

# Betriebssystembau (BSB)

## VL 10 – Architekturen

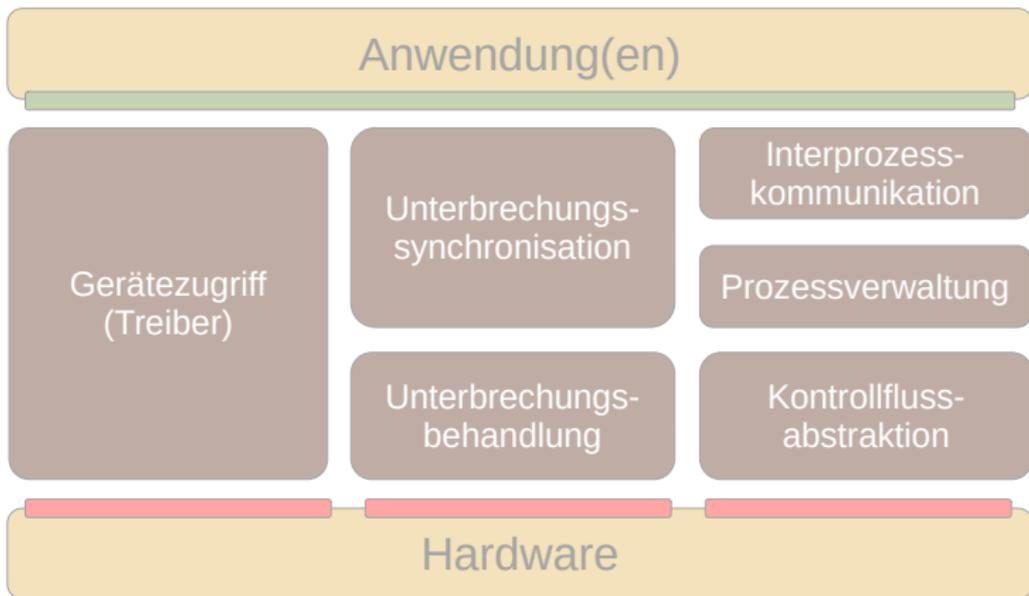
### Alexander Krause

Lehrstuhl für Informatik 12 – Arbeitsgruppe Systemsoftware  
Technische Universität Dortmund

<https://sys.cs.tu-dortmund.de/de/lehre/ws24/bsb>

WS 24 – 10. Dezember 2024

# Überblick: Einordnung dieser VL



Betriebssystementwicklung

# Agenda

---

Einführung  
Geschichte, Mode und Trend  
Zusammenfassung  
Referenzen



## Einführung

Bewertungskriterien für Betriebssysteme

Paradigmen der Betriebssystementwicklung

Geschichte, Mode und Trend

Zusammenfassung

Referenzen



- Anwendungsorientierte Kriterien
  - **Portabilität**
  - **Erweiterbarkeit**
  - **Robustheit**
  - **Leistung**
- Technische Kriterien (Architektureigenschaften)
  - **Isolationsmechanismus**
  - **Interaktionsmechanismus**
  - **Unterbrechungsmechanismus**



## ■ Anwendungsorientierte Kriterien

### ■ **Portabilität**

- *Wie unabhängig ist man von der Hardware?*

### ■ **Erweiterbarkeit**

- *Wie leicht lässt sich das System erweitern (z. B. um neue Gerätetreiber)?*

### ■ **Robustheit**

- *Wie stark wirken sich Fehler in Einzelteilen auf das Gesamtsystem aus?*

### ■ **Leistung**

- *Wie gut ist die Hardware durch die Anwendung auslastbar?*

## ■ Technische Kriterien (Architektureigenschaften)

### ■ **Isolationsmechanismus**

- *Wie werden Anwendungen / BS-Komponenten isoliert?*

### ■ **Interaktionsmechanismus**

- *Wie kommunizieren Anwendungen / BS-Komponenten miteinander?*

### ■ **Unterbrechungsmechanismus**

- *Wie werden Unterbrechungen zugestellt und bearbeitet?*

## Paradigmen der Betriebssystementwicklung

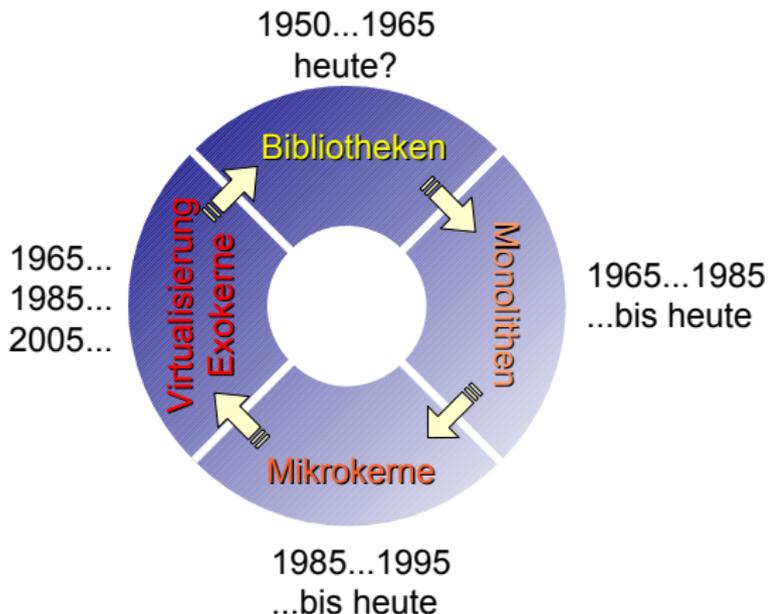
### Definition: Paradigma

Das Wort **Paradigma** [...] bedeutet „*Beispiel*“, „*Vorbild*“, „*Muster*“ oder „*Abgrenzung*“, „*Vorurteil*“; in allgemeinerer Form auch „**Weltsicht**“ oder „**Weltanschauung**“.

