Betriebssystembaukästen THINK und OSKit

Referent: Tobias Jordan

Inhalt

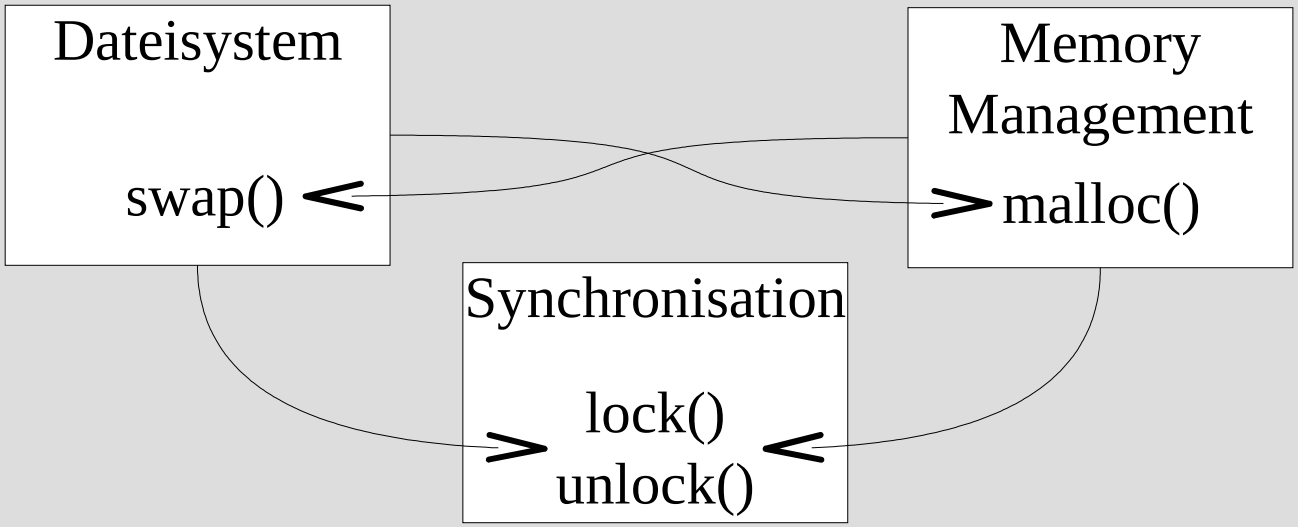
- Motivation
- OSKit
  - Übersicht: Idee / Komponenten
  - Design und Implementierung
  - OSKit in der Praxis
- THINK
  - das THINK Framework
  - Implementierung: KORTEX
  - Praxisbeispiele
- Bewertung / Ausblick

Motivation

- Immer wiederkehrende Aufgaben beim Betriebssystembau
- Betriebssystemkomponenten
  - Bootloader
  - Gerätetreiber
  - Debugging
  - Memory Management
  - ... und jetzt das eigentliche Betriebssystem

Motivation (2)

- “Normale” Vorgehensweise
  - existierenden Code anderer Systeme anpassen
  - schwierig, da normalerweise viele Abhängigkeiten bedacht werden müssen



