfen. Eine allgemeine Speicherseitenverwaltung führt als wichtigstes Merkmal eine Abbildung von virtuellen Speicheradressen auf physikalische Adressen durch. Vor diesem Hintergrund sieht eine einfache Schnittstelle, die von sämtlichen Seitentabellenformaten abstrahiert, folgendermaßen aus:

`enable()`	Aktiviert das Paging
`disable()`	Deaktiviert das Paging
`setPage(vaddr, paddr, pagesize, write, user, exe)`	Legt eine Seite an (s. u.)
`clearPage(vaddr, pagesize)`	Entfernt eine Seite

Neben dem einfachen Anlegen einer Speicherseite, die eine Abbildung von `vaddr` auf `paddr` vornimmt, kann man zusätzlich weitere Attribute angeben. So lassen sich komplette Speicherseiten schreibschützen, vor Zugriffen unprivilegierter Anwendungen schützen oder vor ungewollter Ausführung schützen.

Implementierung

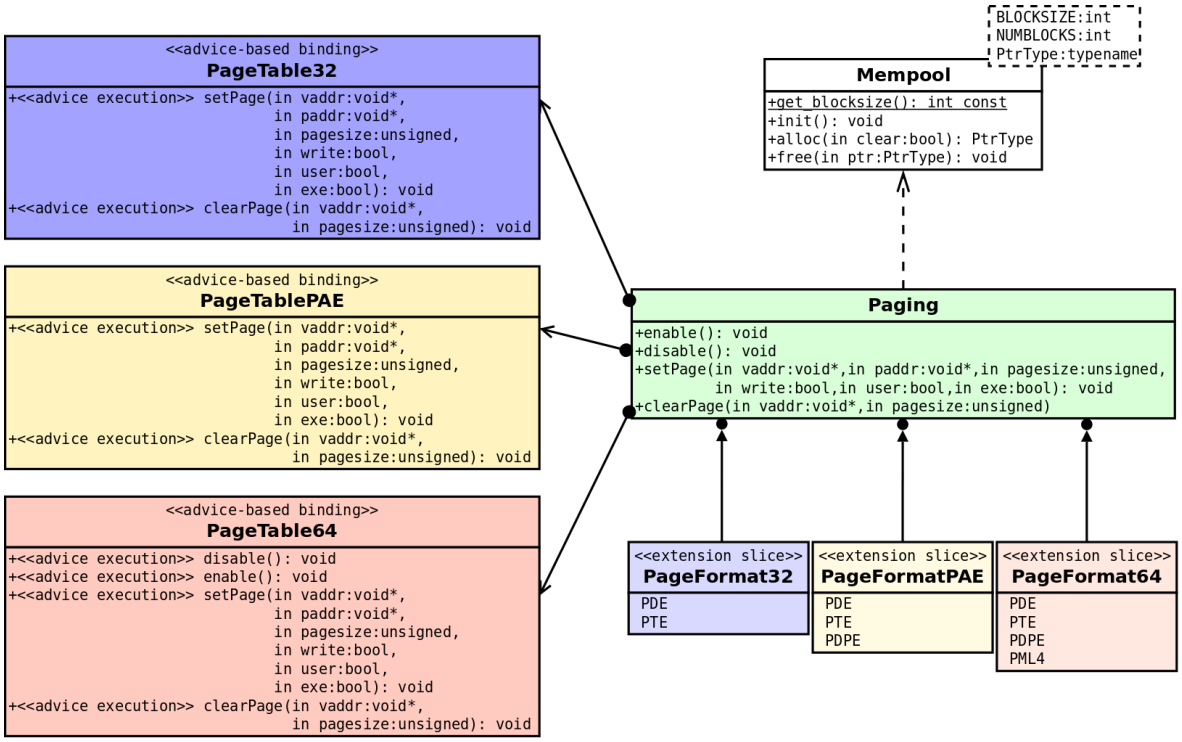

Abbildung 4.4: Paging

Eine größere Hürde, die bei der Umsetzung der Paging-Schnittstelle auf der *x86*-Architektur zu nehmen ist, besteht in der Vielfalt der unterschiedlichen Seitentabellenformate und Seitengrößen, die sich im Laufe der Zeit entwickelt haben. So kann die Seitentabelle einstufig sein, falls man sich im 32-Bit-Adressraum mit 4-MB-Seiten zufrieden gibt. Im anderen Extremfall besteht die Seitentabelle aus ganzen vier Stufen, wenn 4-KB-Seiten im 48-Bit-Adressraum benötigt werden. Da sich die drei Grundformate *32-Bit*, *PAE* und *64-Bit* teils grundlegend voneinander unterscheiden, bietet es sich an, die dazu benötigten Strukturen als *minimale Erweiterung* mit Hilfe der Aspekte `PageFormat32`, `PageFormatPAE` und `PageFormat64` einzubinden. Die Algorithmen zum

Aufbau der Tabellenhierarchie lassen sich ergänzend dazu als *lose Kopplung* der Aspekte `PageTable32`, `PageTablePAE` und `PageTable64` an die Schnittstellenfunktionen realisieren, wobei *pure::variants* sicherstellt, dass nur passende Formate und Algorithmen zusammen ausgewählt werden können.

Da sich die Seiten zur Laufzeit einfügen und entfernen lassen, steht man noch vor einem weiteren Problem. Mit zunehmender Seitenzahl muss die Tabelle wachsen, um neue Seiten aufnehmen zu können. Dies setzt das Vorhandensein einer dynamischen Freispeicherverwaltung voraus, die in *CiAO* so nicht vorgesehen ist. Die triviale Möglichkeit des „Vorreservierens“ der gesamten Tabelle ist insbesondere bei den komplexeren Tabellen keine Lösung, wie man leicht an der Rechnung im Kapitel 2.2.2 erkennen kann. Da jede Tabelle eine konstante Größe von 4096 Bytes besitzt und nahezu frei im Speicher platziert werden kann, bietet sich ein *Speicherpool* an. Dabei wird ein größerer zusammenhängender Speicherblock in 4-KB-Blöcke geteilt. Die einzelnen Blöcke werden in einer Liste eingetragen und bei Speicheranforderungen ausgegeben und aus der Liste entfernt. Falls Seiten wieder komplett entfernt werden können, trägt man sie wieder in die Liste ein, damit sie erneut verwendet werden können:

`Mempool<SIZE, N>`	*Konstruktor*: erzeugt Mempool mit N Blöcken mit Blockgröße SIZE
`alloc()`	gibt einen leeren Speicherblock zurück
`free(ptr)`	gibt einen Speicherblock frei

4.2 Grundlegende Anpassungen

Um die Hardwareabstaktionsschicht der *CiAO*-Betriebssystemfamilie an eine neue Architektur anzupassen, trifft man auf das Problem, das Betriebssystem zunächst überhaupt auf der neuen Plattform starten zu können. Dazu sind mehrere Schritte notwendig, bevor das System in der Lage ist, selbstständig hochzufahren. Anschließend geht es darum, die bereits vorhandenen grundlegenden Schnittstellen für die neue Architektur zu implementieren, sofern die dazu notwendige Hardware vorhanden ist. Im Folgenden werden diese Schnittstellen vorgestellt. Insbesondere wird dabei auf die Besonderheiten der *x86*-Architektur eingegangen, falls diese für die Implementierung relevant sind.

4.2.1 Startupcode

Als *Startupcode* bezeichnet man die Codeabschnitte eines Betriebssystems, die Systeminitialisierungen vornehmen, die so grundlegend sind, dass sie noch vor Ausführung des betriebssystemeigenen Codes vorgenommen werden müssen. Dies betrifft beim Starten eines Betriebssystems der *x86*-Architektur vor allem folgende Punkte:

1. Laden bzw. Initialisieren der `.text`-, `.data`- und `.bss`-Sektionen
2. Anlegen der *Globalen Deskriptor Tabelle* und Initialisierung der Segmentregister
3. Anlegen einer minimalen Seitentabelle (s. Kapitel 4.4)

4. Setzen des Stapelzeigers
5. Einstellen des benötigten Betriebsmodus (s. Kapitel 4.4)
6. Verarbeiten evtl. vorhandener Kommandozeilenparameter des Bootloaders
7. Aufrufen des Einsprungpunktes des Betriebssystem

Einige der erwähnten Punkte können zum Teil bereits vom Bootloader abgearbeitet worden sein. So befindet man sich nach Ausführung des *GRUB* beispielsweise bereits im *32-Bit Protected-Mode*, so dass ein erneutes Umschalten des Betriebsmodus für ein 32-Bit Betriebssystem entfallen kann.

4.2.2 Linkerskript

Genau wie jedes Programm wird auch ein Betriebssystemkern aus mehreren Modulen von einem *Linker* gebunden. Ein *Linkerskript* definiert dabei u. a. die Positionen und das Alignment der einzelnen Sektionen im Kernelimage und auch im Speicher. So lassen sich Komponenten des Systems gezielt im Speicher anordnen, um einen evtl. vorhandenen Speicherschutzmechanismus zu vereinfachen oder überhaupt erst zu ermöglichen (s. Kapitel 4.5). Beim Schreiben dieses Skripts sind zusätzlich die jeweiligen Anforderungen des Systems zu berücksichtigen. So fordert beispielsweise *GRUB*, dass am Anfang des Kernelimages ein Header stehen muss, damit ein Image automatisch als Betriebssystem anerkannt wird.

4.2.3 Unterbrechungsbehandlung

Die *x86*-Variante von *CiAO* verwendet den in Kapitel 2.2.3 beschriebenen *Programmable Interrupt Controller*, um auf Unterbrechungsanforderungen reagieren zu können. Dieser wird so programmiert, dass sämtliche zugelassenen Unterbrechungen eine `guardian()`-Funktion mit der Vektornummer der Unterbrechung als Parameter aufrufen. Die `guardian()`-Funktion ermittelt anhand einer Handler-Tabelle die zuständige konkrete Behandlungsroutine und ruft diese auf, sofern sie vorhanden ist. Ansonsten wird eine `irq_handler_not_found()`-Funktion als möglicher *sichtbarer Übergang* aufgerufen, um eine Fehlerbehandlung zu ermöglichen. Diese zusätzliche Indirektionsstufe scheint hier zunächst unnötig zu sein, vereinfacht später aber die Implementierung der Unterbrechungsbehandlung des Gastsystems, dessen Entwurf in Kapitel 4.3.2 vorgestellt wird.