[Wikipedia]



## Paradigmen der Betriebssystementwicklung



Einführung

**Geschichte, Mode und Trend**

Bibliotheks-Betriebssysteme

Monolithen

Mikrokerne

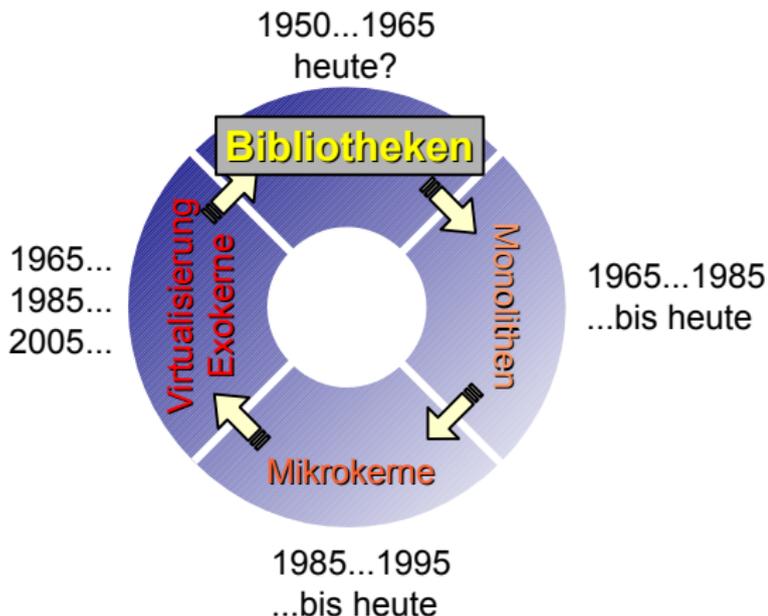
Exokerne und Virtualisierung

Zusammenfassung

Referenzen



## Funktionsbibliotheken als einfache Infrastrukturen

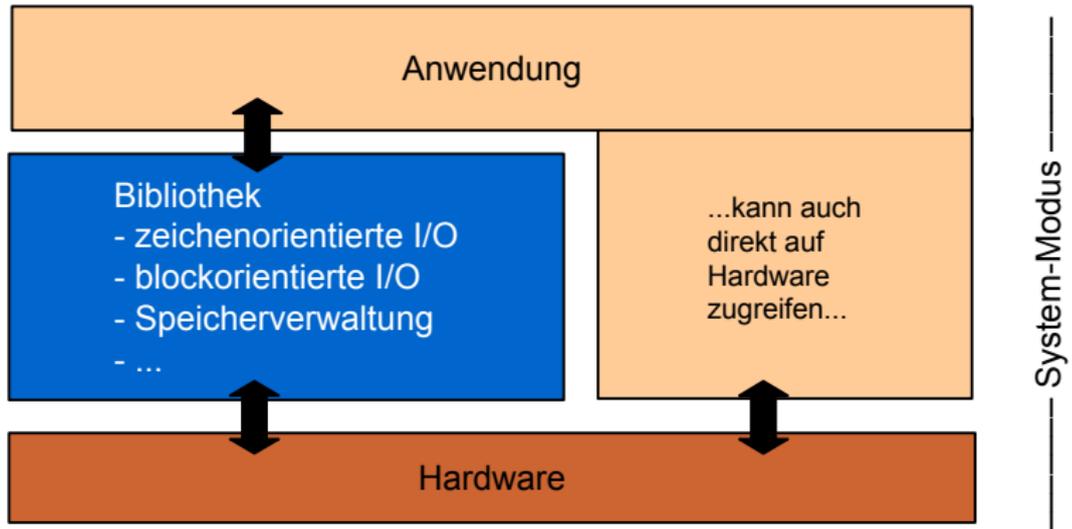


- Erste Rechnersysteme besaßen keinerlei Systemsoftware
  - Jedes Programm musste die gesamte Hardware selbst ansteuern
  - Systeme liefen Operator-gesteuert im Stapelbetrieb
    - single tasking, Lochkarten
  - Peripherie war vergleichsweise einfach
    - Seriell angesteuerter Lochkartenleser und -schreiber, Drucker, Bandlaufwerk
  
- Code zur Geräteansteuerung wurde in jedem Anwendungsprogramm repliziert
  - Die Folge war eine massive Verschwendung von
    - Entwicklungszeit (teuer!)
    - Übersetzungszeit (sehr teuer!)
    - Speicherplatz (teuer!)
  - außerdem eine hohe Fehleranfälligkeit



- **Logische Folge: Bibliotheks-Betriebssysteme**
  - Zusammenfassung von häufig benutzten Funktionen zur Ansteuerung von Geräten in **Software-Bibliotheken** (*Libraries*)
    - Systemfunktionen als „normale“ Subroutinen
  - Funktionen der Bibliothek waren dokumentiert und getestet
    - verringerte Entwicklungszeit (von Anwendungen)
    - verringerte Übersetzungszeit (von Anwendungen)
  - Bibliotheken konnten resident im Speicher des Rechners bleiben
    - verringerter Speicherbedarf (der Anwendungen)
    - verringerte Ladezeit (der Anwendungen)
  - Fehler konnten von Experten zentral behoben werden
    - verbesserte Zuverlässigkeit