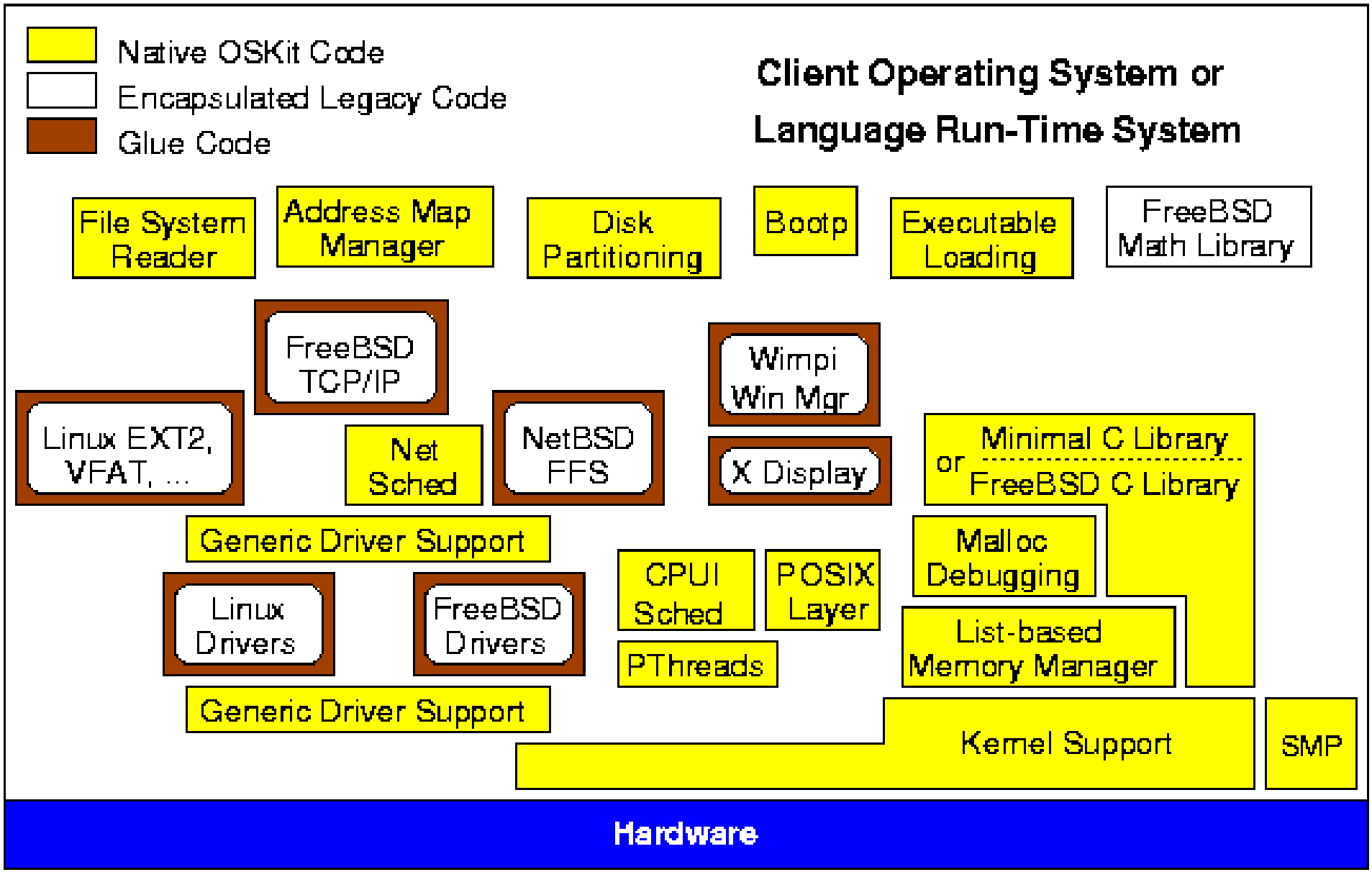
Motivation (3)

- Standardkomponenten
  - austauschbar
  - wiederverwendbar
  - sparen Zeit und Geld
- Wichtig:
  - Kapselung
  - Schnittstellenbeschreibung

OSKit

- Flux Research Group, University of Utah
  - *http://www.cs.utah.edu/flux/oskit*
  - Fluke OS, basierend auf Mach und BSD
- OSKit: Sammlung von Betriebssystemkomponenten
  - Verwendbar als C-Libraries
  - COM-Schnittstellendefinitionen
  - einzelne Komponenten austauschbar

OSKit - Struktur



“Hello World” mit OSKit

```
#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;
}
```

OSKit – Design

- Libraries
  - werden wie gewohnt zum eigenen Code dazugelinkt
  - unabhängig: benötigen keine anderen Pakete
- “Separabiltiy” - Separierbarkeit
  - “glue code” um das Austauschen einzelner Komponenten zu ermöglichen
  - function overriding
    - z.B. `putchar()` oder `malloc()`
  - dynamisches Binden

COM Interfaces

- Sprachunabhängiges Protokoll zur Komponenten-/Schnittstellenbeschreibung
- Implementation Hiding
- Interface Extension/Evolution
  - verschiedene “Sichten” auf ein Objekt
  - Beispiel: `bufio`-Interface “erweitert” `blkio`-Interface
  - “Open Implementation”: Implementierungs-eigenschaften Teil eines erweiterten Interfaces

OSKit – dynamisches Binden

```
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)
```

