Hauptseminar
Aspektorientierte Systemprogrammierung

Betriebssystembaukästen: THINK und OSKIT

Inhalt

1 Betriebssystembaukästen: Motivation

Bei Entwurf und Implementierung eines Betriebssystems stößt man auf immer wieder gleiche Komponenten, die realisiert werden müssen. Beispiele wären Bootloader, Gerätetreiber, Debugging-Eintrichtungen und viele mehr. Gerade im Umfeld von Forschung und Lehre, wo meist nur kleinere Gruppen von Entwicklern zur Verfügung stehen, aber durchaus spezialisierte Betriebssystemkerne benötigt werden, ist es selten zweckmäßig, bei der Entwicklung ganz von vorne anzufangen.

Die traditionelle Vorgehensweise in diesem Fall war es, ein bereits existierendes, möglichst passendes Betriebssystem zu suchen und dieses entsprechend umzuformen. Dieses Vorgehen hat aber ernste Nachteile: so wird einerseits viel Entwicklungszeit durch nötige Anpassungen gebunden, die beispielsweise durch infrastrukturelle Abhängigkeiten des alten Systems bedingt sind, andererseits ist man beim Entwurf des neuen Systemes trotzdem noch von den Eigenheiten des alten beeinflusst. Dies wirkt sich zum Beispiel auch im Bereich der eingebetteten Systeme nachteilig aus, wo man auf möglichst gut angepasste Software ohne unnötigen Balast angewiesen ist.

Eine bessere Lösung dieses Dilemmas bieten sogenannte Betriebssystembaukästen, die Sammlungen von Standardkomponenten anbieten. Dabei wird das Hauptaugenmerk nicht auf die Entwicklung eines gesamten Betriebssystems gelegt, sondern auf möglichst gut gekapselte, wiederverwendbare Standardkomponenten. Durch Schnittstellenbeschreibungen in standardisierten Sprachen schafft man die Möglichkeit, auch nur einzelne Komponenten auszutauschen oder zu verwenden, was letztenendes eine enorme Ersparnis von Zeit und damit Geld bei der Entwicklung bedeutet.

Im Folgenden sollen zwei solcher Betriebssystembaukästen näher vorgestellt werden, namentlich OSKit, das an der University of Utah entwickelt wird, sowie THINK, eine Lösung französischen Ursprungs. Bei beiden Baukästen soll sowohl auf grundlegende Hintergedanken, als auch die Implementierung sowie praktische Umsetzungen und daraus gewonnene Erfahrungen eingegangen werden.

2 OSKit

Der erste betrachtete Baukasten, das Flux OSKit, stammt von der Flux Research Group an der University of Utah. Diese entwickelte zu Forschungszwecken an einem Mikrokernelsystem namens Fluke OS und stiess dabei auf die schon genannten Probleme der Reimplementierung von Standardkomponenten. Nachdem dies erkannt wurde, wurde beschlossen, dieses Problem an der Wurzel anzugehen, und OSKit entstand.

2.1 Übersicht

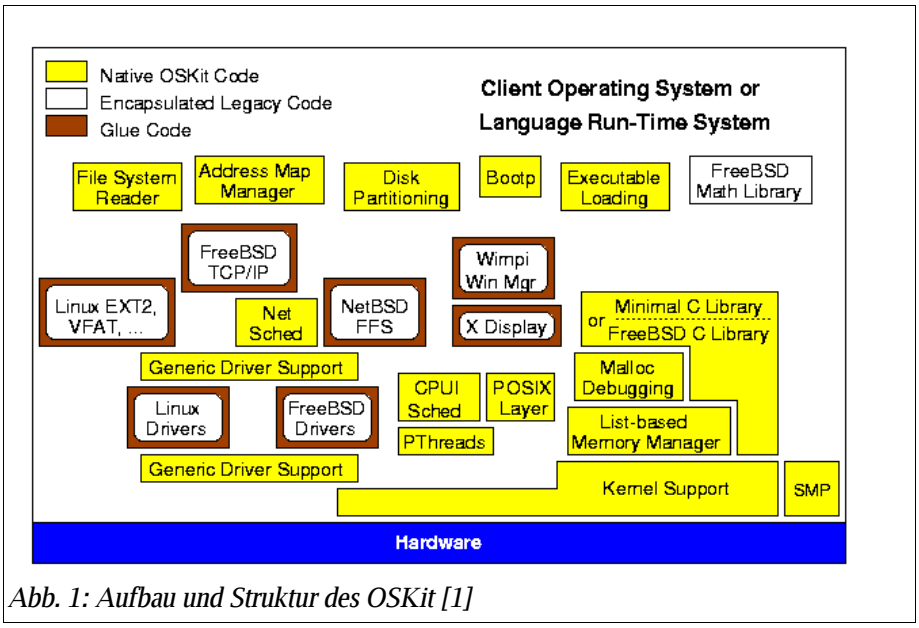

Abb. 1: Aufbau und Struktur des OSKit [1]

Das OSKit ist eine Sammlung von mittlerweile 34 verschiedenen Bibliotheken mit Standardkomponenten, deren groben Aufbau sowie Abhängigkeiten Abbildung 1 veranschaulicht. Diese werden von mehr als 500 Seiten Dokumentation begleitet. Einige der wichtigesten Komponenten sollen nun kurz erwähnt werden.

2.1.1 Bootloader

Dem Bootloader wird bei der Entwicklung eines Betriebssystems meist nur wenig Beachtung zuteil, weswegen gerade im Forschungsbereich oft quick-and-dirty-Lösungen verwendet werden, die sehr stark auf gerade benötigte Bedingungen zugeschnitten und damit selten weiterverwendbar sowie inkompatibel zu anderen Bootloadern sind. Um dem zu entgegnen unterstützt der OSKit Bootloader den *Multiboot Standard*, eine standardisierte Schnittstelle zwischen Bootloader und Kernel. Da dieser Standard relativ neu ist, enthält das OSKit aber auch Werkzeuge, um MultiBoot Kernel von älteren Bootloadern wie *LILO* oder dem BSD Bootloader starten zu können.

2.1.2 Kernel-Support

Die OSKit Kernel-Support-Bibliothek enthält eine grosse Sammlung plattformabhängigen Code zum direkten Zugriff auf die Hardware, beispielsweise (für die x86 Architektur) direkten Zugriff auf page tables, PIT, PIC und ähnliches. Andere OSKit Komponenten verwenden diesen Code, um plattformunabhängige Schnittstellen bereitzustellen; allerdings kann er, um ein Maximum an Flexibilität zu gewährleisten, auch direkt benutzt werden.