## ■ Anwendungsorientierte Kriterien

### ■ **Portabilität**

- keine Standards, eigene Bibliotheken für jede Architektur

gering

### ■ **Erweiterbarkeit**

- theoretisch gut, in der Praxis oft „Spaghetti-Code“

mäßig

### ■ **Robustheit**

- *single tasking*, Kosten für Programmwechsel sehr hoch

sehr hoch

### ■ **Leistung**

- direktes Operieren auf der Hardware, keine Privilegebenen

sehr hoch

## ■ Technische Kriterien (Architektureigenschaften)

### ■ **Isolationsmechanismus**

- Anwendung  $\equiv$  System

nicht erforderlich

### ■ **Interaktionsmechanismus**

- Betriebssystem  $\equiv$  Bibliothek

Funktionsaufrufe

### ■ **Unterbrechungsmechanismus**

- Kommunikation mit Geräten über *polling*

oft nicht vorhanden

# Probleme: Bibliotheks-Betriebssysteme

---

- Teure Hardware wird nicht optimal ausgelastet
  - Hoher Zeitaufwand beim Wechseln der Anwendung
  - Warten auf Ein-/Ausgabe verschwendet unnötig CPU-Zeit
- Organisatorische Abläufe sehr langwierig
  - Stapelbetrieb, Warteschlangen
  - von der Abgabe eines Programms bis zum Erhalt der Ergebnisse vergehen oft Tage – um dann festzustellen, dass das Programm in der ersten Zeile einen Fehler hatte...
- Keine Interaktivität möglich
  - Betrieb durch Operatoren, kein direkter Zugang zur Hardware
  - Programmabläufe nicht zur Laufzeit parametrierbar



Einführung

**Geschichte, Mode und Trend**

Bibliotheks-Betriebssysteme

**Monolithen**

Mikrokerne

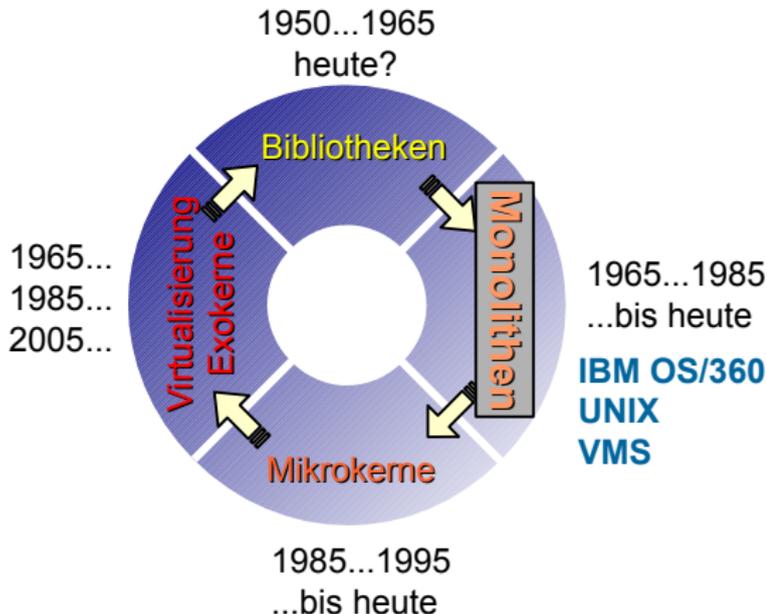
Exokerne und Virtualisierung

Zusammenfassung

Referenzen



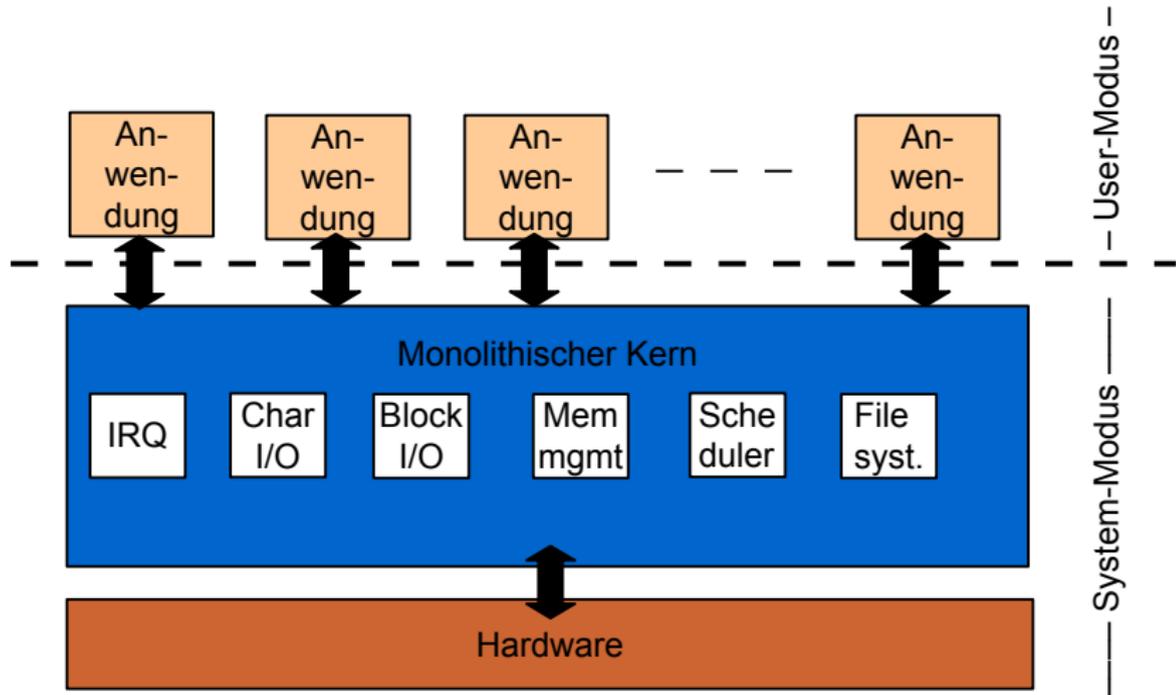
## Monolithen als Herrscher über das System