Performance

- OSKit-Beispielanwendung: Messen von TCP Bandbreite und Latenzzeiten

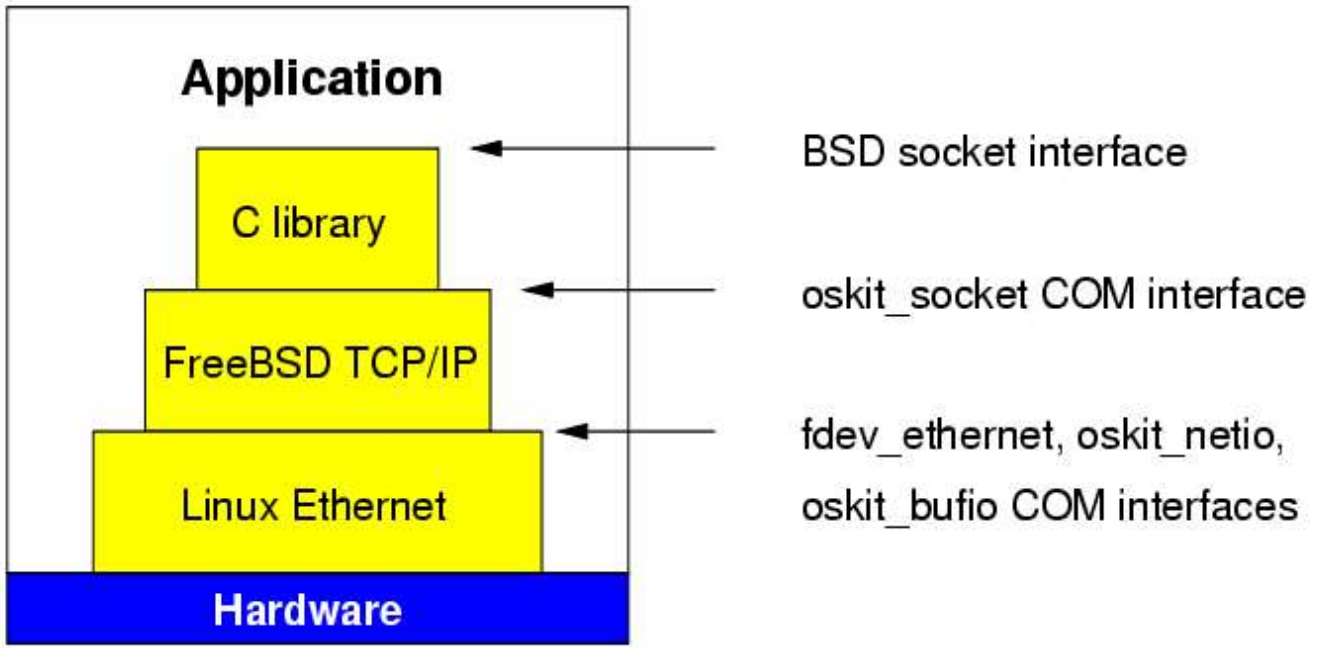


Figure 3: Structure of the `ttcp` and `rtcp` example kernels.

TCP Bandbreite

	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**

TCP-Bandbreite in Mbit/s, gemessen mittels `ttcp` zwischen zwei *PentiumPro*-Rechnern mit 200 MHz, verbunden durch 100 Mbps Ethernet.

TCP Latenz

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

TCP Latenz in µs, gemessen mit `rtcp`

Performance: Ergebnis

- Bandbreite:
  - Senden
    - FreeBSD: mbuf, unzusammenhängend
    - Linux: skbuff, zusammenhängend
    - Umwandlung erfordert manchmal Umkopieren
  - Empfangen
    - keine Probleme
- Latenz
  - glue code

OSKit in der Praxis

- Fluke OS
  - treibende Kraft
- Standard ML
  - funktionale Programmiersprache mit Augenmerk auf Nebenläufigkeit
  - benötigt Zugriff auf Context Switching Code
  - ML/OS auf OSKit-Basis innerhalb eines Semesters fertiggestellt
  - ähnliche Projekte – ohne OSKit – brachten jahrelang keine Ergebnisse

OSKit in der Praxis (2)

- SR/OS
- Java/OS
  - Kaffe JVM auf OSKit “portiert”
  - “Hello World” nach 14 Stunden
  - nach 3 Wochen gleiche Funktionalität wie Suns JavaOS
- diverse andere Projekte
  - Mungi, NILO, L4-Ports, Fiasco, Network Storage Corporation, MzScheme, Janos, (oskitdoom, GNU HURD)