Gerade bei der x86 Architektur ist die Kernel-Support-Bibliothek von grossem Nutzen, da sie hier für eine sinnvolle Umgebung sorgt: Sie übernimmt das Umschalten in den 32-bit-Modus, installiert Interrupt-Handler und ähnliches, lädt eventuell vom Bootloader mitgegebene Module in dafür reservierten Speicher und ruft dann die main-Funktion des Betriebssystems auf, der sie etwaige durch den Bootloader übermittelte Argumente und Umgebungsvariablen übergibt. Ein „Hello World“ Kernel, der diese Bibliothek benutzt, sieht dadurch nicht viel anders aus als ein normales „Hello World“ Programm (siehe Abbildung 2).

```
#include <stdio.h>
#include <oskit/clientos.h>
#include <oskit/startup.h>
#include <oskit/version.h>

int main()
{
        oskit_clientos_init();
        oskit_print_version();
        printf("Hello, World\n");
        return 0;
}
```

Abb. 2: ein “Hello World” Kernel mit OSKit [3]

Zusätzlich bietet die Kernel Support Bibliothek eine Debug-Möglichkeit über die serielle Schnittstelle, die das remote debugging Protokoll des *GNU Debuggers* (GDB) benutzt. Dadurch kann, durch ein entsprechendes Hardware-Setup von zwei Rechnern, die mittels serieller Schnittstelle verbunden sind, ein einfaches, aber effizientes Debugging realisiert werden.

2.1.3 Speicherverwaltung

Speicherverwaltungscode, wie man ihn in einer Standard C Bibliothek findet, ist im Bereich der Kernelentwicklung nicht immer ausreichend, da normalerweise einige Hardwarekomponenten ganz spezifische Anforderungen an zugeteilte Speicher-bereiche stellen. So kann beispielsweise der DMA-Controller eines Standard-PCs nur auf die „unteren“ 16 MB des realen Arbeitsspeichers zugreifen. Um dem Rechnung zu tragen bietet das OSKit zwei unterschiedliche Komponenten zur Speicherverwaltung an: Einerseits einen listenbasierten Memory Manager (*LMM*), der umfangreiche Funktionen zum Allokieren von physischem wie virtuellem Speicher, mit verschiedensten Anforderungen z.B. bezüglich des Alignment umgehen kann und es

darüber hinaus ermöglicht, verschiedene „Speichertypen“ in verschiedene „Pools“ einzuordnen; andererseits einen „Address Map Manager“ (*AMM*), der keine direkte Verbindung zwische Adressraum und physischem oder virtuellem Speicher voraussetzt, und beispielsweise Verwendung bei der Verwaltung des Speichers einzelner Prozesse findet.

Zusätzlich enthält das OSKit eine Bibliothek zum Debuggen der Speicherverwaltung, die es ermöglicht, typische Fehler wie Pufferüberläufe oder *double-frees* zu erkennen; eine ähnliche Einrichtung, die sich in vielen Werkzeugen zur Applikationsentwicklung auch findet, allerdings hier eben auf Kernel-Ebene.

2.1.4 C-Bibliothek

Größere Betriebssystemkerne bringen oft eine Menge an reimplementierten C-Bibliotheksfunktionen mit. Dies liegt darin begründet, dass „normale“ libc-Implementierungen meist eine Unmenge an Funktionalität (und damit auch an Abhängigkeiten) mitbringen, die im Kernelbereich nicht gebraucht wird und damit unnötige Ressourcen- sowie Performanzkosten verursacht.

Auch OSKit bietet hierzu eine minimale C-Bibliothek, die beispielsweise auf locales, Fliesskommabearbeitung und das Buffern der Ausgabe verzichtet. Dafür wird versucht, Abhängigkeiten zwischen Bibliotheksfunktionen zu minimieren. Trotzdem existierende Abhängigkeiten werden so dokumentiert, dass es ein Einfaches ist, einzelne Funktionen auszutauschen um die Bibliothek an jede erdenkliche Umgebung anpassen zu können.

2.1.5 Gerätetreiber

Die Entwicklung von Gerätetreibern stellt wohl eine der umfangreichsten Aufgaben bei der Betriebssystementwicklung dar, bedenkt man die schier endlose Vielfalt an vorhandener Hardware. Vor allem bei neu erschienener Hardware ist es meist ein sehr aufwendiges Unterfangen, spezielle (meist undokumentierte) Eigenheiten zu erfassen und in Software richtig anzusteuern. Um hier das Rad nicht mehrfach neu erfinden zu müssen bedient sich OSKit der Treiber von Linux und FreeBSD. Anstelle diese anzupassen – und damit Inkompatiblitäten zwischen Original und modifiziertem Treiber heraufzubeschwören, die sich beispielsweise bei Updates sehr negativ auswirken würden – verwendet OSKit eine spezielle Kapselung durch sogenannten „glue code“, durch die die existierenden Treibersourcen grösstenteils unmodifiziert übernommen werden können. Durch die umsichtige Kapselung ist es ausserdem problemlos möglich, Linux-, FreeBSD- und möglicherweise in der Zukunft auch Treiber anderer, proprietärer Systeme nebeneinander zu verwenden.

2.1.6 Netzwerkstack und Dateisysteme

Das OSKit enthält einen umfassenden TCP/IP-Protokollstack. Wie auch schon bei den Gerätetreibern wurde hier auf eine Eigenentwicklung verzichtet, sondern Code anderer freier Betriebssysteme durch entsprechende Kapselung eingebunden. Die Treiber für Netzwerkkarten stammen aus Linux, während die eigentlichen

Komponenten aus FreeBSD stammen.

Die Dateisystemunterstützung stammt ebenso aus einem freien Betriebssystem, nämlich NetBSD. Auch hier wurden die originalen Treiber gekapselt, um eine einfache Wartung zu ermöglichen.

2.2 Design und Implementierung

Das Design des OSKit ist stark durch den Gedanken an weitestgehende Flexibilität geprägt. Einige daraus resultierende – für Kernelentwicklung teils auch ungewohnte – Designentscheidungen sollen im folgenden beispielhaft illustriert werden.

2.2.1 Bibliotheken

Ein wichtiges Ziel von OSKit ist es, möglichst einfach und komfortabel verwendbar zu sein. Um das Einbinden von OSKit-Code so einfach wie möglich zu gestalten, wurde OSKit als eine Menge von C-Bibliotheken realisiert, die in die normalen Bibliotheksverzeichnisse des Entwicklungssystems installiert werden und danach – wie von Applikationsprogrammen gewohnt – einfach zum entwickelten Kern dazugelinkt werden können.

Diese Bibliotheken sind in dem Sinne unabhängig, dass sie von keinerlei anderen Bibliotheken auf dem Entwicklungssystem abhängen. Für den jeweiligen Entwickler ist es dadurch erheblich einfacher, eine Bibliothek zu einem Projekt zu linken, als etwaige einzelne Komponenten in mühevoller Kleinarbeit hin- und herzukopieren.

2.2.2 “Separability” – Separierbarkeit

Wichtigste Eigenschaft, sowohl bei normaler Softwareentwicklung, aber gerade bei einer Komponentensammlung, ist natürlich Modularität. OSKit führt diesen Gedanken aber noch weiter: das System soll nicht nur modular aufgebaut sein, sondern die einzelnen Module untereinander sollen beliebig austauschbar oder ersetzbar sein, so dass ein Entwickler einzelne Teile des OSKit verwenden kann, ohne dadurch weitere OSKit-Komponenten verwenden zu müssen. Hier findet sich das Prinzip der „Trennung der Belange“ wieder. Beispielsweise sollen die OSKit Netzwerkkomponenten auch ohne den OSKit Memory Manager verwendbar sein – obwohl die Netzwerkkomponenten natürlich irgendeinen Memory Manager brauchen.