Aufgrund der *AUTOSAR-OS* kompatiblen Schnittstelle implementiert *CiAO* die Unterbrechungsbehandlung als sogenannte *Interrupt Service Routines* (kurz: *ISRs*). Der allgemeine Ablauf der Behandlung sieht dabei folgendermaßen aus [19, Kapitel 2.1.2]:

1. Unterbrechung wird ausgelöst
2. Der vom Gerätetreiber bereitgestellte Interrupt-Handler wird aufgerufen

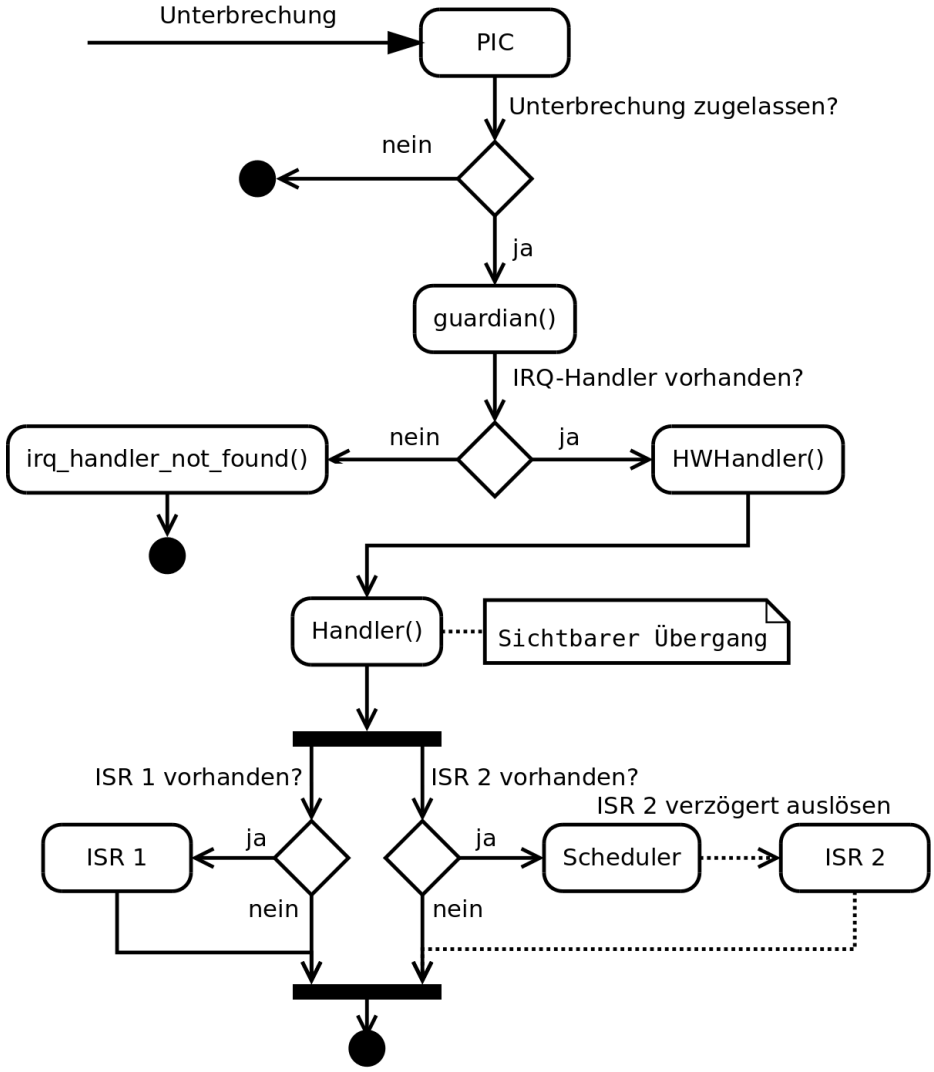

Abbildung 4.5: Unterbrechungsbehandlung

3. *ISR* der Kategorie 1 wird direkt ohne weitere Synchronisation aufgerufen
4. *ISR* der Kategorie 2 wird ausgelöst und bei der nächstmöglichen Gelegenheit ausgeführt.

Bei dem *ISR 1* handelt es sich mehr oder weniger um einen „normalen“ *Prologue*, der sofort ausgeführt wird. Der *ISR 2* erfüllt dementsprechend die Aufgabe des *Epilogue*s, dessen Ausführung unter Umständen auch verzögert werden kann.

4.2.4 AST-Emulation

Einige Architekturen, wie beispielsweise die *TriCore*-Architektur, stellen direkt einen *Asynchronous System Trap* zur Verfügung. Dabei handelt es sich um eine Art niederpriore Unterbrechung, die ihrerseits von „normalen“ Unterbrechungen unterbrochen werden

kann. Damit kann sichergestellt werden, dass länger dauernde Routinen nicht automatisch zu einer signifikant höheren Unterbrechungslatenz führt.

Da die *x86*-Architektur das Konzept des *AST*s nicht kennt, muss diese in Software nachgebildet werden, um den Dispatcher in derselben Art und Weise ausführen zu können.

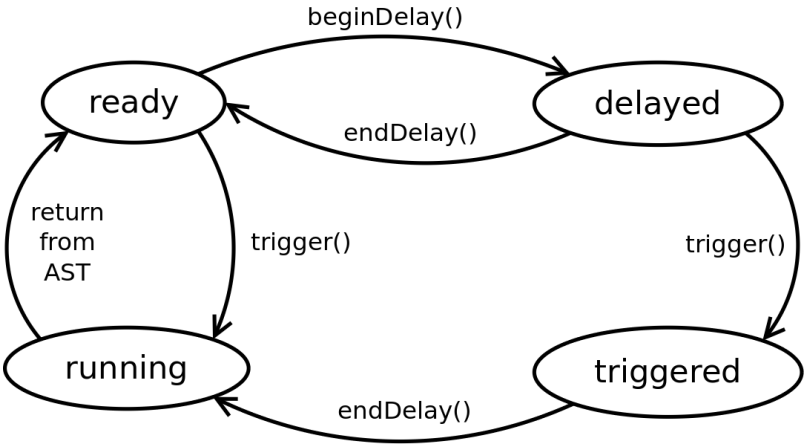

Abbildung 4.6: AST-Emulation

In Abbildung 4.6 wird das Zustandsdiagramm der Softwareimplementation des *AST*s dargestellt. Solange der *AST* freigegeben ist (*ready*), wird er bei Auslösung (*trigger()*) direkt ausgeführt. Falls der *AST* zum Zeitpunkt der Auslösung gesperrt war (*delayed*), wird die Ausführung bis zur Freigabe (*endDelay()*) verzögert. Während ein *AST* ausgeführt wird, werden normale Unterbrechungen zugelassen, so dass diese den *AST* jederzeit unterbrechen können, um ggf. weitere *AST*-Ausführungen anzufordern, die alle zu bearbeiten sind, bevor mit der normalen Taskausführung fortgefahren wird.

Um das Unterbrechen von Systemfunktionen innerhalb des Systemkerns vom *AST* zu verhindern, bindet sich ein Aspekt an die `enterKernel()`- und `leaveKernel()`-Funktionen, die die entsprechenden `delay()`-Funktionen aufrufen (*Lose Kopplung*).

Listing 4.1: AST-Delay Aspekt

```
1  advice execution("% os::krn::enterKernel(...)") : before() {
2    hw::hal::ASTImpl_CPU_SRC0::beginDelay();
3  }
4
5  advice execution("% os::krn::leaveKernel(...)") : after() {
6    hw::hal::ASTImpl_CPU_SRC0::endDelay();
7  }
```

4.2.5 klassischer Kontextwechsel

In einem Multitaskingsystem wie *CiAO* kommt es regelmäßig vor, dass Tasks auf externe Ereignisse warten und vorerst nicht weiter ausgeführt werden können. In diesem

Fall wird auf einen anderen Task umgeschaltet, sofern dieser ausführbereit ist. Um den blockierenden Task später problemlos weiter ausführen zu können, ist es unumgänglich, dessen prozessorinternen Zustand vollständig im Speicher zu sichern. Gerade beim Einsatz einer Hochsprache wie C++ kann man sich dazu die jeweilige *Aufrufkonvention* der verwendeten Plattform zunutze machen. Diese definiert unter anderem, welche Register *flüchtig* sind und demnach bereits von der aufrufenden Funktion auf dem Stack zu sichern sind. Da die Kontextwechselroutinen auch nur als Funktionen implementiert sind, braucht man sich keine weiteren Gedanken um diese Register machen. Es reicht also vollkommen aus, die übrigen *nicht-flüchtigen* Register zu sichern. Dazu gehören im 32-Bit-Modus die Register `ebx`, `esi`, `edi`, `ebp` und `esp`. Da im 64-Bit-Modus eine andere Aufrufkonvention vereinbart wurde, müssen dort die Register `rbx`, `r12`, `r13`, `r14`, `r15`, `rbp` und `rsp` gesichert werden. Diese Register werden bei der *x86*-Variante von *CiAO* in einer Kontextvariablen innerhalb der Task-Struktur gesichert, um sie später wiederherstellen zu können.

4.2.6 Alarm

Mit Hilfe eines *Alarm*s kann sich ein Task für eine bestimme Zeit selbst blockieren oder ein Event auslösen. Der Scheduler weckt den so schlafenden Task erst dann wieder, wenn die Zeit des *Alarm*s abgelaufen ist. Die *TriCore*-Variante von *CiAO* nutzte dazu das *SystemTimerModule* (kurz: *STM*) als Zeitbasis.

Die *x86*-Variante von *CiAO* verwendet den *PIT* als Systemtimer, der in Kapitel 2.2.5.1 beschrieben wurde. Prinzipiell lässt sich aber jeder andere dort erwähnte Timerbaustein nutzen. Um dem Kernel weiterhin eine kompatible *HAL* anbieten zu können, muss die *STM*-Schnittstelle des *TriCore*s in Software nachgebildet werden. Um dies zu erreichen wird der *PIT* auf ein festes Unterbrechungsintervall eingestellt, um damit einen Softwarezähler zu implementieren. Ein Aspekt bindet sich dazu an die entsprechende Unterbrechungsbehandlungsroutine, um den Zähler heraufzuzählen und um den aktuellen Zählerwert mit einem Komparatorwert zu vergleichen. Bei Übereinstimmung wird die rein virtuelle *STM*-Unterbrechung ausgelöst.

4.2.7 einfache Treiber

Neben den grundlegenden Schnittstellen, die zur Umsetzung der klassischen Aufgaben eines Betriebssystems benötigt werden, ist es oft hilfreich, einfache Ein- und Ausgabetreiber zur Verfügung zu stellen, um insbesondere bei der Entwicklung Debugmeldungen ausgeben zu können.

COM-Port

Die in Kapitel 2.2.5.2 beschriebenen *COM-Ports* eines PCs entsprechen genau den *ASCs* des *TriCore*s im asynchronen Modus [20, Kapitel 3.9]. Daher lässt sich die in *CiAO* bereits vorhandene *Serial*-Schnittstelle stringent umsetzen.

Um größtmögliche Konfigurierbarkeit bieten zu können, wurde die gesamte *COM*-Klasse als Klassentemplate realisiert, die eine *I/O*-Basisadresse als Parameter benötigt. Der Wert der Adresse lässt sich im Variantenmodell von *pure::variants* einstellen. So lassen sich relativ leicht nahezu beliebig viele *COM*-Objekte anlegen, was gerade bei *RS-232*-Schnittstellen nützlich ist, da diese zu den Komponenten eines PCs gehören, die häufig mehrfach vorhanden sind.

CGA

Der Treiber des *Color Graphics Adapter*s implementiert den Zugriff auf die in Kapitel 2.2.5.3 beschriebene Schnittstelle, die von allen aktuellen PC-Grafikkarten angeboten wird. Als Basis für den *DebugPort* und das *OutputDevice* stellt der Treiber Methoden zur einfachen Ausgabe von Zeichenketten zur Verfügung. Darüber hinaus werden einfache Steuerungsfunktionen des Cursors sowie Attribute zur Änderung der Text- und Hintergrundfarbe bereitgestellt.