OSKit – Bewertung

- Pro:
  - Sehr einfache Handhabung
    - siehe Praxisberichte
  - Konzept der Trennung der Belange
- Kontra:
  - Teilweise zu „grobkörnig“
  - Teilweise unflexibel

THINK

- THink Is Not a Kernel
- Framework zum komponentenbasierten Bau von Betriebssystemkernen
  - Komponenten verschiedenster Größe
  - flexibles Bindungsmodell
  - Bindung zur Laufzeit
- KORTEX
  - Komponentenbibliothek auf THINK-Basis

THINK – Konzepte

- Domains
  - Ressourcen-, Sicherheits- und Isolationsgrenzen
- Komponenten
  - Laufzeitobjekte
- “Interfaces” - Schnittstellen
  - werden in Java definiert
- Bindungen
  - Verbindungen zwischen Interfaces
- Namen

THINK – Namen/Bindungen

```
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);
}
```

Figure 1: Framework for interfaces, names and bindings

THINK – Implementierung

- In C aus Performancegründen

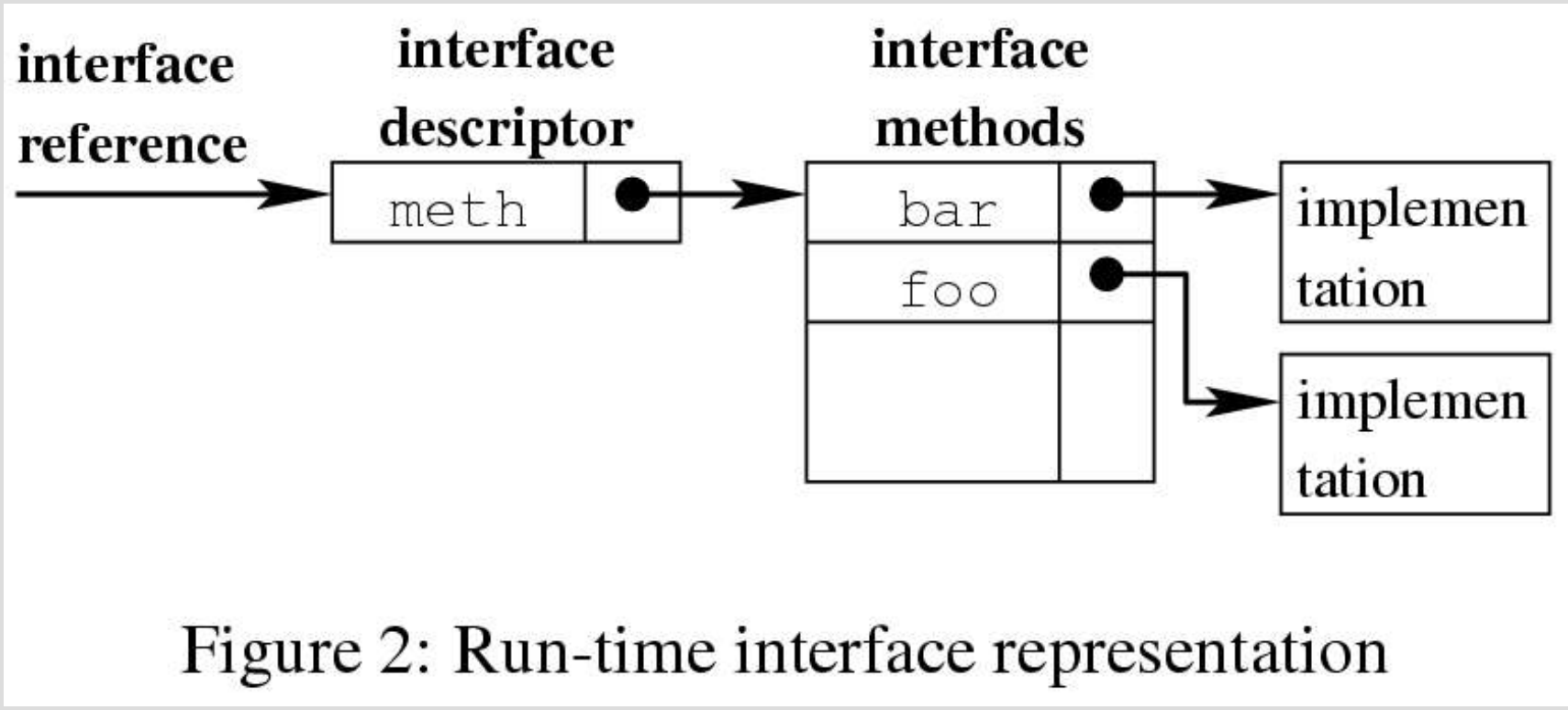


Figure 2: Run-time interface representation

THINK – Implementierung (2)