Um diese Einzelverwendbarkeit von Komponenten zu bewerkstelligen bedient sich OSKit verschiedener Schichten von „glue code“, also Abstraktionsebenen zwischen einzelnen Komponenten. Hierbei kommen zwei verschiedene Techniken zum Einsatz: überschreibbare Funktionen sowie dynamisches Binden.

2.2.2.1 Überschreiben von Funktionen

Das Überschreiben von Funktionen wird überall dort genutzt, wo es systemweit nur eine einzige Implementation der entsprechenden Funktionalität gibt, Beispiele wären `malloc` und `putchar`. Alle Gerätetreiberkomponenten im OSKit verwenden zum Beispiel eine Funktion namens `fdev_mem_alloc` zum Allokieren von Speicher. Hierzu gibt es natürlich auch eine Implementierung in den OSKit-Management-

Routinen, die aber bei Bedarf überschrieben werden kann.

2.2.2.2 Dynamisches Binden

Dynamisches Binden bedeutet in erster Linie den Gebrauch von Funktionszeigern und Funktionstabellen. Die Verwendung dieser macht dort Sinn, wo mehrere Implementierungen derselben Funktionalität nebeneinander existieren müssen. Ein Beispiel wären hier verschiedene Blockgerätetreiber, die von den Dateisystemkomponenten verwendet werden. Der jeweilige Blockgerätetreiber gibt bei seiner Initialisierung eine Schnittstelle zurück – im Prinzip eine Tabelle mit Funktionszeigern. Diese Schnittstelle wird danach einfach an die Dateisystemkomponente weitergegeben; so können auch zur Laufzeit verschiedene Dateisystemkomponenten an verschiedene Blockgeräte gekoppelt werden, ohne dass die eine Komponente von den Eigenheiten der anderen weiss.

2.2.3 COM Interfaces

Wichtig für die Funktion eines jeden Baukastens ist es, eine eindeutige, saubere und einfach verwendbare Möglichkeit der Schnittstellenbeschreibung zur Verfügung zu haben. Dies löst OSKit durch die Verwendung einer Teilmenge des *Component Object Model*, die sich im Endeffekt am einfachsten als sprachunabhängiges Protokoll zur Schnittstellenbeschreibung für Komponenten innerhalb desselben Adressraumes beschreiben lässt. Das COM hat, abgesehen von konsistenten und standardisierten Schnittstellen, noch weitere implementationsspezifische Vorteile, auf die im weiteren näher eingegangen werden soll.

2.2.3.1 „Implementation Hiding“

„Implementation Hiding“ meint in erster Linie eine Trennung von Schnittstelle und Implementierungen, so dass erstere von letzteren vollkommen unabhängig ist. COM-Schnittstellen werden unabhängig von den zugrundeliegenden Komponenten entworfen, so dass in der Praxis auch durchaus mehrere Implementierungen einer einzelnen Schnittstelle koexistieren können; man erinnere sich an das Beispiel der Dateisystemkomponente, die auf verschiedene Blockgeräte zugreift.

In C wird eine COM Schnittstelle üblicherweise als opake Struktur realisiert, deren eigentlicher Inhalt dem Aufrufer unbekannt ist, bis auf die Tatsache, dass der erste Eintrag einen Zeiger auf eine Funktionstabelle darstellt. Abbildung 3 zeigt dies anhand der `blkio`-, also der Blockgeräte-Schnittstelle des OSKit.

```
typedef struct blkio {
  struct blkio_ops *ops;
} blkio_t;

struct blkio_ops {
  error_t  (*query)(blkio_t *io,
                    const struct guid *iid,
                    void **out_ihandle);
  unsigned (*addref)(blkio_t *io);
  unsigned (*release)(blkio_t *io);
  unsigned (*getblocksize)(blkio_t *io);
  error_t  (*read)(blkio_t *io, void *buf,
                   off_t offset, size_t amount,
                   size_t *out_actual);
  error_t  (*write)(blkio_t *io, const void *buf,
                    off_t offset, size_t amount,
                    size_t *out_actual);
  error_t  (*getsize)(blkio_t *io, off_t *out_size);
  error_t  (*setsize)(blkio_t *io, off_t new_size);
};

/* Friendly macros */
#define oskit_blkio_read(io, buf, ofs, amount, out_actual) \
                        ((io)->ops->read((io),
                        (buf), (ofs), (amount), (out_actual)))
....

#define BLKIO_IID GUID(0x4aa7df81, 0x7c74, 0x11cf, \
    0xb5, 0x00, 0x08, 0x00, 0x09, 0x53, 0xad, 0xc2)
```

Abb. 3: Ausschnitt aus der OSKit COM-Schnittstelle für Blockgeräte [3]

Zu sehen sind hier sowohl die Schnittstellenstruktur namens `blkio` als auch die Funktionstabelle, die als `struct blkio_ops` realisiert ist. Zusätzlich sieht man in den letzten beiden Zeilen die Definition einer Schnittstellen-ID, die einen *Globally Unique Identifier* darstellt, anhand dessen die Schnittstelle eindeutig identifiziert werden kann.

2.2.3.2 Schnittstellenerweiterung

Eine weitere Eigenschaft des COM-Modells ist die Möglichkeit, mehrere „Sichten“ auf ein und dasselbe Objekt zu definieren. So können verschiedene Schnittstellen unterschiedliche Teile der Implementierung einer Komponente freigeben. Die einzelnen Schnittstellen werden hier anhand von *DCE Universally Unique Identifiers*, grob gesagt weltweit eindeutigen IDs, die mittels eines speziellen Algorithmus ermittelt werden, unterschieden. Besitzt man einen Zeiger auf eine Schnittstelle einer Komponente, so kann man anhand dieses Zeigers nach weiteren vorhandenen Schnittstellen suchen, eine Technik, die als „narrowing“ oder „safe downcasting“ bekannt ist.

Dieser Mechanismus erlaubt es, für einige Komponenten erweiterte Schnittstellen zur Verfügung zu stellen, ohne Kompatiblität zu den Aufrufern zu verlieren, die nur mit den ursprünglich angebotenen Schnittstellen zurechtkommen.

Ein Beispiel ist die `bufio`-Schnittstelle des OSKit, die auf der vorhin angesprochenen `blkio`-Schnittstelle basiert. Komponenten, die erstere Schnittstelle

anbieten, können mittels dieser über einen Ein-/Ausgabepuffer verfügen, können aber auch durch die `blkio`-Schnittstelle wie ein ungepuffertes Gerät angesprochen werden.