- **Motivation:** Mehrprogrammbetrieb
- **Problem:** Isolation
- **Ansatz:** BS als Super-Programm, Kontrollinstanz
  - Programme laufen unter der Kontrolle des Betriebssystems
  - Dadurch erstmals (sinnvoll) Mehrprozess-Systeme realisierbar
- Einführung eines Privilegiensystems
  - Systemmodus  $\longleftrightarrow$  Anwendungsmodus
  - Direkter Hardwarezugriff nur im Systemmodus
    - $\rightsquigarrow$  Gerätetreiber gehören zum System
- Einführung neuer Hard- und Software-Mechanismen
  - *Traps* in den Kern
  - Kontextumschaltung und -sicherung
  - *Scheduling* der Betriebsmittel



# Architektur: Monolithische Systeme



# Monolithische Systeme: IBM OS/360

---

- Eines der ersten monolithischen Betriebssysteme
  - Ziel: gemeinsames BS für alle IBM-Großrechner
  - Leistung und Speicher der Systeme differierten aber um Zehnerpotenzen zwischen „kleinen“ und „großen“ 360-Systemen
- Diverse Konfigurationen
  - PCP (Primary Control Program) 1965
    - Einprozessbetrieb, kleine Systeme
  - MFT (Multiprogramming with Fixed number of Tasks) 1966
    - mittlere Systeme (256 kB RAM)
    - feste Speicherpartitionierung zwischen Prozessen, feste Anzahl an Tasks
  - MVT (Multiprogramming with Variable number of Tasks): 1967
    - high end
    - Paging, optional Time Sharing Option (TSO) für interaktive Nutzung



# Monolithische Systeme: IBM OS/360



- Richtungsweisende Konzepte
  - Hierarchisches Dateisystem
  - Prozesse können Unterprozesse erzeugen
  - Familienansatz: MFT und MVT sind von API und ABI her kompatibel
- Große Probleme bei der Entwicklung
  - Fred Brooks: „The Mythical Man-Month“ [3] **lesenswert!**
  - Problem der Konzeptuellen Integrität
    - Separation von Architektur und Implementierung war schwierig
  - „Second System Effect“
    - Entwickler wollten die „eierlegende Wollmilchsau“ bauen
  - Zu komplexe Abhängigkeiten zwischen Komponenten des Systems
    - Ab einer gewissen Codegröße blieb die Anzahl der Fehler konstant

~> Geburt der **Softwaretechnik**

- Ziel: Mehrprogrammbetrieb auf „kleinen“ Computern
  - Entwicklung seit Anfang der 70er Jahre
    - Kernelgröße im Jahr 1979 (7th Edition Unix, PDP11): ca. 50kB
  - von ursprünglich 2-3 Entwicklern geschrieben
    - überschaubar und handhabbar, ca. 10.000 Zeilen Quelltext
- Neu: Portabilität durch Hochsprache
  - **C** als domänenspezifische Sprache für Systemsoftware
  - UNIX wurde mit den Jahren auf nahezu jede Plattform portiert
- Weitere richtungsweisende Konzepte:
  - alles ist eine Datei, dargestellt als ein Strom von Bytes
  - komplexe Prozesse werden aus einfachen Programmen komponiert
    - Konzept der Pipe, Datenflussparadigma



# Monolithische Systeme: Bell Labs/AT&T UNIX



- Weitere Entwicklung von UNIX erfolgte stürmisch
  - Systeme mit großem Adressraum (VAX, RISC)
  - Der Kernel ist „mit gewachsen“ (System III, System V, BSD)
    - ohne wesentliche Strukturänderungen
  - Immer mehr komplexe Subsysteme wurden integriert
    - TCP/IP ist ungefähr so umfangreich wie der Rest des Kernels
- Linux orientiert(e) sich an der Struktur von System V
- UNIX war und ist einflussreich im akademischen Bereich durch frühe „Open Source“-Politik der Bell Labs
  - Viele Portierungen und Varianten entstanden
    - oftmals parallel zu Hardwareentwicklungen
  - In der akademischen Welt wurde UNIX zum Referenzsystem
    - Ausgleichspunkt und Vergleichssystem für alle neueren Ansätze



## ■ Anwendungsorientierte Kriterien

- **Portabilität** hoch
  - dank „C“ kann und konnte UNIX einfach portiert werden
- **Erweiterbarkeit** mäßig
  - von Neukompilierung  $\rightsquigarrow$  Modulkonzept
- **Robustheit** mäßig
  - Anwendungen isoliert, nicht jedoch BS-Komponenten (Treiber!)
- **Leistung** hoch
  - Nur Betreten / Verlassen des Kerns ist teuer

## ■ Technische Kriterien (Architektureigenschaften)

- **Isolationsmechanismus** Privilegiebenen, Adressräume
  - Pro Anwendung ein Adressraum, Kern läuft auf Systemebene
- **Interaktionsmechanismus** Funktionsaufrufe, Traps
  - Anwendung  $\rightarrow$  Kern durch *Traps*, innerhalb des Kerns durch `call` / `ret`
- **Unterbrechungsmechanismus** Bearbeitung im Kern
  - interne Unterteilung in UNIX: *bottom half*, *top half*

- Monolithen sind schwer handhabbar
  - Hinzufügen oder Abändern von Funktionalität betrifft oft mehr Module, als der Entwickler vorhergesehen hat
- Eingeschränkte Synchronisationsmechanismen
  - Oft nur ein „Big Kernel Lock“, d. h. nur ein Prozess kann zur selben Zeit im Kernmodus ausgeführt werden, alle anderen warten
  - Insbesondere bei Mehrprozessor-Systemen leistungsreduzierend
- Gemeinsamer Adressraum aller Kernkomponenten
  - Sicherheitsprobleme in einer Komponente (z.B. buffer overflow) führen zur Kompromittierung des gesamten Systems
  - Viele Komponenten laufen überflüssigerweise im Systemmodus
  - Komplexität und Anzahl von Treibern hat extrem zugenommen