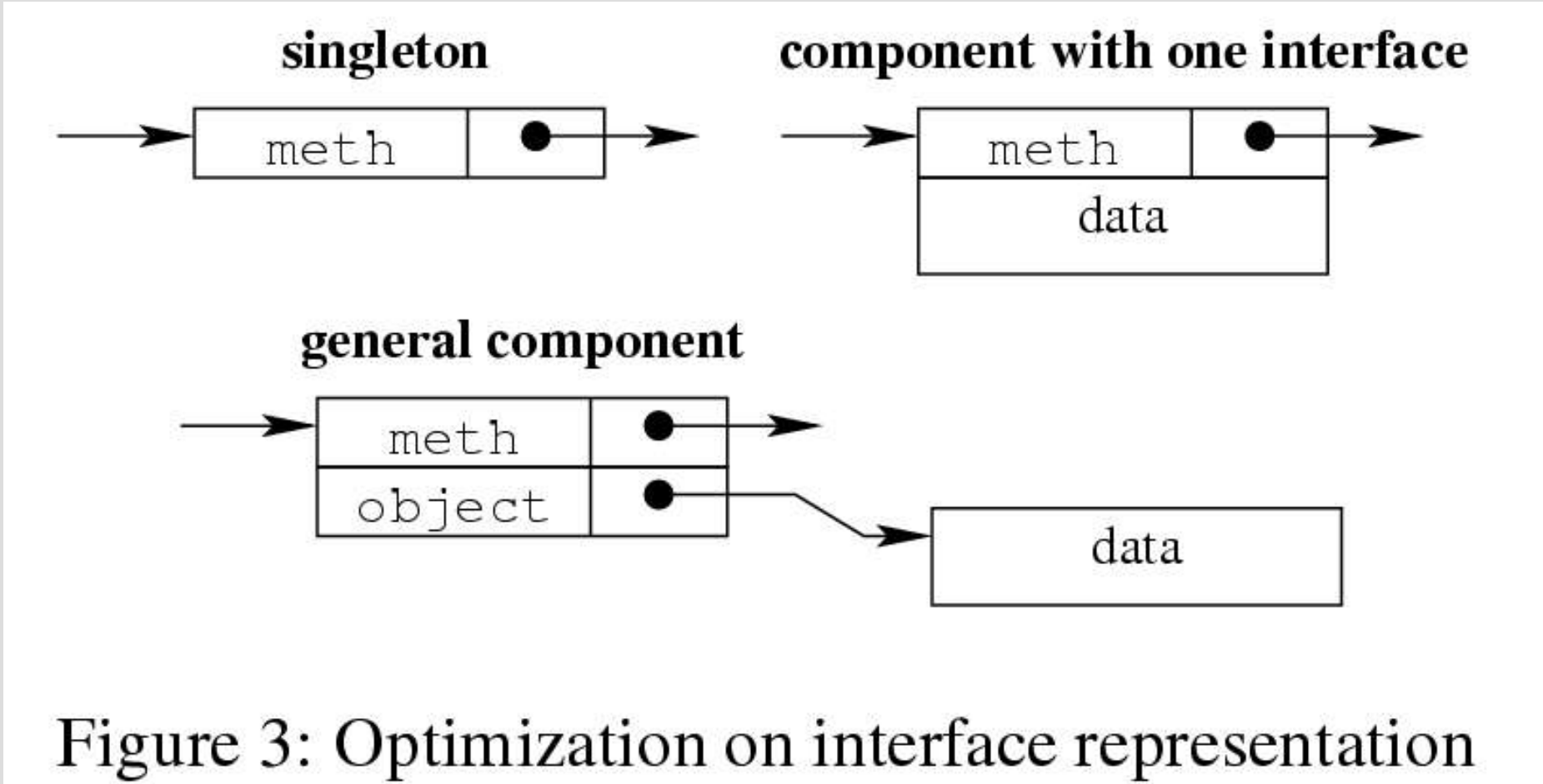


Figure 3: Optimization on interface representation

THINK – Code Generatoren

- Open Interface Compiler
  - erstellt C-/Assembler-Code aus den Java-Schnittstellenbeschreibungen
- Off-line Configurator
  - erstellt Kernel-Images aus UML-Beschreibungen
  - löst Abhängigkeiten zwischen Komponenten

KORTEX

- Komponentenbibliothek für PowerPC
  - HAL-Komponenten
    - Exceptions, MMU, PCI, IDE, Ethernet, Grafik
  - Speicherverwaltung
  - Scheduler/Threads
  - Netzwerk
    - Ethernet, ARP, IP, UDP, TCP, SunRPC
  - Dateisysteme
    - ext2, NFS

KORTEX (2)

- weitere Komponenten
  - Service-Komponenten
    - dynamic linker/loader
    - application loader
  - **Interaktions-Komponenten**
    - stellen Bindungen bereit
  - POSIX-Komponenten

KORTEX – HAL Beispiel

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

Figure 4: Interface for PowerPC exception

Operation	instructions	time (μs)	cycles
`TrapEnter`$_{id}$	57	0.160	80
`TrapReturn`	48	0.110	55
total	105	0.270	135

Table 1: Cost for handling a exception

Ressourcenmanagement

- “Ressource Manager” teilt Ressourcen zu
  - Threads ↔ Scheduler
  - Netzwerksessions ↔ Protokolle

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

Figure 6: Resource management framework

Thread-/Schedulerkomponenten

- drei preemptive Scheduler
  - kooperativ, round-robin, prioritätsbasiert
  - basierend auf HAL-Exception-Komponente, daher sehr performant

	instructions	**time (μs)**	**cycles**
thread switch	111	0.284	142
process switch	147	0.394	197