Eine weitere Anwendung ist die Umsetzung einer „Open Implementation"-Philosophie – abgesehen von den abstrakten, implementationsunabhängigen Schnittstellen bieten viele Komponenten auch Schnittstellen auf niedrigerer Ebene an, die teilweise implementationsspezifische Funktionen enthalten, um einige Fähigkeiten der Implementierung – wenn gewünscht – besser ausnutzen zu können.

2.2.3.3 Infrastruktur-Unabhängigkeit

Ein besonderer Vorteil des OSKit-COM-Ansatzes ist es, dass sämtliche Schnittstellen für sich allein funktionieren und keinerlei Infrastrukturcode oder ähnliches benötigen. Man kann also bei der Verwendung von reinen OSKit-Komponenten auf jeglichen „glue code" verzichten, den man bei Verwendung von Komponenten aus anderen Betriebssystemen benötigt – ein Beispiel hierfür sind die von OSKit gekapselten Linux- und FreeBSD-Treiber und Netzwerkstacks, die jeweils eine bestimmte Pufferimplementierung voraussetzen.

2.3 OSKit in der Praxis

Der beste Baukasten ist nutzlos, wenn sich die daraus entstehenden Komponenten nicht in der Praxis bewähren. Um einerseits die Effizienz der Implementierung zu demonstrieren, andererseits auch die nötige Praxistauglichkeit zu belegen, soll im folgenden kurz auf einen Benchmark-Kern sowie auf mehrere mittels OSKit realisierte Projekte eingegangen werden.

2.3.1 Benchmarks

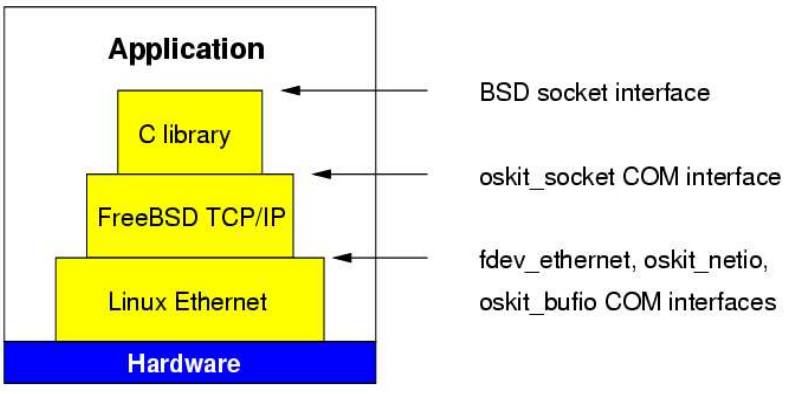

Abb. 4: Struktur der `ttcp` und `rtcp`-Testkerne [3]

Um einen Überblick über Performanceverluste, die durch Kapselung und „glue code" entstehen, zu erhalten, wurden vom OSKit-Team zwei Testkernels entwickelt, die auf den Programmen `ttcp` und `rtcp` basieren (ersteres misst die Bandbreite, zweites die Latenz bei TCP-Verbindungen). Diese Testkernels wurden dann mit Linux- und

FreeBSD-Systemen verglichen. Bei letzteren Systemen wurden die Sende- und Empfangsschleifen des Testprogramms direkt in den Kern eingebaut, um einen Overhead an Systemaufrufen zu verhindern. Dieser Overhead würde die erhaltenen Werte stark nach unten beeinflussen, würde das Testprogramm als normale Applikation laufen. Abbildung 4 zeigt die Struktur der OSKit-Testkerne; die Ergebnisse des Tests finden sich in Tabelle 1 und Tabelle 2.

	Receiver:		
	Linux	FreeBSD	OSKit
Sender:			
Linux	**72.4**	71.2	71.3
FreeBSD	60.0	**78.6**	78.7
OSKit	56.4	68.3	**68.2**

Tabelle 1: TCP Bandbreite in Mbit/s [3]

	Server:		
	Linux	FreeBSD	OSKit
Client:			
Linux	**152**	168	180
FreeBSD	168	**197**	210
OSKit	180	210	**222**

Tabelle 2: TCP Umlaufzeit in µs [3]

An beiden Tests sieht man die Eigenschaften des „glue codes“, der sowohl den Linux Ethernet-Treiber als auch den FreeBSD TCP/IP-Stack in den entsprechenden OSKit Komponenten kapselt. Hierzu muss man wissen, dass die FreeBSD-Implementierung eine Pufferkomponente namens `mbuf` voraussetzt, die auch unzusammenhängende Speicherbereiche verwenden kann, während der Linux-Netzwerktreiber eine Pufferimplementierung namens `skbuff` verwendet, die nur zusammenhängende Pufferbereiche verwalten kann. Sobald also ein von der FreeBSD-Komponente als unzusammenhängend erstellter Puffer an die Linux-Komponente weitergegeben werden soll, erfordert dies zusätzliche Kopieraktionen im „glue code“. Dies schlägt sich deutlich sichtbar bei der Bandbreitenmessung nieder: während die OSKit-Empfängerseite wie zu erwarten ähnliche Werte wie der originale FreeBSD-Empfänger erreicht, bricht die Leistung bei Verwendung von OSKit als Sender deutlich ein. Die Messung der Latenz hingegen zeigt sowohl für den OSKit-Sender als auch für den OSKit-Empfänger Performanceverluste, die hier allgemein auf den Overhead durch „glue code“ zurückzuführen sind.

2.3.2 Fallstudien

Abgesehen von den Testkerneln gibt es mehrere Praxisbeispiele, an denen OSKit „erprobt“ wurde*[3]*. Zwei davon sollen nun beispielhaft aufgeführt werden, sowie eine Sammlung von weiteren Projekten, die OSKit einsetzen.

2.3.2.1 Standard ML

Standard ML ist eine funktionale Programmiersprache, die ein sehr exotisches Laufzeitmodell besitzt. Die dort verwendete Modellierung von Nebenläufigkeiten mit speziellen Semantiken erfordert eine starke Beeinflussung des Kontextwechselcodes. Diese Möglichkeiten bieten normale Betriebssysteme nicht, weswegen der Versuch unternommen wurde, ein eigenes Betriebssystem namens ML/OS zu entwickeln. Die Entwicklung unter Verwendung von OSKit dauerte am MIT insgesamt ein halbes Jahr, das Entwicklungsteam bestand aus einem Diplomanden, der zeitweise von einer wissenschaftlichen Hilfskraft unterstützt wurde – beide hatten vor Beginn des

Projektes keinerlei Erfahrung mit Betriebssystemprogrammierung.

Andere Projekte, bestehend aus grossen Gruppen von erfahrenen Wissenschaftlern und Programmierern, versuchten jahrelang, dasselbe Ziel zu erreichen – ohne die Verwendung eines Betriebssystembaukastens – allerdings erfolglos.