Einführung

**Geschichte, Mode und Trend**

Bibliotheks-Betriebssysteme

Monolithen

**Mikrokerne**

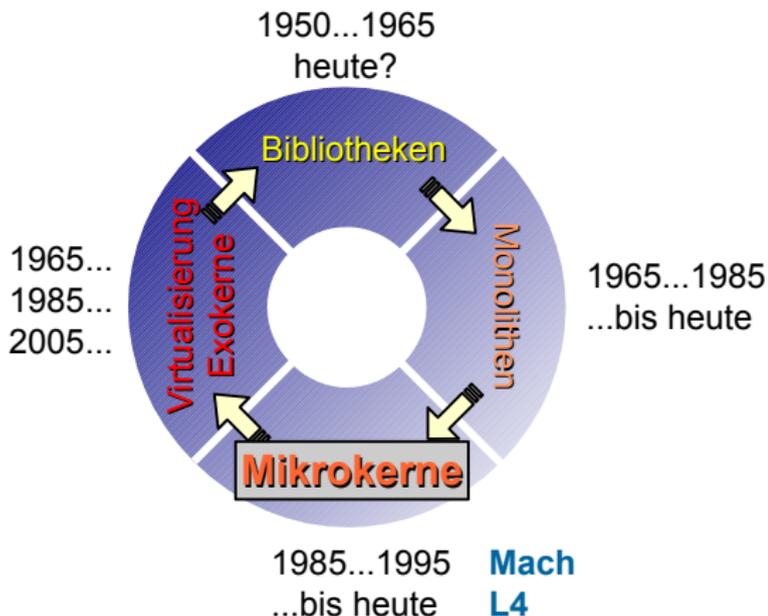
Exokerne und Virtualisierung

Zusammenfassung

Referenzen



## Mikrokerne als Reduktion auf das Notwendige



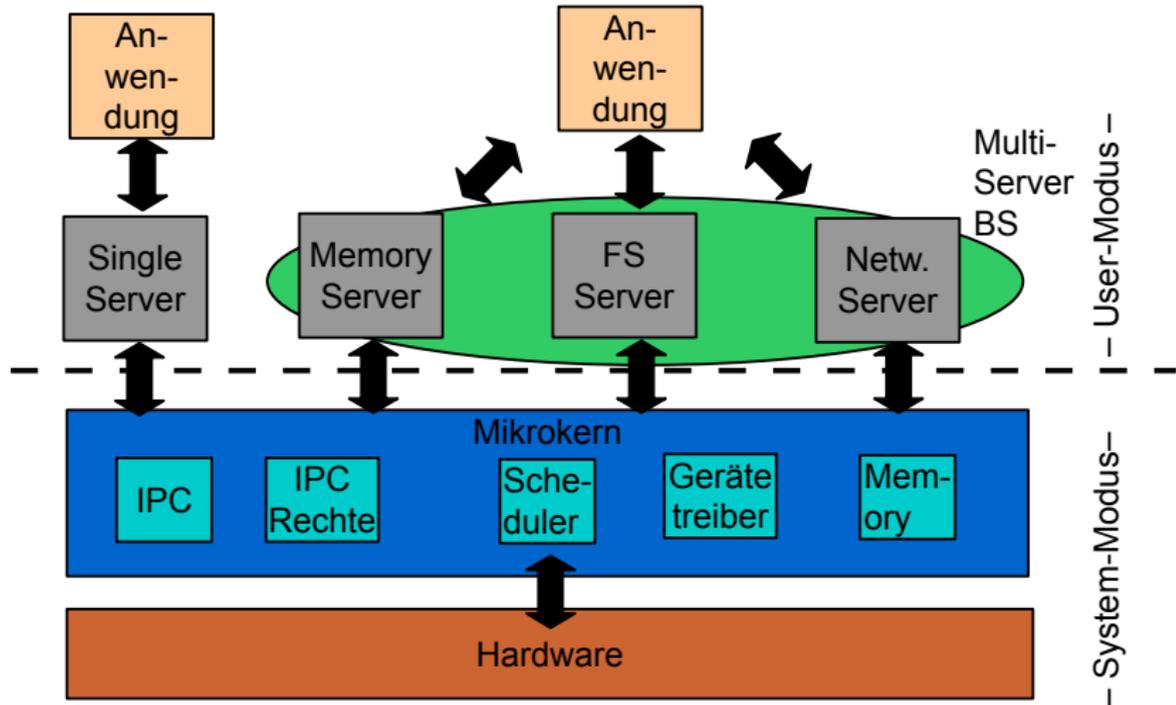
- Ziel: Reduktion der Trusted Computing Base (TCB)
  - Minimierung der im privilegierten Modus ablaufenden Funktionalität
  - BS-Komponenten als Server-Prozesse im nichtprivilegierten Modus
  - Interaktion über Nachrichten (IPC, *Inter Process Communication*)
- Prinzip des geringsten Privilegs
  - Systemkomponenten müssen nur so viele Privilegien besitzen, wie zur Ausführung ihrer Aufgabe erforderlich sind
    - z.B. Treiber: Zugriff auf spezielle IO-Register, nicht auf die gesamte HW
  - Nur der Mikrokern läuft im Systemmodus
- Geringere Codegröße
  - L4: 10 kloc C++  $\longleftrightarrow$  Linux: 1 Mloc C (ohne Treiber)
  - Ermöglicht Ansätze zur formalen Verifikation des Mikrokerns [6]