Keyboard

Die Treiberklasse des in Kapitel 2.2.5.4 beschriebenen *AT-Keyboard*s dient hauptsächlich als Basis für das *InputDevice*. Da das *InputDevice* ausschließlich Zeichen verarbeitet, muss der Keyboard-Treiber die von der Tastatur empfangenen *Scancodes* dekodieren, wobei man hierzu das jeweils verwendete Tastaturlayout berücksichtigen muss. Die für

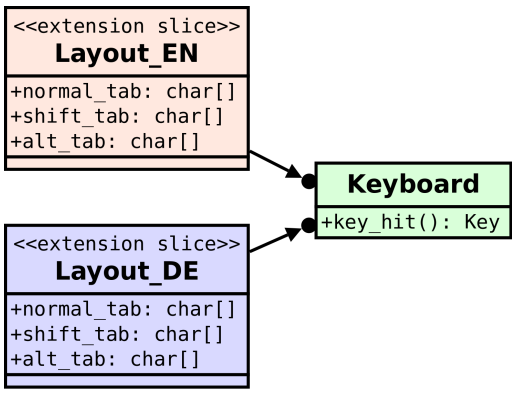

Abbildung 4.7: konfigurierbares Tastaturlayout

das Dekodieren notwendigen Scancodetabellen können per *minimale Erweiterung* an das jeweilige Layout angepasst werden.

4.3 Linux Gast Implementierung

4.3.1 Startupcode

Da die grundlegende Initialisierung der Laufzeitumgebung und das Laden der Applikationssektionen in den Speicher grundsätzlich vom Hostsystem übernommen wird, fallen viele der in Kapitel 4.2.1 genannten Punkte weg. Übrig bleiben im Wesentlichen nur folgende drei Punkte:

1. Verarbeitung der Kommandozeilenparameter, die der Startfunktion auf dem Stack übergeben werden
2. Initialisierung genutzter Systembibliotheken, falls vorhanden
3. Aufruf des Einsprungspunktes des Gastsystems

Da sich abgesehen vom letzten Punkt der gesamte Startupcode unterscheidet, wird bei der Konfigurierung eines Linux-Gast-Systems gleich der gesamte Startupcode von *pure::variants* ersetzt.

4.3.2 Unterbrechungsbehandlung mit Signalen

Linux bzw. allgemein *POSIX*-konforme Systeme bieten Applikationen ein Signalisierungssystem an, dass große Ähnlichkeiten zu den „echten“ Unterbrechungen ausweist. Daher bietet es sich an, die *POSIX*-Signale auf die *CiAO*-Unterbrechungsbehandlung abzubilden. Dazu reicht es beim Zulassen einer Unterbrechung aus, die `guardian()`-Funktion, die im nativen System beim *PIC* registriert wurde, als *POSIX*-Signalhandler einzutragen. Beim Eintreffen eines Signals wird diese Funktion mit der entsprechenden Signalnummer als Parameter aufgerufen. Die gesamte restliche Infrastruktur zur Unterbrechungsbehandlung kann unverändert vom nativen System übernommen werden.

4.3.3 Linux Syscall-Wrapper

Als Gastsystem läuft *CiAO* als normale Applikation. Als solche hat sie vollen Zugriff auf sämtliche Bibliotheksfunktionen des Hostsystems. Dementsprechend könnten alle Tasks des Gastsystems ebenfalls auf diese Funktionen zugreifen. So ein Vorgehen würde zwar die Entwicklung von *CiAO*-Tasks erheblich vereinfachen, stünden doch sämtliche *POSIX*-Funktionen auf einmal zur Verfügung. Aber dies würde einen massiven Bruch des Schichtenmodells bedeuten, da aus der Sicht des *CiAO*-Kerns eine Applikation direkt auf die „Hardware“ zugreifen würde. In diesem Fall umgeht man sämtliche Strukturen des Gastsystems, so dass man nicht mehr weit davon entfernt ist, eine normale Linux Applikation zu schreiben. Das Gastsystem wäre damit quasi überflüssig.

Abhilfe schafft wie bereits in Kapitel 3.1.2.0.11 beschrieben ein *Syscall-Wrapper*, der als Schnittstelle zwischen Gast- und Hostsystem dient. Der Wrapper selbst steht somit auf der Ebene der Gerätetreiber und lässt einen kontrollierten Zugriff auf Funktionen

des Hostsystems zu. Entsprechende *HAL*-Funktionen implementieren damit soweit möglich dieselben Schnittstellen wie das native System, so dass *CiAO*-Applikationen ohne Änderungen auch unter einem Gastsystem ausgeführt werden können.

4.3.4 grundlegende Treiber

Im Folgenden werden einfache beispielhafte Treiber für das Gastsystem von *CiAO* vorgestellt.

4.3.4.1 Input- und OutputDevice

Jeder Anwendung unter einem *POSIX*-Betriebssystem stehen grundsätzlich jeweils ein *Standardeingabe*-, ein *Standardausgabe*- und ein *Standardfehler*-Stream zur Verfügung. Um diese auch in den Applikationen des *CiAO*-Gastsystems nutzen zu können, bietet es sich an, die *InputDevice*- und *OutputDevice*-Schnittstellen auf die Standardeingabe bzw. Standardausgabe abzubilden. Die im *OutputDevice* definierten Attribute können dazu genutzt werden, um evtl. vorhandene Sonderfunktionen des Terminals wie Änderung der Vordergrund- und Hintergrundfarbe anzusteuern. Der konventionelle *CiAO-DebugPort* wird auf die Standardfehlerausgabe gelenkt, um Debugmeldungen von „normalen“ Ausgaben unterscheiden zu können. Somit stehen Tasks des Gastsystems dieselben Ein- und Ausgabeschnittstellen wie die des nativen Systems zur Verfügung. Auch die gepufferten *Stream*-Klassen können normal genutzt werden.

Standardmäßig werden die vom Hostsystem angebotenen Streams zeilenweise gepuffert, was dazu führt, dass ein Gastsystem eine Benutzereingabe erst dann verarbeiten kann, wenn der Benutzer diese mit einem Zeilenumbruch (bzw. *ENTER*-Taste) abschließt. Um dennoch eine zeichenweise Eingabe realisieren zu können, wird die Standardeingabe in den sogenannten *RAW*-Modus versetzt, der sämtliche Eingaben direkt an das Gastsystem weiterleitet. Dieser muss beim Beenden des Gastsystems wieder aufgehoben werden, damit nachfolgende Anwendungen nicht beeinträchtigt werden.

4.3.4.2 POSIX Interval Timer

Ähnlich zu den Streams stehen jedem *POSIX*-Prozess mehrere Timer zur Verfügung, die in regelmäßigen zeitlichen Abständen ein Signal auslösen. Dazu wird ein Zähler auf einen frei wählbaren Wert gesetzt und heruntergezählt. Wenn dieser Zähler den Wert *0* erreicht, wird das Signal ausgelöst und der Zähler, falls gewünscht, wieder zurückgesetzt.

POSIX definiert mindestens drei Timer [21]:

1. Der **Realtime-Timer** zählt in Echtzeit herunter und läuft auch weiter, wenn *CiAO* vom Hostsystem unterbrochen wurde, um beispielsweise andere Prozesse des Hostsystems ausführen zu können.

2. Der **Virtual-Timer** zählt nur dann herunter, wenn *CiAO* aktiv ist und damit *Prozessorzeit* beansprucht.

3. Der **Profiling-Timer** zählt sowohl die *Prozessorzeit* von *CiAO* als auch die *Systemzeit* des Hostsystems, wenn dieses Syscalls vom Gastsystem bearbeitet.

Insbesondere der Echtzeit-Timer eignet sich zusammen mit der in Kapitel 4.3.2 beschriebenen Unterbrechungsbehandlung, um einen typischen Systemtimer für das Gastsystem zu implementieren. Mit dessen Hilfe lässt sich ebenfalls die in Kapitel 4.2.6 beschriebenen *Alarme* realisieren.

4.4 64-Bit Long-Mode

Mit Einführung des 64-Bit-Modus bzw. *Long-Mode*s wurde nach dem *Real-Mode* und dem *Protected-Mode* der dritte umfangreiche Betriebsmodus des *x86* eingeführt. Wie auch schon beim *32-Bit Protected-Mode* gehen auch beim *Long-Mode* einige Implikationen einher, die bei der Implementierung zu berücksichtigen sind.

4.4.1 Umschaltung in den Long-Mode

Das Aktivieren des 64-Bit-Modus ist eine aufwändige Prozedur, die im Startupcode implementiert wird. Erschwerend kommt hinzu, dass die Umschaltung eine Reihe von Abhängigkeiten besitzt, die vorher zu erfüllen sind und damit ebenfalls im Startupcode geregelt werden müssen.

1. Anlegen von 32-Bit- und 64-Bit-Codesegmenten in der *Globalen Deskriptor Tabelle*

2. Aktivierung der *Physical Address Extension*

3. Aktivierung des Long-Modes im Kompatibilitätsmodus

4. Einrichten einer 64-Bit Seitentabelle und Aktivierung des Pagings

5. Intersegmentsprung in das 64-Bit-Codesegment

Problematisch ist dabei insbesondere die Seitentabelle, da diese auch während des normalen Betriebs manipuliert werden kann und die dazu benötigte Infrastruktur auch außerhalb des Startupcodes nutzbar sein muss. Auf der anderen Seite ist der Startupcode nicht ohne Weiteres in der Lage, diese externe Infrastruktur zu nutzen, da sie erst nach der Ausführung des Startupcodes initialisiert wird. Daher wird im 64-Bit-Startupcode von *CiAO* eine temporäre Seitentabelle erzeugt, die später vom System komplett ersetzt wird. Diese vorübergehende Seitentabelle bildet gerade ausreichend viele Seiten des virtuellen Adressraumes direkt auf den physikalischen Speicher ab, um das System starten zu können.

4.4.2 Aufrufkonvention

Mit der Einführung des 64-Bit-Modus wurde die Anzahl der Allzweckregister verdoppelt, so dass man sich darauf geeinigt hat, Parameter bei Funktionsaufrufen nicht mehr wie bisher auf dem Stack abzulegen, sondern direkt in Registern zu übergeben. Dies hat zur Folge, dass der für den Kontextwechsel zuständige Code nicht nur längere Register sichern und wiederherstellen muss, sondern teilweise auch völlig andere. Da diese hardwarenahen Prozeduren somit kaum noch Ähnlichkeiten aufweisen, werden sie analog zum Startupcode vollständig von *pure::variants* ersetzt.

4.4.3 unterschiedliche Adresslänge

Innerhalb eines Hochsprachen-Programms wird man recht wenig davon bemerken, ob ein System im 32-Bit- oder im 64-Bit-Modus läuft. Reine 64-Bit Arithmetik wird vom Compiler abhängig vom jeweiligen Modus entweder direkt ausgeführt (64-Bit-Modus) oder in mehrere kleinere Operationen aufgeteilt (32-Bit-Modus). Ein reiner Anwendungsentwickler bemerkt davon nichts. Auch innerhalb von *CiAO* funktionieren die meisten Komponenten ohne Änderung des Quellcodes im 32-Bit- und im 64-Bit-Modus. Sie sind *quelltext-kompatibel*. Die einzigen wirklich problematischen Fälle sind Komponenten, die direkt Speicheradressen manipulieren müssen, da sich die Länge der Adressen unterscheidet. Dies betrifft insbesondere die Seitentabelle und die Speicherschutzkomponenten aber auch Treiber für Geräte mit *memory-mapped* Registern. Hier ist es angebracht, statt Datentypen fester Länge wie `UInt32` oder `UInt64` solche mit variabler Länge zu verwenden. Es bietet sich daher in *CiAO* an, `UIntPtr` zu benutzen.