2.3.2.2 Java/OS

Ein weiteres Projekt der Flux Research Group war die Entwicklung eines Java-basierenden Betriebssystems. Hierzu wurde die freie virtuelle Maschine *Kaffe* mittels OSKit zu einem Betriebssystem „umgerüstet". Die Entwicklung übernahm ein einzelner Student, dessen System nach 14 Stunden in der Lage war, eine „hello world"-Java-Applikation laufen zu lassen; nach drei Wochen Entwicklungszeit entsprach das System vom Funktionsumfang her *Sun*'s *JavaOS* und unterstützte komplexeste Anwendungen.

2.3.2.3 Weitere Projekte

Eine grosse Reihe weiterer Projekte nutzt mittlerweile die Vorzüge des OSKit – zuviele, um hier auf jedes einzelne näher einzugehen. Daher folgt hier nur eine kleine Auswahl, ohne besondere Reihenfolge: das *Mungi* Projekt von der University of New South Wales in Australien, der GNU Network Bootloader namens *NILO*, mehrere Portierungen des L4-Mikrokernels, das *Fiasco* Projekt der Technischen Universität Dresden, diverse Projekte der Network Storage Corporation, das *Exokernel*-Projekt des MIT, eine Scheme-Implementation namens *MzScheme* von der Rice University (Entwicklungszeit ganze fünf Stunden) sowie ein weiteres Java-Betriebssystem namens *Janos*.

2.4 Bewertung

Das erklärte Zeil von OSKit war es, eine Grundlage zum einfachen Entwickeln dedizierter Betriebssystemkerne, hauptsächlich im Bereich von Forschung und Lehre, zur Verfügung zu stellen. Dieses Ziel wurde erreicht, die genannten Beispielprojekte sowie die grosse, auch internationale Verbreitung von OSKit sind wohl der beste Beweis dafür.

Allerdings gibt es auch für OSKit natürlich Erweiterungsmöglichkeiten – einen umfassenden, für jeglichen erdenklichen Zweck geeigneten Baukasten stellt es definitiv nicht dar. Ein Problem liegt beispielsweise in der Granularität der einzelnen Komponenten: gerade die gekapselten Treiberkomponenten aus anderen Systemen sind eigentlich eine „alles oder nichts"-Lösung, bei der man nur wenig Einflussmöglichkeiten auf interne Verhaltensweisen hat.

Trotzdem ist die vollzogene „Trennung der Belange" ein wegweisender Schritt in die richtige Richtung, der zum Beispiel vom im folgenden besprochenen „THINK" Projekt aufgegriffen und erweitert wurde.

3 THINK

Der zweite betrachtete Betriebssystembaukasten nennt sich THINK, ein rekursives Akronym für „Think Is Not a Kernel“. Er ist um knapp fünf Jahre jünger als OSKit und hat daher einige Gemeinsamkeiten, die im weiteren Verlauf noch sichtbar werden.

THINK selbst beinhaltet nur ein Framework zum komponentenbasierten Bau von Betriebssystemkernen, das zum Ziel hat, Komponenten verschiedenster Größe durch ein flexibles Bindungsmodell verknüpfen zu können. Hierbei soll sowohl eine statische Bindung als auch eine Bindung zur Laufzeit möglich sein. Ausgemachtes Ziel von THINK sind eingebettete Systeme, bei denen einerseits meist sehr spezielle Anwendungsszenarien vorliegen, andererseits sowohl CPU als auch Speicher stark begrenzt sind.

Das Framework wird begleitet von einer Referenzimplementierung, eine Komponentenbibliothek namens KORTEX. Im weiteren sollen sowohl THINK als auch KORTEX näher vorgestellt werden.

3.1 THINK – das Framework

3.1.1 Konzepte

Das THINK Framework baut auf einer Reihe von Konzepten auf, namentlich Domänen, Komponenten, Schnittstellen, Bindungen und Namen. Eine *Domäne* entspricht einer Ressourcen-, Sicherheits- oder Isolationsgrenze; beispielsweise entsprechen privilegierter Kernelcode einerseits und unprivilegierter Anwendungscode andererseits verschiedenen Domänen. Eine Domäne enthält eine Menge von *Komponenten*, Komponenten kommunizieren untereinander durch *Bindungen*. Hierbei können sowohl Domänen als auch Bindungen wieder als Komponenten aufgefasst werden – ein Beispiel wäre eine RPC-Bindung, die ihrerseits aus Netzwerkkomponenten besteht. Aus diesem Beispiel ist auch ersichtlich, dass eine Bindung Domänengrenzen überschreiten kann.

Eine Komponente implementiert ein oder mehrere *Schnittstellen*. Schnittstellen werden in der Sprache Java definiert; durch deren Typsicherheit gewinnt man eine erste Sicherheitsstufe beim Verbinden von Komponenten: Eine Bindung zwischen Komponenten kann nur erstellt werden, wenn beide Schnittstellen typkompatibel sind. Schnittstellen werden durch *Namen* angesprochen. Diese Namen sind kontext-abhängig, das heisst, sie gelten nur innerhalb eines bestimmten *Namenskontext*.

```
interface Top { }
interface Name {
        NamingContext getNC();
        String toByte();
}
interface NamingContext {
        Name export(Top itf, char[] hint);
        Name byteToName(String name);
}
interface BindingFactory {
        Top bind(Name name, char[] hint);
}
```

Abb. 5: Framework für Interfaces, Namen und Bindungen [5]

Weitere Zusammenhänge zeigt Abbildung 5: Alle Schnittstellentypen sind Erben des Typs `Top`. Der Typ `Name` ist der übergeordnete Typ aller Namen. Die Methode `getNC` liefert den zum jeweiligen Namen gehörenden Namenskontext, die Methode `toByte` liefert eine serialisierte Form des Namens. Der Typ `NamingContext` ist der Vatertyp aller Namenskontexte, die Methode `export` generiert einen neuen Namen für die übergebene Schnittstelle, die Methode `byteToName` gibt zu einem gegebenen serialisierten Namen ein Namensobjekt zurück. Vom Typ `BindingFactory` erben alle *binding factories*, Einrichtungen zum expliziten Erstellen von Bindungen. Die Methode `bind` erstellt eine Bindung zu der zum übergebenen Namen gehörenden Schnittstelle, der Parameter `hint` kann hierbei verwendet werden, um weitere Informationen zur Bindung festzulegen, etwa zur Dienstgüte.

3.1.2 Umsetzung

THINK liesse sich natürlich in verschiedenen Sprachen implementieren. Die Referenzimplementierung verwendet jedoch C aus Gründen der Effizienz. Schnittstellen werden hier genauso wie auch in OSKit realisiert: durch eine Beschreibungsstruktur, die einen Zeiger auf eine Funktionstabelle enthält (siehe Abbildung 6).