Table 2: Context switching costs

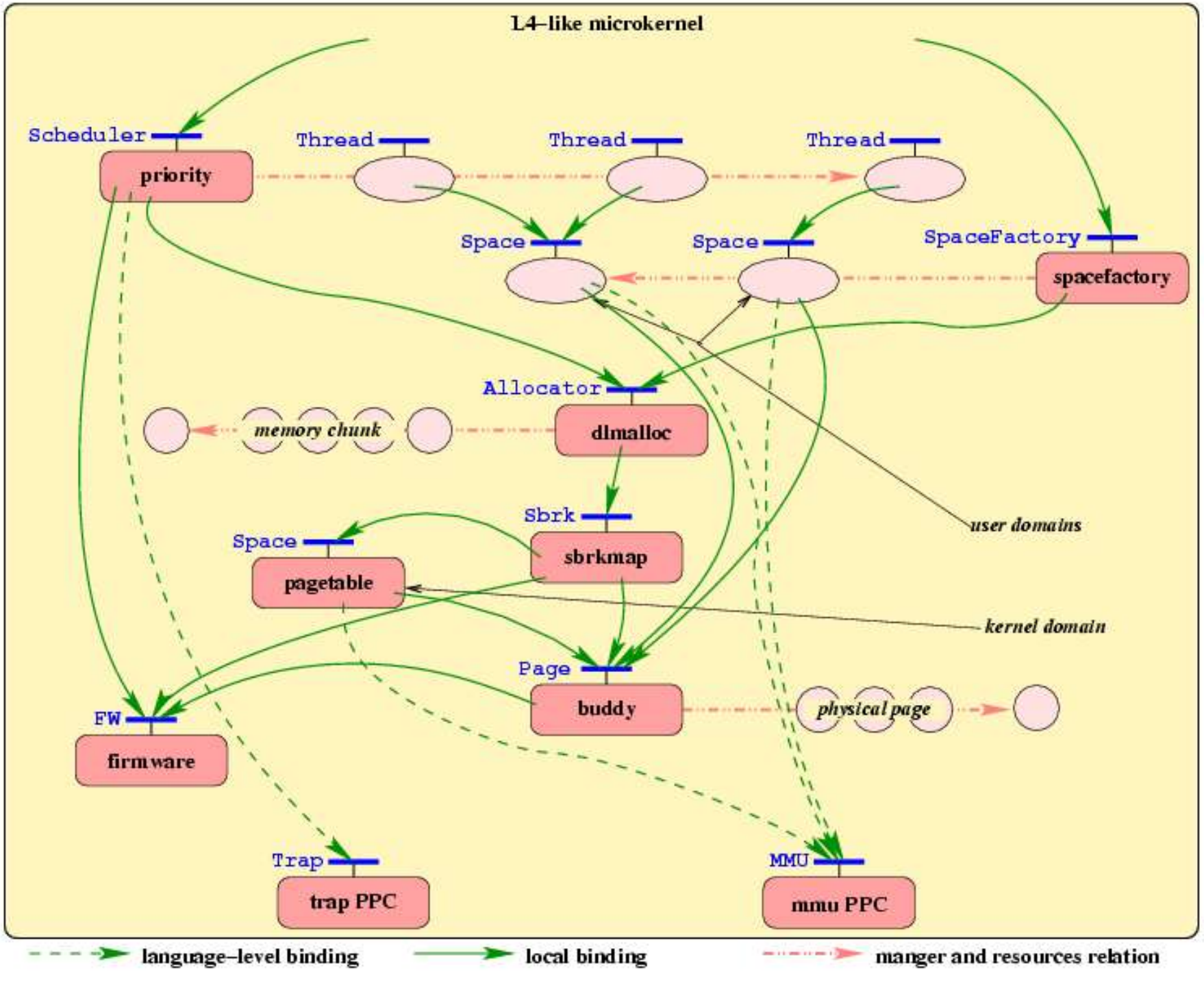
Interaktions-Komponenten

- Lokale Bindung
  - für Komponenten derselben Domain
- Syscalls
  - Aufruf von Kerneldiensten aus Applikationen
  - Exception → Exception-Handler → Komponente
- Upcalls / Signale
  - Mitteilen von Kernel-Ereignissen an Applikationen
  - Kernel → Signal-Handler-Komponente

Interaktions-Komponenten (2)

- Synchrones LRPC
  - Kommunikation zwischen Applikationen
  - benutzt Syscall und Upcall
- Remote RPC
  - Kommunikation zwischen Kerneln auf verschiedenen Rechnern
  - benutzt Netzwerkkomponenten
  - kann durch Syscall/Upcall auch auf Applikationsebene benutzt werden

Praktische Umsetzung



Praktische Umsetzung (2)

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

Table 3: Performance of KORTEX bindings

vgl.: Linux 2.4 getpid-Syscall: 217 Zyklen

Network type	Time (μs)			
	total	network		marshall. +null call
		link	driver	
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%)

Table 4: Performance of synchronous remote binding

Weitere Tests

- PlanP-Bridge
  - Sprache zur Programmierung aktiver Netzwerkkomponenten