4.4.4 Wegfall des TSS-Taskwechsels

Neben der segmentierten Speicherverwaltung fiel der 64-Bit Erweiterung auch der hardwaregestützte Taskwechsel über das *Task-State-Segment* zum Opfer. Durch die längeren 64-Bit-Register und den 8 zusätzlichen Allzweckregister würde das *TSS* von 104 Bytes auf 272 Bytes anwachsen. Vergleicht man diese Zahl mit den 56 Bytes, die ein manuell gesicherter Taskkontext explizit benötigt, wird schnell deutlich, dass ein blindes Sichern aller Register unnötig viel Zeit beanspruchen würde. Selbst unter Berücksichtigung der implizit gesicherten Register, die von der Aufrufkonvention für Funktionen gefordert wird und daher ohnehin vom Compiler für die Kontextwechselfunktionen auf dem Stack gesichert werden, müssen bei einem manuell gesicherten Taskkontext insgesamt nur 128 Bytes gespeichert und geladen werden. Daher ist der Wegfall des TSS-Taskwechsel kein wirkliches Problem für die 64-Bit-Implementierung von *CiAO*.

4.5 Seitenbasierter Speicherschutz

Mit Hilfe der Seitentabelle, die in Kapitel 4.1.4 beschrieben wurde, lässt sich ein effektiven Speicherschutzmechanismus implementieren. Die *TriCore*-Variante nutzte dazu

eine *Memory Protection Unit* (kurz: *MPU*), mit der globale Zugriffsbeschränkungen auf Speicherbereiche festgelegt werden können [20, Kapitel 3.4]. Bei aktiviertem Speicherschutz setzt der Kernel für jeden Applikationswechsel die entsprechenden Rechte neu, um so individuelle Speicherberechtigungen zu implementieren

Die auf der modernen *x86*-Architektur vorhandene *Memory Management Unit* (kurz: *MMU*), die für das in Kapitel 4.1.4 beschriebene Paging zuständig ist, geht etwas weiter und speichert keine globalen Zugriffsbeschränkungen, die regelmäßig zu ändern sind. Stattdessen werden die jeweiligen Rechte innerhalb der Seitentabellen festgelegt, die in Kapitel 2.2.2 bereits erläutert wurden. Ein Austausch der Seitentabelle bei einem Kontextwechsel reicht also aus, um die Zugriffsregeln auf die jeweiligen Bedürfnisse anzupassen. Dabei gibt es grundsätzlich drei Möglichkeiten:

1. Eine bzw. zwei **globale Seitentabellen** beschränken den Zugriff der unprivilegierten Applikationen auf die Strukturen des Kernels.

2. Eine **Seitentabelle pro Applikation** bietet zusätzlich noch Schutz der applikationseigenen Daten.

3. Eine **Seitentabelle pro Task** schützt innerhalb einer Applikation zusätzlich noch die privaten Daten der einzelnen Tasks.

Für die in dieser Arbeit entwickelten *CiAO*-Variante, wurde die dritte Möglichkeit mit jeweils einer *Seitentabelle pro Task* beispielhaft implementiert. Dabei stellte sich heraus, dass sich die *MPU*-Schnittstelle nur schlecht für die Implementierung dieses feingranularen Speicherschutzes über die *MMU* eignet, da die *MPU*-Funktionen keine Möglichkeit haben, herauszufinden, für welche Applikation, bzw. für welchen Task die jeweiligen Schutzregeln zu aktivieren sind. Dies ist bei der *MMU* aber zwingend notwendig, da die nur hier vorhandene Seitentabelle zum Applikationskontext bzw. Taskkontext gehört.

Wie bei der Analyse im Kapitel 3.2.3.1 bereits festgestellt wurde, handelt es sich bei dem *seitenbasierten Speicherschutz* um einen *querschneidenden Belang*. Zum einen muss der Schutz sämtliche Speicherzugriffe eines Tasks überwachen, zum anderen benötigen die Funktionen der *HAL* und des Kernels ständig vollen Zugriff. Aus diesem Grund wird das Konzept der Trennung in einen *supervisor-mode* und in einen *user-mode* eingeführt. Alle Applikations-Tasks werden im *user-mode* ausgeführt, während der *supervisor-mode* dem Kernel und den Gerätetreibern vorbehalten ist. Zur Umschaltung verwendet die *x86*-Variante von *CiAO* die *sichtbaren Übergänge* `enterSystem()` und `leaveSystem()`, die per Aspekt an sämtlichen von einer Applikation aus aufgerufenen Systemfunktionen sowie alle Unterbrechungen gebunden werden. Der plattformunabhängige Teil von *CiAO* verwendet die ähnlichen Funktionen `enterKernel()` und `leaveKernel()`, um den Kernel selbst zu sperren (*Locking*), damit dieser nicht unterbrochen wird. Anders als bei den hier vorgestellten sichtbaren Übergängen, dürfen diese nicht an die Unterbrechungsbehandlungsroutinen gebunden werden. Bei dieser Art von Aspekten werden die Prinzipien *quantification* und *obliviousness* besonders gut sichtbar.

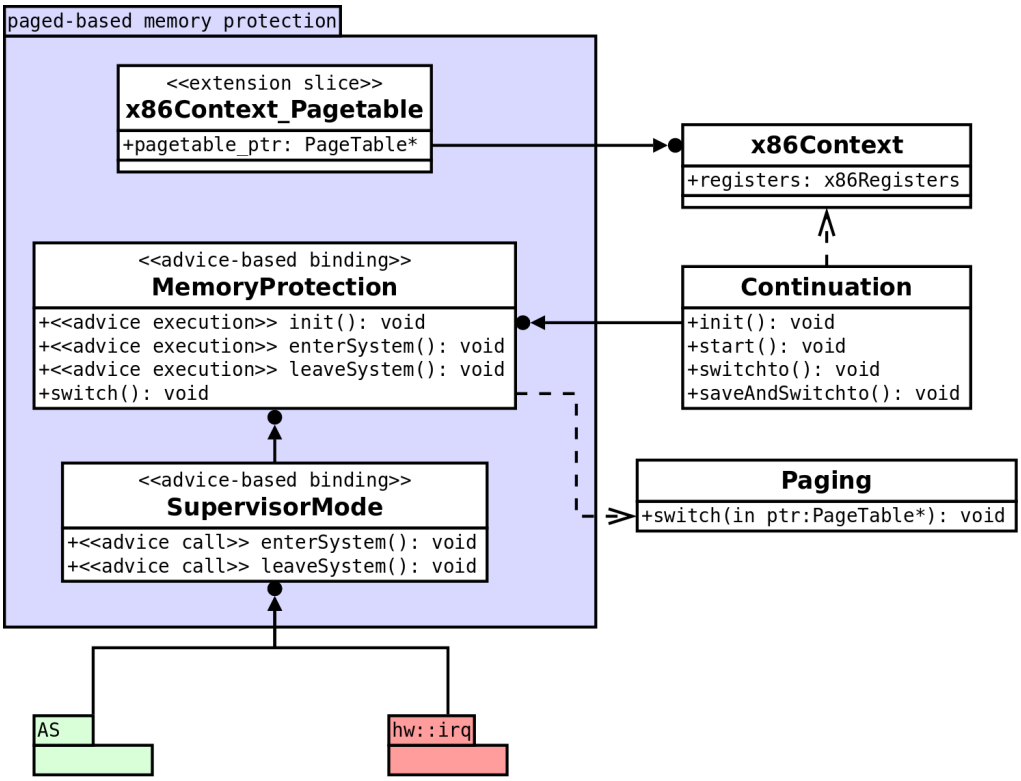

Abbildung 4.8: Seitenbasierter Speicherschutz

Listing 4.2: supervisor-mode

```
1 advice call("% AS::%(...)") && within(asTaskFunctions ())
2     || execution("% hw::irq::...::%(...)")
3       : around() {
4   hw::enterSystem();
5   tjp->proceed();
6   hw::leaveSystem();
7 }
```

Der im Listing 4.2 gezeigte Advice sorgt dafür, dass vor und nach jedem Systemaufruf sowie bei jedem Auftreten einer Unterbrechung in den jeweils passenden Modus geschaltet wird, ohne dass dazu Änderungen am Applikationscode oder am Kernelcode notwendig sind.

Ein weiterer Aspekt sorgt dafür, dass die *Enter*-Funktion in die kerneleigene Seitentabelle wechselt, in der dem Kern vollen Zugriff gewährt wird. Die *Leave*-Funktion setzt die Seitentabelle schließlich wieder zurück auf die taskeigene Tabelle. Diese beiden Funktionen können in Zukunft aber auch als *sichtbarer Übergang* für den Einsatz der *Schutzringe* dienen.

Bei den *x86-Schutzringen* handelt es sich um ein vierstufiges Privilegiensystem, wobei die Berechtigungen von innen nach außen abnehmen. Ein Betriebssystemkern im *Ring 0* hat demnach vollen Systemzugriff und darf sämtliche Instruktionen ausführen, während eine Applikation im *Ring 3* allein beim Versuch eine privilegierte Operation auszufüh-

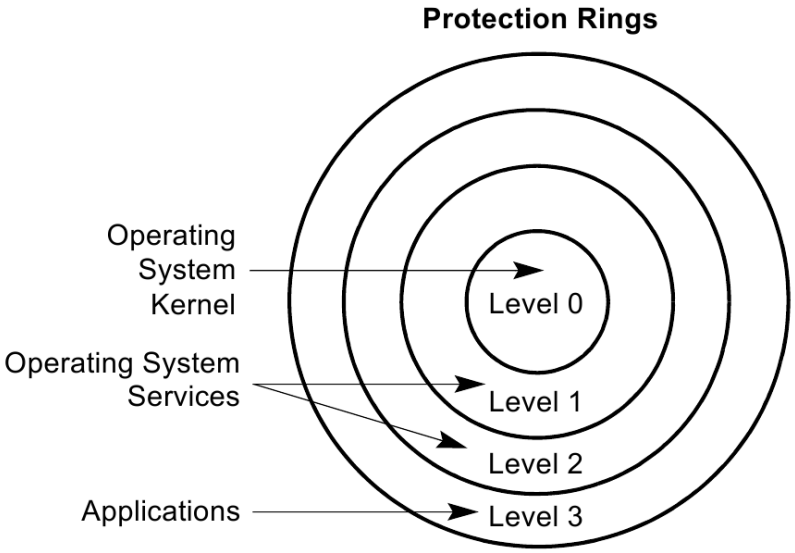

Abbildung 4.9: *x86*-Schutzringe (Quelle: [9, Kapitel 5.5])

ren, einen *General-Protection-Fault* (kurz: *#GP*) auslöst. Der Einsatz der Schutzringe ist prinzipiell notwendig, da sonst eine böswillige Applikation im *Ring 0* trotz aktiviertem Speicherschutz die Verwendung einer eigenen, „liberaleren“ Seitentabelle veranlassen kann. Da aber für das Umschalten der Seitentabelle Zugriffe auf die Kontrollregister notwendig sind, könnte eine statische Analyse des Programmcodes frühzeitig eine potentielle Gefahr entdecken [22].