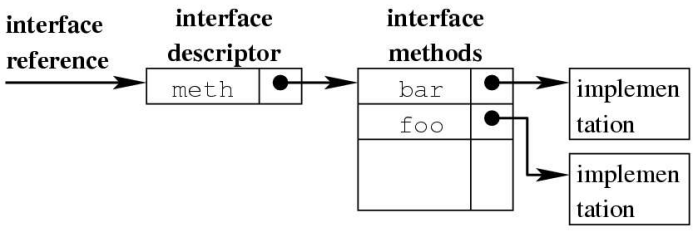

Abb. 6: Realisierung von Schnittstellen zur Laufzeit [5]

Hierzu wurden bei der Implementierung von KORTEX verschiedene

Optimierungsmethoden angewandt, die sich auf den Speicherplatz der Objektdaten beziehen. Je nach Art der Komponente lassen sich hierdurch Kosten für das Allokieren von Speicher einsparen (siehe Abbildung 7). Liegt ein *Singleton* vor, das heisst, es gibt keine andere Komponente in derselben Domäne, die dieselbe Schnittstelle implementiert, so kann man den Speicherplatz für Objektdaten statisch vom Compiler allokieren lassen. Ist die fragliche Komponente kein Singleton, aber hat nur genau eine Schnittstelle, die sie implementiert, so bietet es sich an, die Objektdaten in der Schnittstellen-Beschreibungsstrukur unterzubringen. Allen anderen Fällen werden so gelöst, dass in der Schnittstellenbeschreibung sowohl ein Zeiger auf die Funktionstabelle als auch ein Zeiger auf die Objektdaten ihren Platz finden. Da die KORTEX-Bibliothek zu einem Grossteil aus Singletons bzw. Komponenten mit nur einem Schnittstellen besteht, lässt sich hierdurch im Vergleich zur generischen Implementierung einige Performanz gewinnen.

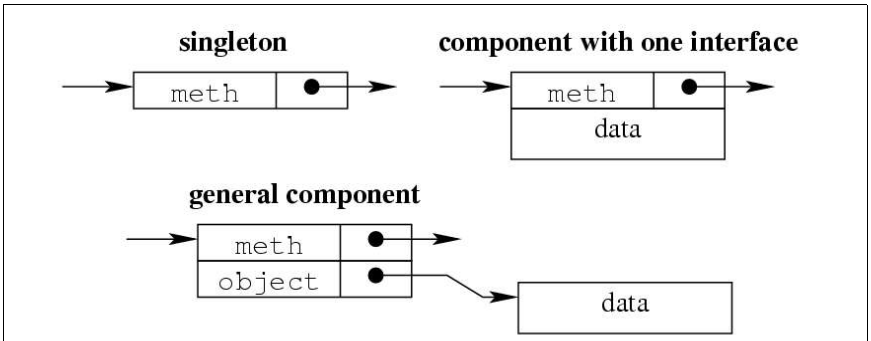

Abb. 7: Optimierungsmöglichkeiten der Schnittstellenrealisierung [5]

3.1.3 Codegenerierung und Werkzeuge

Um mittels des THINK Frameworks zu einem funktionierenden Betriebssystem zu kommen bedarf es noch zweier spezieller Werkzeuge: Einem *Interface Compiler* und einem sogenannten *Off-Line Configurator*.

Aufgabe des Interface Compilers ist es, aus den Java-Schnittstellenbeschreibungen Code zu erzeugen, seien es Deklarationen und Code für die Interfacebeschreibungen oder auch Komponenten-Stubs für die verschiedenen Bindungen. Wenn möglich kann dieser Interface Compiler auch Assemblercode generieren, um hardwarespezifische Features voll auszunutzen.

Der Off-Line Configurator erstellt den eigentlichen Kernel aus einem Komponentengraphen, der in UML erstellt wird. Hierbei wird auf Abhängigkeiten zwischen einzelnen Komponenten eingegangen und die Initialisierungsreihenfolge der einzelnen Komponenten festgelegt; ausserdem enthält der Configurator ein graphisches Werkzeug zum Betrachten des Komponentengraphen.

Die typische Codegenerierung bei einem THINK-basierenden System beinhaltet also zuerst das Kompilieren der Schnittstellen mit dem Interface Compiler und danach das eigentliche Assemblieren des Kerns durch den Configurator. Während der Laufzeit kann ein solcher Kern dann weitere Komponenten oder Applikationen über spezielle Komponenten- bzw. Applikationsloader nachladen.

3.2 KORTEX – die Referenzimplementierung

Um die Vorzüge von THINK verwenden zu können bedarf es natürlich auch einer Komponentenbibliothek, die den THINK-Konzepten folgt; KORTEX stellt eine solche Referenzimplementierung für Apples *PowerPC*-Plattform dar. Im Gegensatz zu OSKit wurde – mit Ausnahme der Gerätetreiber – keinerlei Code aus anderen Projekten verwendet, sondern sämtliche Komponenten neu implementiert. Beispiele aus der grossen Auswahl an Komponenten:

- Komponenten zur Hardware-Abstraktion (HAL),
- Speicherverwaltung,
- Scheduler- und Threadkomponenten,
- Netzwerkkomponenten für verschiedenste Protokolle (Ethernet, ARP, IP, UDP, TCP und SunRPC),
- Dateisystemkomponenten für *ext2* und *NFS*,
- Service-Komponenten wie ein dynamische Linker oder der schon erwähnte Applikationsloader,
- Interaktions-Komponenten, die eine Vielzahl an Bindungsmöglichkeiten bereitstellen,
- sowie Komponenten, die Teile des POSIX-Standards realisieren.

Zusätzlich stellt KORTEX noch Frameworks zum Ressourcenmanagement und zur Kommunikation bereit. Einige dieser Komponenten sollen nun beispielhaft angesprochen werden.

3.2.1 Beispiel einer HAL-Komponente

Als Beispiel einer Komponente zur Hardwareabstraktion soll hier die Komponente zum Exception-Handling genauer untersucht werden. Abbildung 8 zeigt das zu ihr gehörende Interface.

```
interface Trap {
        void TrapRegister(int id, Handler handler);
        void TrapUnregister(int id);
        void TrapSetContext(int phyctx, Context virtctx);
        Context TrapGetContext();
        void TrapReturn();
}
```

Abb. 8: Interface der PowerPC-Exception-Handling-Komponente [5]

Hierbei dient die Methode `TrapRegister` dazu, einen `Handler` für die Exception `id` zu registrieren. Tritt eine Exception auf, so ruft der Prozessor eine interne Methode namens `TrapEnter` auf, die die generischen Register sichert, einen Stack für den Exception-Handler anlegt und danach den entsprechenden Handler aufruft. Wenn dieser seine Arbeit beendet hat, ruft er die Methode `TrapReturn` auf, die die gesicherte Aufrufumgebung wieder restauriert. Die Methode `TrapSetContext` dient zum Setzen der Speicheradresse, die für das Zwischenspeichern der Aufrufumgebung verwendet wird.

3.2.2 Ressourcenmanagement

Ein weiterer Teil der KORTEX Bibliothek enspricht weniger einer Komponente als einem eigenständigen Framework zur Ressourcenverwaltung. Die verwendeten Konzepte von Ressource und Ressourcenmanager werden aus Abbildung 9 ersichtlich.

```
interface AbstractResource {
        void release();
}
interface ResourceManager extends BindingFactory {
        AbstractResource create(...);
}
```

Abb. 9: Das Ressourcenmanagement-Framework [5]

Neue Ressourcen (z.B. Threads) können hier über die `create`-Methode kreiert werden, zugewiesen werden sie dann durch die `bind`-Methode, die eine Bindung zu der entsprechenden Ressource erstellt. Dieses Framework wird bei mehreren anderen KORTEX-Komponenten verwendet: Threads und die sie verwaltenden Scheduler, Speicher und Speicherverwaltung, Netzwerk-Sessions und die sie „verwaltenden" Protokolle.