- **Ziel:** Reduktion der TCB
- **Ziel:** Schaffung eines extrem portablen Systems
- **Ziel:** Verbesserung der Unix-Konzepte
  - Neue Kommunikationsmechanismen via IPC und Ports
    - Ports sind sichere IPC-Kommunikationskanäle
    - IPC ist optional netzwerktransparent: Unterstützung für verteilte Systeme
  - Parallele Aktivitäten innerhalb eines Prozessadressraums
    - Unterstützung für Fäden  $\rightsquigarrow$  neuer Prozessbegriff als „Container“
    - Bessere Unterstützung für Mehrprozessorsysteme
  - Unterstützung „fremder“ Systemschnittstellen durch Personalities
- Ausgangspunkt: BSD UNIX
  - Schrittweise Separation der Funktionalität, die nicht im privilegierten Modus laufen muss in Benutzermodus-Prozesse
  - Anbindung über Ports und IPC



# Architektur: Mikrokerne erster Generation



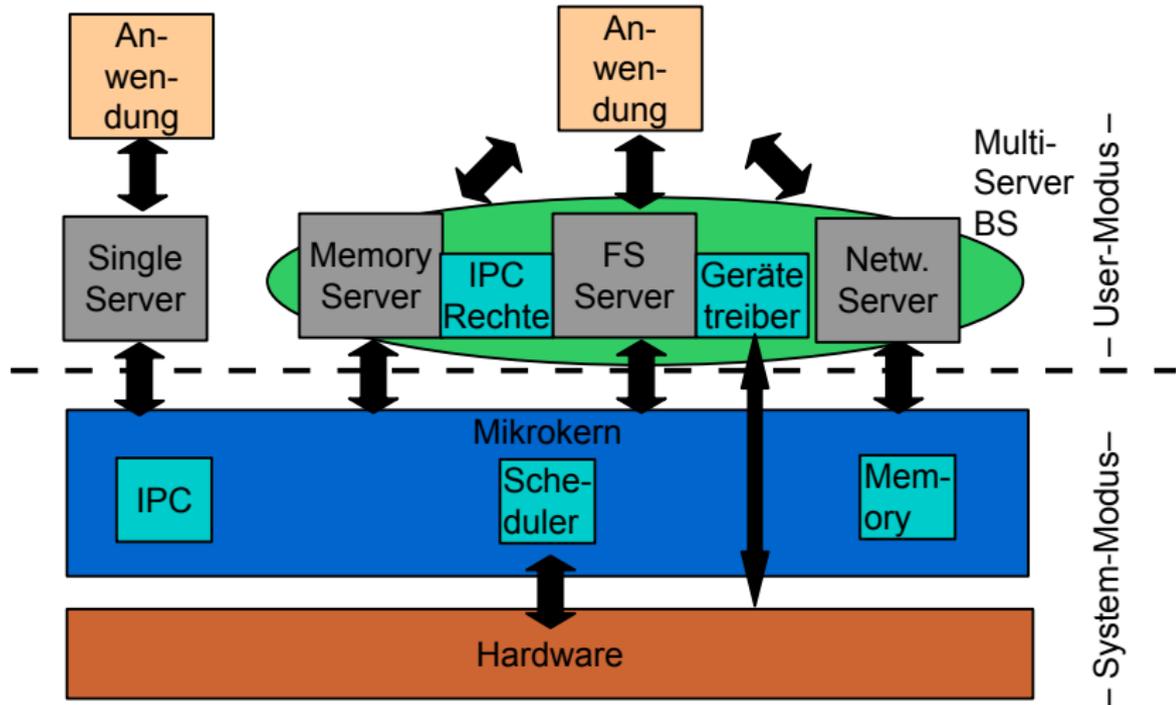
- Probleme von Mach
  - hoher Overhead für IPC-Operationen
    - Systemaufrufe **Faktor 10** langsamer gegenüber monolithischem Kern
  - Immer noch viel zu große Code-Basis
    - Gerätetreiber und Rechteverwaltung für IPC im Mikrokernel
    - $\rightsquigarrow$  die eigentlichen Probleme nicht gelöst!
  - Führt zu schlechtem Ruf von Mikrokerneln allgemein
    - Einsetzbarkeit in der Praxis wurde bezweifelt
- Die Mikrokernel-Idee galt Mitte der 90er Jahre als tot
  - Praktischer Einsatz von Mach erfolgte nur in hybriden Systemen
    - Separat entwickelte Komponenten für Mikrokernel und Server
    - Kolokation der Komponenten in einem Adressraum, Ersetzen von in-kernel IPC durch Funktionsaufrufe
  - Bekanntestes Beispiel: Apple OS X  $\rightarrow$  Mach 3 Mikrokernel + FreeBSD



- **Ziel:** Mikrokern, diesmal aber richtig!
  - Verzicht auf Sekundärziele: Portabilität, Netzwerktransparenz, ...
- **Ansatz:** Reduktion auf das Notwendigste
  - Ein Konzept wird nur dann innerhalb des Mikrokerns toleriert, wenn seine Auslagerung die Implementierung verhindern würde.
  - synchroner IPC, Kontextwechsel, CPU Scheduler, Adressräume
- **Ansatz:** Gezielte Beschleunigung
  - fast IPCs durch Parameterübergabe in Registern
  - Gezielte Reduktion der Cache-Load (durch sehr kleinen Kern)
- Viele von Mikrokernen der 1. Generation noch im Systemmodus implementierte Funktionalität ausgelagert
  - z. B. Überprüfung von IPC-Kommunikationsrechten
  - vor allem aber: Treiber