5 Evaluation

Im folgenden Kapitel wird eine Evaluation der im Rahmen dieser Arbeit entwickelten *x86*-Variante von *CiAO* durchgeführt. Zunächst wird eine *quantitative Evaluation* durchgeführt. Dort werden zum einen die Sektionsgrößen unterschiedlicher Konfigurationen ermittelt und zum anderen Laufzeitunterschiede gemessen. Im Anschluss folgt eine *Evaluation der aspektorientierten Programmierung*, die bei der Entwicklung dieser *CiAO*-Variante angewendet wurde.

5.1 quantitative Evaluation

Wie in Kapitel 2.1 festgestellt wurde, ist einer der größten Vorteile der hochgradigen Konfigurierbarkeit, dass nicht benötigte Komponenten vollständig deaktiviert werden können und demnach weder Speicherplatz noch Rechenleistung verbrauchen.

Für die *quantitative Evaluation* der *x86*-Variante von *CiAO* wurden verschiedene Konfigurationen zusammengestellt, um sie anschließend direkt miteinander zu vergleichen. Die Ergebnisse spiegeln die Vorteile wider, die ein Anwender des konfigurierbaren Systems hat, wenn dieser das Betriebssystem optimal auf seine Bedürfnisse anpasst.

5.1.1 Sektionsgrößenvergleich

Zunächst werden die einzelnen Sektionsgrößen der jeweiligen Kernelimages verglichen. Im Wesentlichen betrifft dies die Größe der `.text`-, `.data`- und `.bss`-Sektionen. Diese lassen sich relativ leicht ermitteln, da diese Informationen direkt im *ELF*-Header des Kernelimages vermerkt sind und sich mit den Hilfsprogrammen `size`, `objdump` und `readelf` der verwendeten Toolchain auslesen lassen.

In Tabelle 5.1 werden verschiedene Konfigurationen des nativen *CiAO*-Systems verglichen.

Im ersten Abschnitt der Tabelle wird das 32-Bit Minimalsystem mit der Maximalkonfiguration verglichen. Anhand der Größe der `.text`-Sektion kann man erkennen, wie wichtig insbesondere bei eingebetteten Betriebssystemen die statische Konfigurierbarkeit ist. Gerade hier haben Schutzmechanismen und Treiber einen hohen Anteil an der Gesamtgröße eines Systems. Der 64 KB große Initialstack erklärt die verhältnismäßig große `.bss`-Sektion.

Im zweiten Abschnitt werden einige Treiberkonfigurationen miteinander verglichen. Zunächst wird der automatisierte Taskwechsel über das *TSS* betrachtet. Der leichte Anstieg der `.data`-Sektionsgröße wird durch die Verwaltungsstrukturen der *Globalen Deskriptor Tabelle* verursacht, die in allen anderen 32-Bit-Konfigurationen nicht zwingend

aktivierte Merkmale														Sektionsgrößen		
64-Bit	**Paging**	**PAE-Paging**	**Speicherschutz**	**TSS-Taskwechsel**	**x87**	**PIT**	**AT-Keyboard**	**COM1**	**PCI**	**PRO/1000**	**Virtionet**	**IDE**	**Input-/Outputstream**	***.text***	***.data***	***.bss***
☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	7718	2127	73038
☐	☒	☒	☒	☐	☒	☒	☒	☒	☒	☒	☒	☒	☒	19290	2127	4578121
☐	☐	☐	☐	☒	☐	☐	☐	☐	☐	☐	☐	☐	☐	9249	2133	73678
☐	☐	☐	☐	☐	☐	☐	☐	☐	☒	☐	☐	☐	☐	8346	2127	74608
☐	☐	☐	☐	☐	☐	☐	☐	☐	☒	☒	☐	☐	☐	10407	2127	140821
☐	☐	☐	☐	☐	☐	☐	☐	☐	☒	☐	☐	☒	☐	9717	2127	74780
☐	☒	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	8344	2127	4275540
☐	☒	☒	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	8908	2127	4275540
☐	☒	☒	☒	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	14906	2127	4275880
☒	☒	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	11744	4245	4280888
☒	☒	☐	☒	☐	☒	☒	☒	☒	☒	☒	☒	☒	☒	26053	4245	4586461
☒	☒	☐	☐	☐	☐	☐	☐	☐	☒	☐	☐	☐	☐	12366	4245	4282456
☒	☒	☐	☐	☐	☐	☐	☐	☐	☒	☒	☐	☐	☐	14605	4245	4348784
☒	☒	☐	☐	☐	☐	☐	☐	☐	☒	☐	☐	☒	☐	13749	4245	4282632

Tabelle 5.1: Codegrößenvergleich

benötigt werden. Der Anstieg der `.bss`-Sektionsgröße spiegelt die nun größere Taskkontextstruktur wider. Die folgenden drei Konfigurationen vergleichen die PCI-basierten Treiber. Hier hebt der PCI-Basistreiber die Größe der `.bss`-Sektion bereits um 1,5 KB an, da ein Teil des PCI-Konfigurationsraums in den Hauptspeicher kopiert werden muss. Besonders auffällig ist die nahezu Verdoppelung des von der `.bss`-Sektion belegten Speicherplatzes beim Aktivieren der *PRO/1000*-Treiber. Diese benötigen jeweils 32 KB für die Sende- und Empfangspuffer. Der IDE-Treiber benötigt im Vergleich dazu keine nennenswerten großen Puffer.

Im dritten Abschnitt der Tabelle wird die seitenbasierte Speicherverwaltung untersucht. Ins Auge sticht der explosionsartige Anstieg der Größe der `.bss`-Sektion. Dies liegt in dem in Kapitel 4.1.4 beschriebenen Einsatz eines 4 MB großen Speicherpools begründet, der pauschal Speicher reserviert, um eine schrittweise Vergrößerung der Seitentabelle zuzulassen. Eine zukünftige Freispeicherverwaltung kann dieses Problem beheben. Das *PAE-Paging* nutzt ausschließlich den Speicherpool, um die zusätzliche Tabellenstufe zu realisieren, so dass weder die `.data`- noch die `.bss`-Sektionsgröße beeinflusst wird. Zum Abschluss wird der Speicherschutzmechanismus betrachtet, der insgesamt den höchsten

Anstieg der `.code`-Sektionsgröße verursacht. Einen großen Anteil davon werden durch die *Binding-Aspekte* zum Umschalten der Seitentabelle beigetragen, die jeden Systemaufruf beeinflussen.

Im unteren Tabellenteil werden die 64-Bit-Variaten des *CiAO*-Systems untersucht. Zunächst werden wiederum die Minimal- und Maximalkonfigurationen gegenübergestellt. Im Vergleich zum 32-Bit-System fällt die nahezu doppelt so große `.data`-Sektion auf, die größtenteils durch die Verdoppelung der Länge aller Deskriptoren verursacht wird.

Im letzten Abschnitt der Tabelle 5.1 werden einige 64-Bit Treiber betrachtet. Wie bereits in Kapitel 3.1.3 erklärt wurde, bleibt die Ansteuerung der Peripheriegeräte bei der Umstellung von 32-Bit auf 64-Bit unverändert. Dies spiegelt sich auch an den Größenzuwachs der `.code`-Sektionen bei Aktivierung der Treiber wider, der sich in ähnlicher Größenordnung wie beim 32-Bit-System bewegt.

5.1.2 Laufzeitenvergleich Kontextwechsel

Im ersten Laufzeitenvergleich wird die Kontextwechselzeit eines Testprogramms gemessen, um die Auswirkungen einzelner Merkmale auf die Laufzeit experimentell zu bestimmen. Als Beispielprogramm dient ein einfaches Taskwechselprogramm mit zwei Tasks, die jeweils 100 Mio. Taskwechsel auf den jeweils anderen Task anfordern. Durch die ständigen Umschaltvorgänge lassen sich die zeitlichen Kosten aller Erweiterungen bestimmen, die den Taskkontext erweitern. Im Folgenden werden so der Speicherschutzmechanismus, der TSS-Taskwechsel und die Gleitkommazahl-Unterstützung genauer betrachtet.

Messmethode

Um die benötigte Zeit für 200 Mio. Kontextwechsel zu messen, wurde der *Time-Stamp Counter* moderner *x86*-Prozessoren genutzt. Dabei handelt es sich um einen 64-Bit Zähler, der bei jedem Systemreset zurückgesetzt wird und mit dem maximalen Prozessortakt hochgezählt wird [9, Kapitel 16.11]. Bei dem verwendeten Testrechner *Dell Optiplex 755* mit einem *Intel Core 2 Quad Q9550* Prozessor zählt der *Time-Stamp Counter* mit 2,83 GHz herauf. Direkt nachdem die Tasks initialisiert wurden, wird der Zähler ausgelesen, um die Startzeit zu ermitteln. Anschließend werden die Taskwechsel durchgeführt. Zum Schluss wird der Zähler erneut ausgelesen und die Differenz zum Startwert berechnet und ausgegeben. Daraus kann anschließend die Zeit für einen Kontextwechsel bestimmt werden.

Ergebnis

In Abbildung 5.1 werden die Messergebnisse des Laufzeitenvergleichs dargestellt. Getestet wurden einige native 32-Bit- und 64-Bit-Konfigurationen sowie die Gastsysteme. Die jeweiligen Minimalsysteme führen einen Kontextwechsel in unter 6 ns durch, wobei auffällt, dass die 64-Bit-Variante etwas schneller ist. Der Grund dafür liegt an der

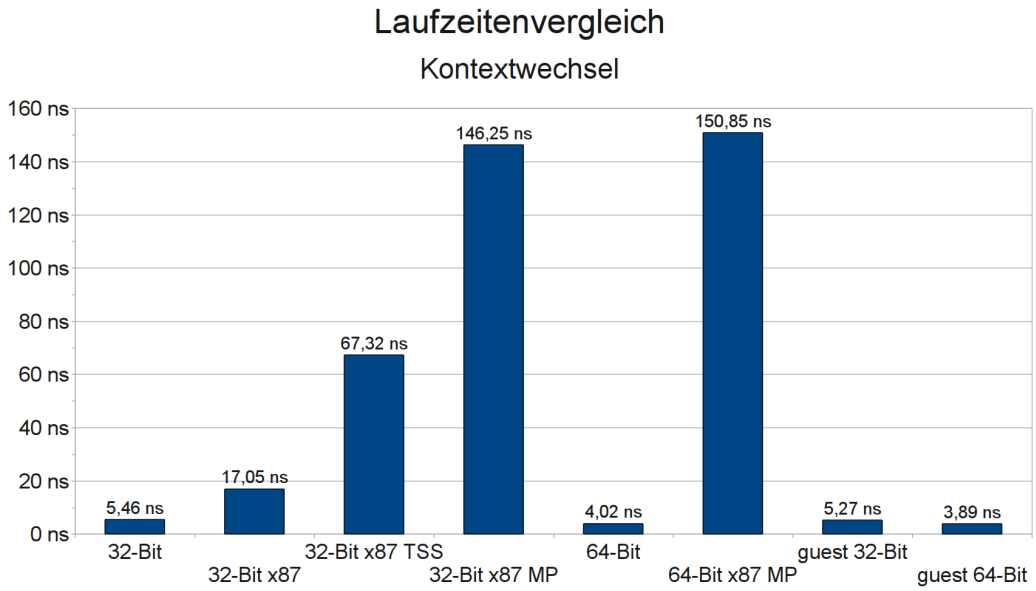

Abbildung 5.1: Laufzeitenvergleich Kontextwechsel

neuen Aufrufkonvention, bei der die Funktionsparameter per Register übergeben werden [23, Kapitel 3.2]. Die Linux-Gastsysteme, die als einzige Applikation unter einem unmodifizierten Linux 2.6.34 getestet wurden, waren minimal schneller als die nativen Varianten.