3.2.3 Threads/Scheduler

Die KORTEX Bibliothek stellt drei verschiedene, verdrängende Scheduler-Komponenten zur Verfügung: einen kooperativen Scheduler, einen einfachen Round-Robin-Scheduler sowie eine prioritätsbasierende Schedulerkomponente. Je nach Verwendung (ob als Prozessscheduler oder als Threadscheduler) führt die jeweilige Schedulerkomponente zusätzlich zum Kontextwechsel auch einen eventuell nötigen Wechsel des Adressbereiches durch.

Alle Scheduler verwenden die PowerPC HAL-Exception-Komponente, durch deren Einfachheit sie sehr effizient arbeiten können: Auf einem PowerPC G4 mit 500 Mhz dauert ein Prozesswechsel 0.394 µs oder 147 Instruktionen, ein Threadwechsel nur 0.284 µs oder 111 Instruktionen. Hierdurch können sehr enge Zeitschlitze für das Scheduling verwendet werden, was die Scheduler-Komponenten beispielsweise auch für Echtzeitsysteme interessant macht.

3.2.4 Interaktions-Komponenten

Insgesamt stellt die KORTEX-Bibliothek vier unterschiedliche Bindungstypen zur Verfügung, die im folgenden kurz erklärt werden sollen.

3.2.4.1 Lokale Bindungen

Eine lokale Bindung bedeutet nichts weiter als einen einfachen Zeiger auf ein Interface. Sie findet Verwendung zwischen Komponenten aus derselben Domain.

3.2.4.2 Syscall-Bindung

Die Syscall-Bindung findet bei Systemen Verwendung, die verschiedene Adressbereiche für Kernel und Applikationen vorsehen. Sie entspricht der Kommunikation zwischen einer Applikation und dem Kern. Die Applikation erhält eine Stub-Komponente, die bei Aufruf eine Exception aufruft; der Exception-Handler des Kernels ruft dann die eigentliche Zielkomponente auf.

3.2.4.3 „Upcalls" / Signale

Durch sogenannte *Upcalls* oder Signale teilt der Kern den Applikationen Ereignisse mit. Eine Signal-Bindung steht hier für Interaktion mit der aktuell laufenden Applikation, während ein Upcall für Interaktion mit einer Applikation in einem anderen Adressbereich verwendet wird. Beide bedeuten eigentlich nur den Aufruf einer speziellen Handler-Methode in der jeweiligen Applikation.

3.2.4.4 Synchrones LRPC

LRPC, „Lightweight RPC", kann von Applikationen verwendet werden, um zwischeneinander Nachrichten auszutauschen. Effektiv besteht es aus einer Syscall-Bindung, durch die eine Upcall-Bindung aufgerufen wird.

3.2.4.5 Remote RPC

Eine Möglichkeit rechnerübergreifender Bindungen stellt die RPC-Bindung dar. Sie erstellt und versendet Ethernet-Pakete über die verschiedenen Netzwerkkomponenten und ist hauptsächlich zur Kommunikation zwischen Kernels gedacht. Man kann sie aber durch Verbindung mit Syscall- und Upcall-Bindungen auch zur Kommunikation zwischen verteilten Anwendungen verwenden.

3.3 Praxisbeispiele

Um die praktische Verwendbarkeit von THINK und Kortex zu demonstrieren, wurden von den Entwicklern mehrere experimentelle Betriebssystemkerne entwickelt und zur Leistungsmessung herangezogen. Diese – und die erbrachten Ergebnisse – sollen im weiteren kurz untersucht werden.

3.3.1 Ein erweiterbarer, verteilter Mikrokernel

Zuerst wurde ein minimaler Mikrokernel implementiert, dessen schematischer Aufbau in Abbildung 10 wiedergegeben ist. Mit diesem wurden Zeitmessungen für die verfügbaren Bindungstypen durchgeführt. Tabelle 3 zeigt die Ergebnisse für lokale Bindungen, Tabelle 4 für RPC-Kommunikation. Vergleicht man diese Ergebnisse mit Referenzwerten, so bewegen sie sich durchaus im selben Rahmen. Dies zeigt, dass die Verwendung des THINK Frameworks keine merkliche Verschlechterung der Performance bedeutet.

Interaction	instructions	time(μs)	cycles
local	6	0.016	8
syscall	115	0.300	150
optimized syscall	50	0.162	81
signal	35	0.128	64
upcall	107	0.346	173
LRPC	217	0.630	315
optimized LRPC	152	0.490	245

Tabelle 3: Performance der KORTEX Bindungen [5]

Network type	Time (μs) total	network link	network driver	marshall. +null call
10baseT	180	164.7 (91.5%)	11.3 (6.3%)	4 (2.2%)
100baseT	40	24.7 (61.7%)	11.3 (28.3%)	4 (10%)

Tabelle 4: Performance der RPC-Bindung [5]