# Architektur: Mikrokerne zweiter Generation



## ■ Anwendungsorientierte Kriterien

### ■ **Portabilität**

- ursprünglich rein in Assembler, aktuell in C+ entwickelt

mäßig

### ■ **Erweiterbarkeit**

- durch neue Server im Benutzermodus, auch zur Laufzeit

sehr hoch

### ■ **Robustheit**

- durch strikte Isolierung

sehr hoch

### ■ **Leistung**

- IPC-Performance ist **der** kritische Faktor

mäßig – gut

## ■ Technische Kriterien (Architektureigenschaften)

### ■ **Isolationsmechanismus**

- Ein Adressraum pro Anwendung,  
ein Adressraum pro Systemkomponente

Adressräume

### ■ **Interaktionsmechanismus**

- Anwendungen und Systemkomponenten interagieren über Nachrichten

IPC

### ■ **Unterbrechungsmechanismus**

- Unterbrechungsbehandlung erfolgt durch Faden im Benutzermodus

IPC an Server-Prozess

Einführung

**Geschichte, Mode und Trend**

Bibliotheks-Betriebssysteme

Monolithen

Mikrokerne

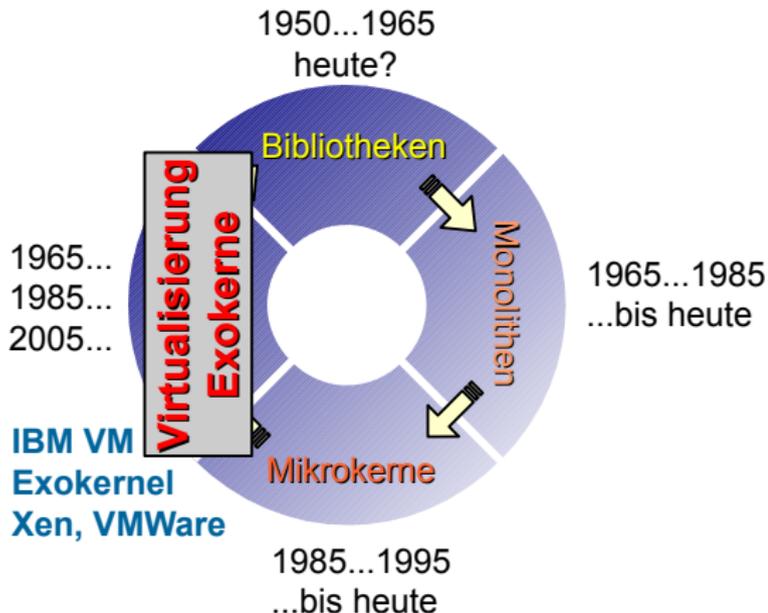
**Exokerne und Virtualisierung**

Zusammenfassung

Referenzen



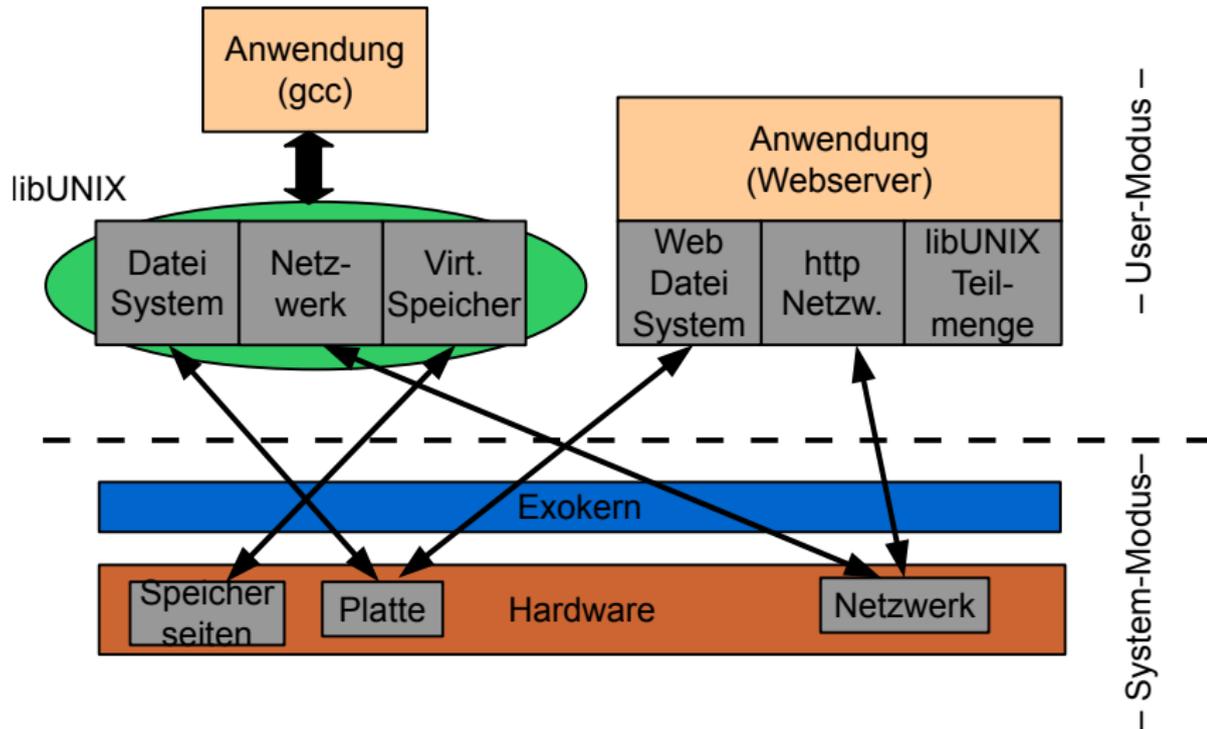
## Exokerne und Virtualisierung als weitere Reduktion