Interessanter gestaltet sich die Betrachtung der erweiterten Systeme. Bei Nutzung der *x87*-Einheit zur Gleitkommazahlberechnung verlängert sich die Kontextwechselzeit auf 17 ns und dauert demnach ein Vielfaches länger. Ursachen hierfür finden sich bei den Kontextsicherungs- und Wiederherstellungsroutinen, die nun zusätzlich den gesamten 108 Bytes großen FPU-Kontext berücksichtigen müssen [8, Kapitel 8.1.10].

Der automatisierte Kontextwechsel über das *TSS* dauert im Vergleich dazu mehr als 67 ns. Dies scheint zunächst überraschend, da man eigentlich davon ausgehen würde, dass ein hardwaregestützter Taskwechsel im Vorteil sein müsste. Bei genauerer Betrachtung erkennt man allerdings die Nachteile dieses Systems. Das *CiAO*-System ohne Speicherschutz verwendet genau eine Seitentabelle, die niemals gewechselt werden muss. Bei Verwendung des *TSS* wird aber zwangsläufig ein Seitentabellenwechsel ausgelöst, auch wenn die Zieltabelle mit der Quelltabelle identisch ist. Dies invalidiert unter anderem auch den *Translation Lookaside Buffer* (kurz: *TLB*), der anschließend erst wieder aufgebaut werden muss, was folgende Speicherzugriffe verzögert. Weiterhin tauscht das *TSS* alle Speichersegmentregister aus, die beim flachen Speichermodell ebenfalls konstant sind. Auch dies kostet unnötig viel Zeit. Ein weiteres Problem beim *TSS-Taskwechsel* ist der Einsatz der *x87*-Einheit. Diese wird vom hardwaregestützten Kontextwechsel nicht berücksichtigt, so dass die FPU-Register wie beim konventionellen Taskwechsel manuell gesichert und wiederhergestellt werden müssen.

Die *CiAO*-Systeme mit Speicherschutz brauchen mit 146 ns bzw. 151 ns am längsten

für einen Kontextwechsel. Grund ist hierbei das ständige Wechseln der Seitentabelle bei jedem Systemaufruf und jedem Taskwechsel. Beim *TSS*-Taskwechsel wird genau einmal die Seitentabelle ausgetauscht, während beim manuellen Taskwechsel mit Speicherschutz zweimal die Seitentabelle ausgewechselt werden muss.

5.1.3 Laufzeitenvergleich Ethernet

Im zweiten Laufzeitenvergleich wird die Sendedauer eines Ethernetpaketes mit 1500 Bytes Nutzlast gemessen. Damit kann die Geschwindigkeit des *PRO/1000*-Treibers mit dem *VirtioNet*-Treiber verglichen werden. Ein Beispielprogramm versendet dazu 1 Mio. Ethernetpakete. Dieses Programm wird jeweils mit einem der beiden Treiber und jeweils in der 32-Bit- und der 64-Bit-Varianten ausgeführt.

Messmethode

Um einen direkten Vergleich zu ermöglichen werden die Zeiten mit *QEMU* ohne *KVM-Unterstützung* gemessen. Als virtuelles Netzwerk dient das *dump*-Backend des *QEMUs*, das dazu verwendet werden kann, sämtliche versendete Netzwerkpakete des Gastsystems aufzuzeichnen, um sie später analysieren zu können. Bei den Zeitmessungsdurchgängen wurde diese Protokolldatei allerdings sofort verworfen, da sonst bei den zu erwartenden Durchsatzraten Verzögerungen auftreten können, die vom Hostsystem verursacht werden. Durch die rein virtuelle Umgebung spielen physikalische Gegebenheiten keine Rolle, so dass eine 1000-MBit-Ethernetkarte deutlich mehr als 1 GBit/s erreichen kann. Die Geschwindigkeit wird somit nur von der Rechenleistung des Hostsystems, in diesem Fall ein *x86*-Rechner mit einem *Intel Core i5 750* Prozessor, begrenzt. Da die Taktfrequenz konstant auf 1,2 GHz festgesetzt wurde, können die gemessenen Geschwindigkeiten direkt miteinander verglichen werden. Zur Zeitmessung dient erneut der *Time-Stamp Counter*, der auch unter *QEMU* mit 2,67 GHz, der nominalen Maximalfrequenz des Hostprozessors, heraufzählt.

Ergebnis

In Abbildung 5.2 werden die gemessenen Ergebnisse dargestellt. Die unteren Balken geben die verbrauchte Zeit für das Senden eines Ethernetpaketes wieder, während die oberen Balken den daraus resultierenden Datendurchsatz zeigen.

Es fällt sofort auf, dass der emulierte *PRO/1000*-Netzwerkadapter mit 7,5 GBit/s bzw. 8,2 GBit/s wesentlich schneller ist als der *Virtio*-Adapter, der nur 5,1 GBit/s erreicht. Dies ist überraschend, da der *Virtio*-Adapter im Gegensatz zur *PRO/1000* dahingehend optimiert ist, in virtualisierten Umgebungen betrieben zu werden. Grund für dieses verblüffende Ergebnis könnten die nicht genutzten Sonderfunktionen des *Virtio*-Adapters sein. So kann das *VirtioNet* mit einer *Maximum Transmission Unit* (kurz: *MTU*) von 65535 Bytes betrieben werden, während normale Netzwerkkarten meistens auf 1500 Bytes beschränkt sind (ausgenommen *Jumbo Frames*). Dies würde die Anzahl benötigter Systemaufrufe um ein Vielfaches verringern. Ein weiterer Grund für das Messergebnis

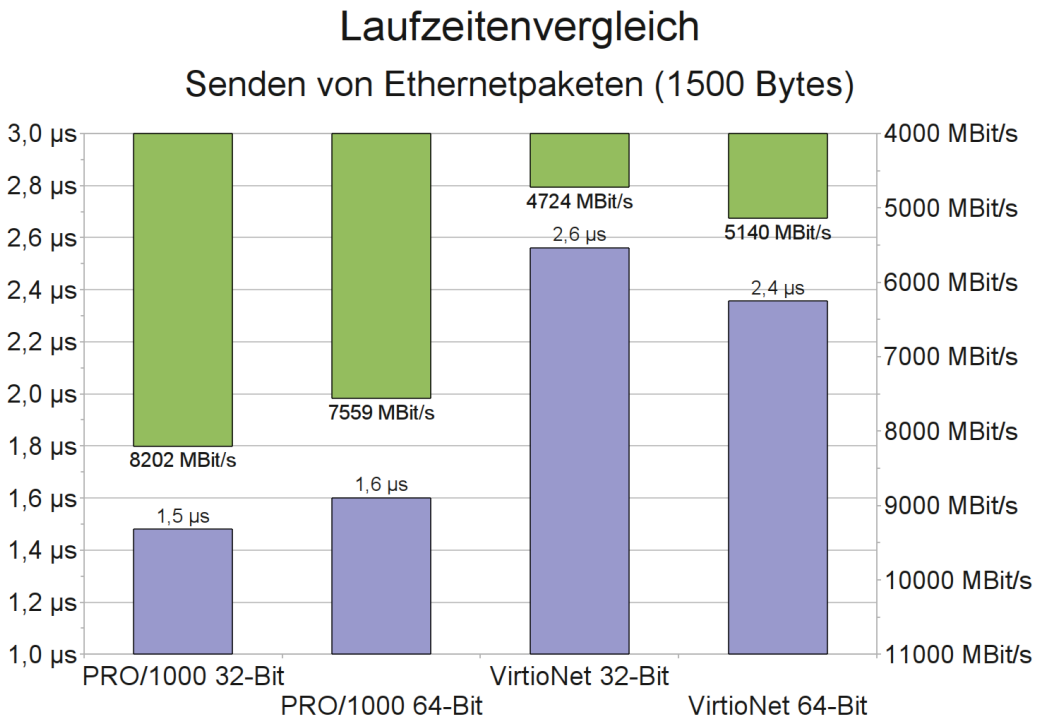

Abbildung 5.2: Laufzeitenvergleich Ethernet

könnte an *QEMU* selbst liegen. Im Vergleich zur *PRO/1000*-Emulation sind die *Virtio*-Module des *QEMUs* noch relativ jung. Es ist daher zu vermuten, dass sich die Laufzeit in Zukunft noch verbessern könnte.

Ein weiteres interessantes Ergebnis ist die Tatsache, dass der 64-Bit-Treiber von *VirtioNet* schneller ist als sein 32-Bit-Pendant, beim *PRO/1000*-Treiber aber genau der umgekehrte Fall vorliegt. Dies liegt am Aufbau der internen Register der *PRO/1000* begründet. Dabei handelt es sich größtenteils um 32-Bit-Register. Einzig allein bei den Adressen im Deskriptorring handelt es sich um 64-Bit-Werte. Unter Umständen muss im 64-Bit-Modus häufig Code generiert werden, um 32-Bit-Werte auf 64-Bit aufzufüllen, was im 32-Bit-Modus entfallen kann.

5.2 softwaretechnische Evaluation der Aspektorientierten Programmierung

Die *softwaretechnische Evaluation* betrachtet die neu entstandene *x86*-Variante von *CiAO* unter dem Gesichtspunkt der *Aspektorientierten Programmierung*. Insbesondere soll in diesem Kapitel festgestellt werden, wie Aspekte den Entwurf und die daraus resultierende Implementierung beeinflusst haben.

5.2.1 Auswirkungen der Belange

In diesem Abschnitt werden die Auswirkungen der in Kapitel 3 behandelten Belange untersucht.