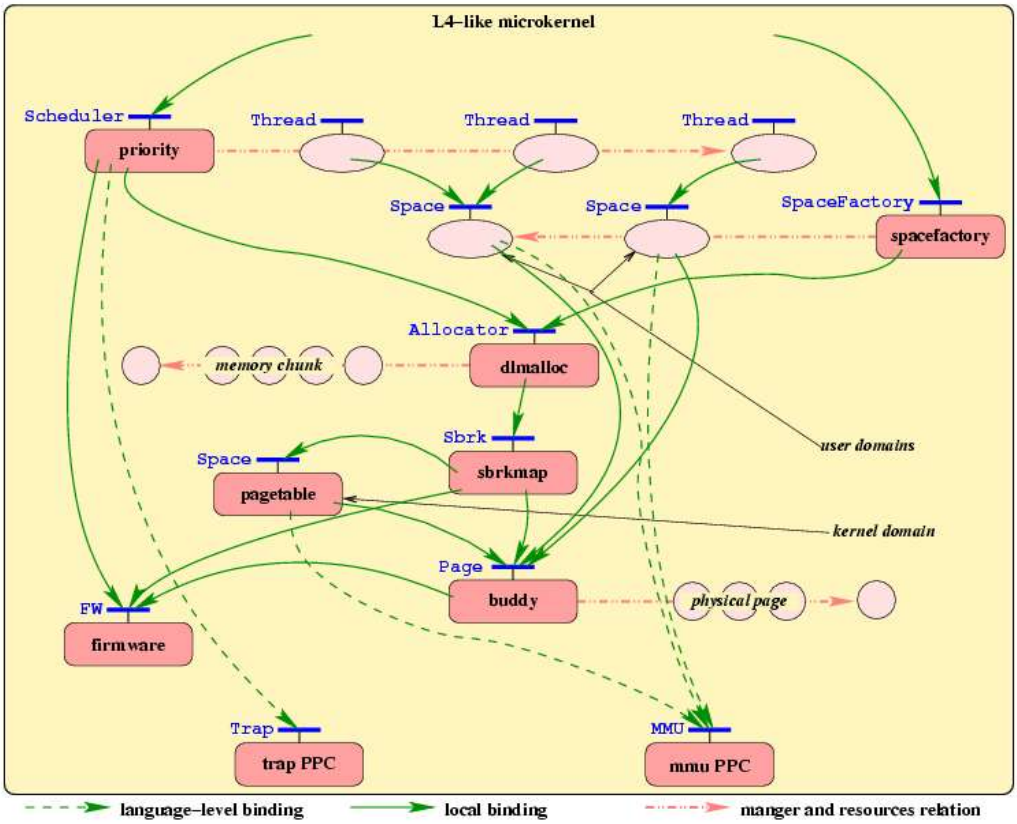

Abb. 10: Konfigurationsgraph eines Beispielkernels [5]

3.3.2 Weitere Experimente

Mehrere weitere Experimente zur Performanzmessung wurden durchgeführt, deren Aussage allerdings wenig überrascht. So wurde einerseits ein spezialisierter Kern für die Sprache PlanP entwickelt – eine Sprache mit JIT-Compiler, die zur Beschreibung aktiver Netzwerkkomponenten wie Router und Switches verwendet wird – und der Datendurchsatz eines auf diesem Kernel laufenden PlanP-Programmes, das eine Netzwerkbridge implementiert, gemessen. Dieser Durchsatz ist erwartungsgemäß höher als der Durchsatz desselben Programmes, das in einem PlanP-Interpreter sowohl unter Solaris als auch unter Linux läuft, obwohl der Interpreter auch auf letzteren Systemen im Supervisor-Modus, d.h. im Kernel läuft. Die Ergebnisse zeigt Tabelle 5.

bridge	throughput
none	91.6Mbps
PlanP/Solaris, Sparc 166Mhz	42.0Mbps
PlanP/Linux, PowerPC 350Mhz	65.5Mbps
PlanP/KORTEX, PowerPC 350Mhz	87.6Mbps

Tabelle 5: Performance des PlanP-Kernels [5]

Weitere Tests wurden mit einer *Kaffe*-Portierung auf einen dedizierten THINK-Kern durchgeführt, sowohl unter Implementierung der Java-Thread-Semantik, als auch unter Verwendung der Thread-Semantik der KORTEX-Schedulerkomponente. Auch hier ergibt sich für den dedizierten Kern ein Performanzgewinn (siehe Tabelle 6).

Benchmark	Kaffe/Linux (java-thread)	Kaffe/KORTEX (java-thread)	Kaffe/KORTEX (native-thread)
synchronized(o) {}	0,527 μs	0,363 μs	0,363 μs
try {} catch(...) {}	1,790 μs	1,585 μs	1,594 μs
try {null.x()} catch(...) {}	12,031 μs	5,094 μs	5,059 μs
try {throw} catch(...) {}	3,441 μs	2,448 μs	2,434 μs
Thread.yield()	6,960 μs	6,042 μs	6,258 μs

Tabelle 6: Performance der Kaffe-Implementierung [5]

Das letzte Experiment besteht aus einer Portierung des Videospiels *Doom*. Hier wurde die Linux-Version *LxDoom* in einen dedizierten Kern umgewandelt, einmal unter Benutzung der MMU, einmal mit einem „flachen“ Speichermodell. Auch hier ergibt sich eine – zu erwartende – Performanzverbesserung bei der Verwendung eines dedizierten Kerns (siehe Tabelle 7).

	external resolution		
	320x200	640x480	1024x768
KORTEX(flat)	1955	491	177
KORTEX(MMU)	1914	485	171
Linux	1894	483	167

Tabelle 7: Performance des dedizierten Doom-Kernels [5]

3.4 Bewertung

Das THINK-Framework stellt eigentlich eine Erweiterung des OSKit-Konzeptes dar; nicht nur einzelne Komponenten werden gekapselt, sondern auch Bindungen, also Beziehungen der Komponenten untereinander. Die dazu durchgeführten Geschwindigkeitsmessungen zeigen, dass diese zusätzliche Abstraktionsebene keine nennenswerten Einbußen hinsichtlich der Performanz bedeutet.

Diese Möglichkeiten machen das THINK Framework durchaus für verschiedenste Projekte interessant. Hauptaugenmerk der Entwickler lag auf dem Bereich der eingebetteten Systeme – inwieweit es sich hier durchsetzt wird die Zukunft zeigen. Die verwendeten Konzepte und Ansätze sind jedenfalls, wenn auch eventuell noch erweiterbar, eine gute Grundlage.

4 Zusammenfassung, Ausblick

Beide Betriebssystembaukästen bieten die Möglichkeit, mit minimalem Aufwand ein dediziertes Betriebssystem zu erstellen, das sich für vielfältigste Zwecke optimieren lässt. Durch konsequentes Weiterdenken der verwendeten Ansätze rücken universell konfigurierbare Baukästen in greifbare Nähe, mit denen der Bau eines spezialisierten Betriebssystems mit wie auch immer gearteten Ansprüchen nur noch ein Minimum an Aufwand erfordern würde.

Die verwendeten Konzepte der Kapselung und damit Trennung der Belange sind der richtige Weg, das Problem anzugehen, allerdings stoßen dabei die objektorientierten Ansätze schnell an Grenzen. Hier könnte die aspektorientierte Programmierung Lücken schliessen – gerade wenn es sich um querschneidende Belange handelt, bei denen eine objektorientierte Kapselung schwierig, wenn nicht unmöglich ist.

5 Quellen

Webseiten der einzelnen Projekte:

[1] The OSKit Project: `http://www.cs.utah.edu/flux/oskit/`

[2] THINK – Home Page: `http://think.objectweb.org/`

Referenzen:

[3] B. Ford, G. Back, G. Benson, J. Lepreau, A. Lin, O. Shivers. „The Flux OSKit: A Substrate for OS and Language Research“. Proceedings of the 16th ACM Symposium on Operating Systems Principles, Oktober 1997.
`http://www.cs.utah.edu/flux/papers/oskit-sosp16-abs.html`

[4] B. Ford, K. Van Maren, J. Lepreau, S. Clawson, B. Robinson, J. Turner. „The Flux OS Toolkit: Reusable Components for OS Implementation“. Proceedings of the Sixth IEEE Workshop on Hot Topics in Operating Systems, Mai 1997.
`http://www.cs.utah.edu/flux/papers/oskit-hotos6-abs.html`

[5] J.-P. Fassino, J.-B. Stefani, J. Lawall, and G. Muller. „THINK: A Software Framework for Component-based Operating System Kernels“. Proceedings of Usenix Annual Technical Conference, Juni 2002.
`http://www.usenix.org/events/usenix02/fassino.html`