  - Durchsatz des dedizierten KORTEX-Kerns besser als unter Solaris/Linux
- Kaffe – Java VM
  - Thread-Performanz besser als unter Linux
    - Ausnutzung der HAL-Komponenten

DOOM

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

Table 7: Doom frames per second

THINK/OSKit: Unterschiede

- THINK: Bindungskonzept
  - dadurch zusätzliche Abstraktionsebene
- OSKit verwendet viel “Legacy Code”, KORTEX eigene Komponenten
  - OSKit-Komponenten dadurch “grobkörniger”
- OSKit enthält keine speziellen Frameworks
  - KORTEX: Frameworks zu Kommunikation und Ressourcenverwaltung

Bewertung, Ausblick

- Zentrales Thema: Trennung der Belange
  - bei beiden Baukästen objektorientierter Ansatz
  - Aber: bei manchen querschneidenden Belangen keine objektorientierte Kapselung möglich

⇒ Lösung: aspektorientierte Programmierung

Quellen

- Webseiten der einzelnen Projekte:
  - The OSKit Project: `http://www.cs.utah.edu/flux/oskit/`
  - THINK – Home Page: `http://think.objectweb.org/`
- Referenzen:
  - The Flux OSKit: A Substrate for OS and Language Research
  - The Flux OS Toolkit: Reusable Components for OS Implementation
  - THINK: A Software Framework for Component-based Operating System Kernels