5.2.1.1 Gastsystem

Initialisierung (guest1) Bei der Initialisierung des Gastsystems wurde die gesamte Initialisierungsroutine ausgetauscht. Ein aspektorientierter Entwurf ist hier nicht möglich, da der *Startupcode* vollständig in der *x86*-Assemblersprache geschrieben wurde. Der Einsatz von *pure::variants* ermöglicht es aber, zum Konfigurationszeitpunkt eine unabhängige Ersatzimplementierung der Startroutinen für das Gastsystem anstelle des Originalcodes bereitzustellen. ×

Treiber (guest2) Sämtliche Treiber des nativen Systems können prinzipbedingt nicht im Gastsystem verwendet werden. Eine aspektorientierte Modifizierung der nativen Treiber wäre zwar möglich, aber dann müssten alle hardwarespezifischen Zugriffe emuliert werden, um dieselben Schnittstellen anbieten zu können. Daher wurde beim Entwurf ähnlich wie bei der Initialisierung eine Ersatztreiberschicht per *pure::variants* eingebunden. ×

5.2.1.2 64-Bit-Unterstützung

Initialisierung (lm1) Bei der Initialisierung des 64-Bit-Systems stößt man auf dasselbe Problem wie bei der Initialisierung des Gastsystems. Auch hier muss der hardwarenahe *Startupcode* angepasst werden, was wiederum nur auf Dateiebene funktioniert. Zusätzlich beinhaltet der 64-Bit-Startupcode teilweise auch 32-Bit-Code. Da die verwendeten Hochsprachen von dieser Unterscheidung abstrahieren, lässt sich die *64-Bit-Umschaltung* im *Startupcode* nicht mit *AspectC++* umsetzen. ×

Kontextwechsel (lm2) Der Kontextwechsel, insbesondere das Sichern und Wiederherstellen der Register, wurde ebenfalls in Assemblersprache realisiert. Da die Prozeduren für den Kontextwechsel keine Gemeinsamkeiten aufweisen, wo ein aspektorientiertes Anknüpfen sinnvoll wäre, wurde auch hier ausschließlich auf Dateiebene gearbeitet. ×

Paging (lm3) Im 64-Bit-Modus kann ausschließlich nur das *64-Bit-Paging* genutzt werden. Das *Paging* wird in Kapitel 5.2.1.5 näher betrachtet.

5.2.1.3 *x87* Gleitkommazahleinheit

Initialisierung (fpu1) Die Initialisierungsroutinen für die *FPU* wurden per *loser Kopplung* an die `start()`-Funktionen der *Continuation*-Klasse gebunden, damit sie bei jedem Starten eines Tasks initialisiert wird. Somit befindet sich die *FPU* für jeden Task zu Beginn in einem definierten Zustand. ✓

Taskkontext (fpu2) Um den 108 Byte großen *FPU*-Zustand beim Taskwechsel sichern zu können, musste die Taskkontextstruktur erweitert werden. Dies wurde mit einer *minimalen Erweiterung* gelöst, die den benötigten Speicherplatz zur Verfügung stellt.

Die Kontextwechselroutinen selbst konnten mit einer *losen Kopplung* ebenfalls gut aspektorientiert umgesetzt werden. Hilfreich waren dazu auch die von *CiAO* bereitgestellten *sichtbaren Übergänge* `after_CPUReceive()` zum Wiederherstellen des *FPU*-Kontextes und `before_CPURelease()` zum Sichern des Zustands. ✓

TSS-Kontextwechsel (fpu3) Immer wenn die Gleitkommazahleinheit zusammen mit dem *TSS-Taskwechsel* genutzt werden soll, muss die *TSS*-Taskwechselroutine erweitert werden, um das in Kapitel 3.2.1 beschriebene Problem der *#NM*-Exception zu umgehen. Dazu wurden die oben beschriebenen *sichtbaren Übergänge* genutzt, um das *task switched*-Bit zu löschen. ✓

5.2.1.4 TSS-Taskwechsel

Taskkontext (tss1) Um den Taskkontext abzuspeichern, wird beim *TSS-Taskwechsel* das *Task-State-Segment* genutzt, dass ein völlig anderes Format aufweist, wie ein konventioneller Taskkontext. Demnach musste die gesamte Kontext-Datenstruktur ersetzt werden, was durch ein Austauschen der Header mit den `struct`-Definitionen realisiert wurde. ×

Kontextwechsel (tss2) Ähnlich wie bei der *64-Bit-Unterstützung* unterscheiden sich die Kontextwechselroutinen so grundlegend, dass diese komplett ausgetauscht werden mussten. So werden auch hier die Implementierungsdateien vollständig ausgetauscht. ×

Deskriptortabelle (tss3) Das Anlegen und Eintragen der *Task-Deskriptoren* in die *Globale Deskriptor Tabelle* wird von einem *Binding-Aspekt* übernommen, der sich an die Initialisierungsroutine hängt. ✓

5.2.1.5 seitenbasierte Speicherverwaltung

Seitenformat (mmu1) Wie bereits in Kapitel 4.1.4 beschrieben wurde, binden sich die Funktionen zum Erzeugen eines seitenformatspezifischen Tabelleneintrags per *loser Kopplung* an die `setPage()`-Funktionen. Die dazu benötigten Hilfsstrukturen wie der Speicherpool werden per *minimaler Erweiterung* der *Paging*-Klasse hinzugefügt. ✓

Initialisierung (mmu2) Zur Initialisierung und Aktivierung des *Paging*s, wird die Initialisierungsroutine per *loser Kopplung* an die von *CiAO* bereitgestellten `init()`-Funktionen angebunden. ✓

Speicherzugriff (mmu3) Der Speicherzugriff selbst läuft vollkommen transparent und ohne Eingriff seitens der *Paging*-Schnittstelle ab. Die *Pagefault-Exception* kann mit dem *sichtbaren Übergang* des *Protection-Hooks* behandelt werden. ✓

5.2.1.6 seitenbasierter Speicherschutz

Initialisierung (mp1) Der Aufbau der taskeigenen Seitentabellen wird von einem *Binding-Aspekt* realisiert, der sich an die `init()`-Funktion der *Continuation*-Klasse hängt. Somit wird bei jeder Initialisierung eines Tasks automatisch eine frische Seitentabelle generiert. ✓

Kontextwechsel (mp2) Der zusätzliche Eintrag für die Seitentabelle wurde als *minimale Erweiterung* in die Kontextdatenstruktur eingefügt. Der Wechsel der Seitentabelle muss nicht zusätzlich implementiert werden, da die Kontextwechselfunktionen selbst Systemaufrufe sind, und daher vom folgenden Punkt mit übernommen werden. ✓

Systemaufrufe (mp3) Wie bereits in Kapitel 4.5 erkannt wurde, handelt es sich beim Erfassen sämtlicher Systemaufrufe um ein „Paradebeispiel“ der aspektorientierten Programmierung, da ein Aspekt ausreicht, um alle (*quantification*) Systemfunktionen, die dafür nicht speziell präpariert werden mussten (*obliviousness*), beeinflussen zu können. ✓

Unterbrechungsbehandlung (mp4) Die Routinen der Unterbrechungsbehandlung können ähnlich wie die Systemaufrufe mit einem *Binding-Aspekt* erfasst werden. ✓

5.2.1.7 AT-Keyboard

Tastaturlayout (kb1) Zur Dekodierung der Scancodes wird eine einfache Tabelle benötigt, die als *minimale Erweiterung* realisiert wurde. In der aktuellen Implementierung befinden sich Scancodetabellen für das deutsche und englische Tastaturlayout, die als *Alternativen* unter *pure::variants* zur Verfügung gestellt werden. Bei Bedarf lassen sich aber leicht neue Tabellen hinzufügen. ✓

Initialisierung (kb2) Wie bei meisten Initialisierungsroutinen bindet sich auch diese Routine an die globale `init()`-Funktion. ✓

5.2.1.8 Intel PRO/1000

Sende- und Empfangsfunktionen (e1) Die allgemeinen Sende- und Empfangsfunktionen wurden als *Klassen-Slices* implementiert und lassen sich daher optional als *minimale Erweiterung* in das System weben. Damit besteht auch die Möglichkeit, den Speicherplatz für nicht benötigte Puffer zu sparen. ✓

Offloading-Unterstützung (e2) Die *Offloading*-Sendefunktion wurde als *lose Kopplung* an eine der normalen `send()`-Funktionen realisiert. Im Normalfall berechnet diese

Funktion keine Prüfsummen, so das der Benutzer des Treibers die Berechnung in Software vornehmen muss. Bei aktiviertem *Offloading* wird diese Funktion so modifiziert, dass sie einen Teil der Berechnungen auf die Netzwerkhardware auslagert.

Die *Offloading*-Empfangsfunktion benötigt wie in Kapitel 4.1.2 beschrieben wurde, nur eine leicht modifizierte Initialisierungsroutine der Netzwerkkarte. Diese wurde als *lose Kopplung* an die `init()`-Funktion realisiert. ✓

Initialisierung (e3) Eine *lose Kopplung* an die Initialisierungsfunktion des PCI-Treibers stellt sicher, dass die *PRO/1000* erst initialisiert wird, wenn die Infrastruktur des PCI-Treibers bereits zur Verfügung steht. Alternativ hätte dies mit einem *Order-Advice* gelöst werden können, dessen Flexibilität hier aber nicht benötigt wurde, da die Netzwerkkarte immer vom PCI-Treiber abhängt. ✓

Paging (e4) Da die *PRO/1000*-Netzwerkkarten ihre internen Register in den normalen Speicherraum des *x86* einblendet, unterliegt sie der Adressumrechnung des *Pagings*. Bei gleichzeitiger Aktivierung des *PRO/1000*-Treibers und der *seitenbasierten Speicherverwaltung*, hängt sich ein *Binding-Aspekt* vor die Initialisierungsfunktion der Netzwerkkarte, um mit Hilfe der *Paging*-Schnittstelle passende Seiten für die *memory-mapped* Register anzulegen. ✓

5.2.1.9 IDE

Adressierungsart (ide1) Die drei Adressierungsarten *CHS*, *LBA28* und *LBA48* ließen sich leicht als *minimale Erweiterung* implementieren. Diese fügen die entsprechenden sektorbasierten Lese- und Schreibfunktionen in die `IDE`-Klasse ein. Der Aspekt für die *CHS*-Adressierung stellt außerdem noch Umrechnungsfunktionen bereit, damit alle Funktionen mit lineare Sektoradressen arbeiten können. ✓

DMA (ide2) Die *DMA-Unterstützung* wurde ebenfalls mit einem Aspekt zu *minimalen Erweiterung* realisiert. Diese stellen neue Funktionen bereit, die Lese- und Schreibanforderungen im Hintergrund erledigen. ✓

Initialisierung (ide3) Die Initialisierung des IDE-Treibers wurde wie beim *PRO/1000*-Treiber umgesetzt. Auch hier ist ein Zugriff auf den PCI-Treiber notwendig, so dass sich die Initialisierungsroutine des IDE-Treibers per *loser Kopplung* an die `init()`-Funktion des PCI-Treibers bindet. ✓

5.2.2 Zusammenfassung

Die Tabelle 5.2 fasst den Einsatz von Aspekten innerhalb der *x86*-Variante abschließend nochmal zusammen.

In der Tabelle wird jeweils die Art dargestellt, wie die einzelnen Belange des *CiAO*-Systems umgesetzt wurden. Dabei steht „∞“ für einen *Binding-Aspekt* zur Umsetzung einer *losen Kopplung*, „+“ für einen *Extension-Slice* zur Realisierung einer *minimalen*

	Beeinflussungen					
	Init.	**Kontextwechsel**	**Paging**	**Systemaufrufe**	**Unterbrechungen**	**Treiberkonfig.**
guest1	×					
guest2					×	
lm1	×				×	
lm2		×				
lm3			⊚ +			
fpu1	⊚					
fpu2		§ +				
fpu3		§				
tss1		×				
tss2		×				
tss3	⊚					
mmu1			⊚ +			
mmu2			⊚			
mmu3				§		
mp1	⊚					
mp2		+				
mp3				⊚		
mp4					⊚	
kb1						+
kb2	⊚					
e1						+
e2						⊚
e3	⊚					
e4			⊚			
ide1						+
ide2						+
ide3	⊚					

Legende:

⊚ : *Binding-Aspekt*
§ : *Policy-Aspekt*
\+ : *Extension-Slice*
× : nicht-aspektorientierte Umsetzung

guest:	Gastsystem	**lm**:	64-Bit-Modus
fpu:	*x87*-Gleitkommazahleinheit	**tss**:	*TSS*-Taskwechsel
mmu:	Paging	**mp**:	Speicherschutz
kb:	Keyboard-Treiber	**e**:	*PRO/1000*-Treiber
ide:	*IDE*-Treiber		

Tabelle 5.2: Einsatz von Aspekten

Wirkbereich	**mögliche Pointcuts**
Initialisierung	`"% hw::init()"`, `"% hw::hal::init()"`
Kontextwechsel	`"% after_CPUreceive()"`, `"% before_CPUrelease()"`
Paging	`"% enable()"`, `"% disable()"`, `"% setPage(...)"`
Systemaufrufe	`"% AS::%(...)"`
Unterbrechungen	`"% hw::irq::guardian(%)"`, `"% hw::irq::%::Handler()"`
Treiberkonfiguration	`"% hw::dev::%::%(...)"`

Tabelle 5.3: Pointcuts

Erweiterung und „§“ für den Einsatz eines *Policy-Aspekts* bei einem *sichtbaren Übergang*. Eine nicht-aspektorientierte Implementierung auf Dateiebene wird mit „×“ dargestellt.

In Tabelle 5.3 werden nochmal die einzelnen Beeinflussungsorte mit den zugehörigen *Pointcuts* zusammengefasst.

Fazit

Bei genauerer Betrachtung der Punkte, wo die aspektorientierte Programmierung nicht eingesetzt werden konnte, erkennt man, dass es sich dabei hauptsächlich um Änderungen in sehr hardwarenahen Bereichen handelt. Insbesondere die frühen Systeminitialisierungsroutinen des *Startupcodes* lassen keine aspektorientierte Programmierung zu. Dies ist nicht zwingend ein Problem der Aspektorientierung, da in diesen Bereichen selbst einfachere Hochsprachen versagen.

Besonders gut lässt sich die aspektorientierte Programmierung hingegen bei Erweiterungen einsetzen. So ließ sich die Unterstützung für das *Paging* oder der *x87-FPU* nahtlos ins System integrieren, obwohl diese Belange in *CiAO* ursprünglich nicht vorgesehen waren. Treiber ließen sich ebenfalls leicht in *CiAO* integrieren, da sie in der Regel keine tiefgreifenden Änderungen des Systems erfordern. Genutzt wurden dazu hauptsächlich *Binding-Aspekte* zur Beeinflussung des Kontrollflusses und *Extension-Slices* zur Erweiterung von bestehenden Datenstrukturen sowie zur Bereitstellung von erweiterten Funktionen. *Policy-Aspekte* hingegen konnten nur relativ selten eingesetzt werden.

6 Zusammenfassung und Ausblick

Mit der im Rahmen dieser Arbeit entwickelten *x86*-Variante des *CiAO*-Systems hat die Betriebssystemfamilie *CiAO* ein neues Familienmitglied erhalten. Damit wurde der Grundstein gelegt, das bislang nur auf eingebetteten Mikrocontrollersystemen lauffähige hochkonfigurierbare Betriebssystem auf der weitverbreiteten PC-Architektur ausführen zu können. Mit der Entwicklung des Gastsystems und der Unterstützung des *Virtio*-Frameworks wurde außerdem der Weg geebnet, *CiAO* auf virtualisierten und paravirtualiserten Umgebungen zu nutzen. Die eingesetzte *aspektorientierte Programmierung* konnte insbesondere beim Einbinden der neuen *x86*-spezifischen Funktionen überzeugen. Trotz der teils sehr plattformspezifischen Eigenschaften dieser Funktionen, konnten diese in das bestehende System integriert werden, ohne dass dazu das Kernsystem oberhalb der Hardwareabstraktionsschicht verändert werden musste.

Die neue *CiAO*-Variante kann in Zukunft als Grundlage für weitere Entwicklungen dienen.

POSIX-Personality

Das Konzept der unterschiedlichen *Personalities* könnte wieder aufgegriffen werden. Dabei handelt es sich um eine Abstraktionsschicht für Systemaufrufe. Zur Zeit existiert nur eine *AUTOSAR-OS*-kompatible *Personality* für *CiAO*. Denkbar wäre aber auch eine konfigurierbare *POSIX-Personality*. Mit einer Untermenge der *POSIX*-Funktionen könnte die Zahl der unter *CiAO* lauffähigen Anwendungen explosionsartig steigen, da damit ansatzweise eine Quellcode-Kompatibilität hergestellt werden könnte. Durch die Konfigurierbarkeit könnten die Kosten für unnötige *POSIX*-Funktionalitäten vermieden werden.

Gastsystem Treibermodell

Ein weiterer Ansatzpunkt für zukünftige Entwicklungen bietet aber auch die neue Gast-Variante von *CiAO*. Das dort verwendete Treibermodell orientiert sich zunächst an der Struktur der nativen Treiber. Allerdings könnte es sich lohnen, ein eigenes Treibermodell für die Gast-Implementierung zu entwickeln.

Paravirtualisierung

Das *Virtio*-Framework bietet die Möglichkeit, paravirtualisierte Peripheriegeräte auf definierte Art und Weise zu nutzen. Die Paravirtualisierung ist aber nicht nur auf Peripheriegeräte beschränkt. Durch Einsatz einer allgemeineren Virtualisierungsschnittstelle

wie *VMware-VMI* [24] oder *Xen Hypercalls* [25] können auch grundlegende Belange wie die Unterbrechungsbehandlung oder die Speicherverwaltung vom Hostsystem virtualisiert werden. Dies könnte zu einer allgemeinen Leistungssteigerung beitragen.

Literaturverzeichnis

[1] Lohmann, Daniel ; Hofer, Wanja ; Schröder-Preikschat, Wolfgang ; Streicher, Jochen ; Spinczyk, Olaf: CiAO: An Aspect-Oriented Operating-System Family for Resource-Constrained Embedded Systems. In: *Proceedings of the 2009 USENIX Annual Technical Conference*. Berkeley, CA, USA : USENIX Association, Juni 2009, S. 215–228

[2] Spinczyk, Olaf: *Vorlesungsfolien zu Software ubiquitärer Systeme*. 2009

[3] Beuche, Danilo: *pure::variants: Einführung Konzepte und UI*. 2005

[4] Lohmann, Daniel ; Spinczyk, Olaf ; Schröder-Preikschat, Wolfgang: Lean and Efficient System Software Product Lines: Where Aspects Beat Objects. In: Rashid, Awais (Hrsg.) ; Aksit, Mehmet (Hrsg.): *Transactions on AOSD II*, Springer-Verlag, 2006 (Lecture Notes in Computer Science 4242), S. 227–255

[5] Spinczyk, Olaf ; Lohmann, Daniel:

[6] Messmer, Hans-Peter ; Dembowski, Klaus: *PC Hardwarebuch*. Addison-Wesley, 2003. – ISBN 3–8273–2014–3

[7] Thieser, Michael: *PC-Schnittstellen*. Franzis-Verlag, 1996. – ISBN 3–7723–8092–1

[8] Intel Corporation (Hrsg.): *Intel 64 and IA-32 Architectures Software Developer's Manual*. Volume 1: Basic Architecture. Intel Corporation, Dezember 2009

[9] Intel Corporation (Hrsg.): *Intel 64 and IA-32 Architectures Software Developer's Manual*. Volume 3A: System Programming Guide, Part 1. Intel Corporation, Dezember 2009

[10] Intel Corporation (Hrsg.): *IA-PC HPET (High Precision Event Timers) Specification*. Intel Corporation, Oktober 2004

[11] Intel Corporation (Hrsg.): *82540EM Gigabit Ethernet Controller*. Intel Corporation, September 2006

[12] Intel Corporation (Hrsg.): *PCI/PCI-X Family of Gigabit Ethernet Controllers Software Developer's Manual*. Intel Corporation, März 2009

[13] Global Engineering (Hrsg.): *Information Technology - AT Attachment with Packet Interface - 6 (ATA/ATAPI-6)*. 15 Inverness Way East, Englewood, CO, USA: Global Engineering. – Working Draft

[14] Tanenbaum, Andrew S.: *Modern Operating Systems*. Upper Saddle River, NJ, USA : Prentice Hall, 2001. – ISBN 0–13–092641–8

[15] Neiger, Gil ; Santoni, Amy ; Leung, Felex ; Rodgers, Dion ; Uhlig, Rich: Intel Virtualization Technology: Hardware support for efficient processor virtualization. In: *Intel Technology Journal* 10 (2006), August, Nr. 3, S. 167–178. http://dx.doi.org/10.1535/itj.1003.01. – DOI 10.1535/itj.1003.01

[16] Bellard, Fabrice: QEMU, a fast and portable dynamic translator. In: *ATEC '05: Proceedings of the annual conference on USENIX Annual Technical Conference*. Berkeley, CA, USA : USENIX Association, 2005, S. 41–46

[17] Russell, Rusty: virtio: towards a de-facto standard for virtual I/O devices. In: *SIGOPS Oper. Syst. Rev.* 42 (2008), Nr. 5, S. 95–103. http://dx.doi.org/10.1145/1400097.1400108. – DOI 10.1145/1400097.1400108. – ISSN 0163–5980

[18] Dike, Jeff: *User Mode Linux*. Upper Saddle River, NJ, USA : Prentice Hall, 2006. – ISBN 0–13–186505–6

[19] Hofer, Wanja: *Aspect-Oriented Design and Implementation of an AUTOSAR-Like Operating System Kernel*, Friedrich-Alexander-Universität Erlangen-Nürnberg, Diplomarbeit, Oktober 2007

[20] Infineon Technologies AG (Hrsg.): *TC1796 32-Bit Single-Chip Microcontroller TriCore*. Edition 2008-04. Infineon Technologies AG, April 2008

[21] The Open Group (Hrsg.): *Standard for Information Technology - Portable Operating System Interface (POSIX)*. System Interfaces, Issue 6. The Open Group, September 2001

[22] Stilkerich, Michael ; Lohmann, Daniel ; Schröder-Preikschat, Wolfgang: Memory protection at option. In: *CARS '10: Proceedings of the 1st Workshop on Critical Automotive applications*. New York, NY, USA : ACM, 2010. – ISBN 978–1–60558–915–2, S. 17–20

[23] *System V Application Binary Interface AMD64 Architecture Processor Supplement*. : *System V Application Binary Interface AMD64 Architecture Processor Supplement*. Draft Version 0.99.4, Januar 2010

[24] VMware, Inc. (Hrsg.): *Paravirtualization API Version 2.5*. VMware, Inc., Februar 2006

[25] Barham, Paul ; Dragovic, Boris ; Fraser, Keir ; Hand, Steven ; Harris, Tim ; Ho, Alex ; Neugebauer, Rolf ; Pratt, Ian ; Warfield, Andrew: Xen and the art of virtualization. In: *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*. New York, NY, USA : ACM, 2003. – ISBN 1–58113–757–5, S. 164–177

Abbildungsverzeichnis