- Ziel: Leistungsverbesserung durch Reduktion
  - Entfernung von Abstraktionsebenen
  - Implementierung von Strategien (z.B. Scheduling) in der Anwendung
- Extrem kleiner Kern, dieser implementiert nur
  - Schutz
  - Multiplexing von Ressourcen (CPU, Speicher, Disk-Blöcke, ...)
- Trennung von Schutz und Verwaltung der Ressourcen!
  - Keine Implementierung von IPC-Mechanismen (Mikrokern) oder weiterer Abstraktionen (Monolithen)
  - Anwendungen können die *für sie* idealen Abstraktionen, Komponenten und Strategien verwenden



# Architektur: Exokern-Betriebssysteme



## ■ Anwendungsorientierte Kriterien

### ■ **Portabilität**

- Exokerne sind sehr klein

sehr hoch

### ■ **Erweiterbarkeit**

- aber auch erforderlich! – der Exokern stellt kaum Funktionalität bereit

sehr hoch

### ■ **Robustheit**

- Schutz wird durch den Exokern bereitgestellt

gut

### ■ **Leistung**

- Anwendungen operieren nahe an der Hardware, wenige Abstraktionsebenen

sehr gut

## ■ Technische Kriterien (Architektureigenschaften)

### ■ **Isolationsmechanismus**

- Ein Adressraum pro Anwendung  
+ von ihr gebrauchter Systemkomponenten

Adressräume

### ■ **Interaktionsmechanismus**

- wird von der Anwendung bestimmt

nicht vorgegeben

### ■ **Unterbrechungsmechanismus**

- Exokern verhindert nur die Monopolisierung der CPU

nicht vorgegeben

## Probleme: Exokern-Betriebssysteme

---

- Exokernel sind nicht als Basis für die Verwendung mit beliebigen „legacy“-Anwendungen geeignet
- Anwendungen haben volle Kontrolle über Abstraktionen
  - müssen diese aber auch implementieren
  - hohe Anforderungen an Anwendungsentwickler
- Definition von Exokern-Schnittstellen ist schwierig
  - Bereitstellung adäquater Schnittstellen zur System-Hardware
  - Genaue Abwägung zwischen Mächtigkeit, Minimalismus und ausreichendem Schutz
- Bisher kein Einsatz in Produktionssystemen
  - Es existieren lediglich einige *proof-of-concept-Systeme*
  - Viele Fragen der Entwicklung von BS-Bibliotheken noch offen



- Ziel: Isolation und Multiplexing
  - unterhalb** der Systemebene
- Ansatz: Virtual Machine Monitor (VMM) / Hypervisor
  - Softwarekomponente, läuft direkt auf der Hardware
  - stellt Abstraktion Virtual Maschine (VM) zur Verfügung
- VM simuliert die gesamten Hardware-Ressourcen
  - Prozessoren, Speicher, Festplatten, Netzwerkkarten, ...
  - Container für beliebige Betriebssysteme nebst Anwendungen
- Vergleich zu Exokernen
  - gröbere Granularität der zugeteilten Ressourcen
    - z.B. gesamte Festplattenpartition vs. einzelne Blöcke
  - „brute force“ Ansatz
    - Multiplexen ganzer Rechner statt einzelner Betriebsmittel
  - Anwendungen (und BS) brauchen nicht angepasst werden



- Für IBM 360-Großrechner existierten mehrere Betriebssysteme
  - DOS/360, MVS: Stapel-orientierte Bibliotheks-Betriebssysteme
  - OS/360: Midrange Server-System
  - TSS/360: Interaktives Mehrbenutzersystem mit Time-Sharing
  - Kundenspezifische Entwicklungen
- Problem: wie kann man Anwendungen für all diese Systeme *gleichzeitig* verwenden?
  - Hardware war teuer (Millionen von USD)
- Entwicklung der ersten Systemvirtualisierung „VM“ durch Kombination aus Emulation und Hardware-Unterstützung
  - Harte Partitionierung der Betriebsmittel
  - Gleichzeitiger Betrieb von stapelverarbeitenden und interaktiven Betriebssystemen wurde ermöglicht



- Ausgangslage: Problematik wie bei IBM in den 60er Jahren
  - Hardware wird immer leistungsfähiger – wohin mit den Ressourcen?
  - Ablauf mehrerer Betriebssystem-Instanzen gleichzeitig
  - Serverkonsolidierung, Kompatibilität zu Anwendungen
- Problem: IA-32 ist eigentlich nicht virtualisierbar
  - Virtualisierungskriterien von Popek und Goldberg [8] sind nicht erfüllt
  - Insbesondere: Äquivalenzanforderung – nicht alle Ring 0 Befehle trappen bei Ausführung auf Ring 3
- Ansatz: Paravirtualisierung
  - „kritische Befehle“ werden ersetzt
    - entweder zur Übersetzungszeit (Xen) oder zur Laufzeit (VMWare)
  - VMs laufen in Ring 3, Ringmodell durch Adressräume nachgebildet
    - Die meisten BS verwenden eh nur Ring 0 und Ring 3
- Neue IA-32 CPUs unterstützen Virtualisierung in HW (↪ VL 6)
  - Paravirtualisierung in der Praxis oft noch performanter



## ■ Anwendungsorientierte Kriterien

### ■ **Portabilität**

- sehr hardware-spezifisch, Paravirtualisierung ist aufwändig

gering

### ■ **Erweiterbarkeit**

- in den üblichen VMMs nicht vorgesehen

keine

### ■ **Robustheit**

- grobgranular auf der Ebene von VMs

gut

### ■ **Leistung**

- stark abhängig vom Einsatzszenario (CPU-lastig, IO-lastig, ...)

mäßig – gut

## ■ Technische Kriterien (Architektureigenschaften)

### ■ **Isolationsmechanismus**

- Jede Instanz bekommt einen eigenen Satz an Hardwaregeräten

VM, Paravirtualisierung

### ■ **Interaktionsmechanismus**

- Anwendungen in den VMS kommunizieren miteinander über TCP/IP

nicht vorgesehen

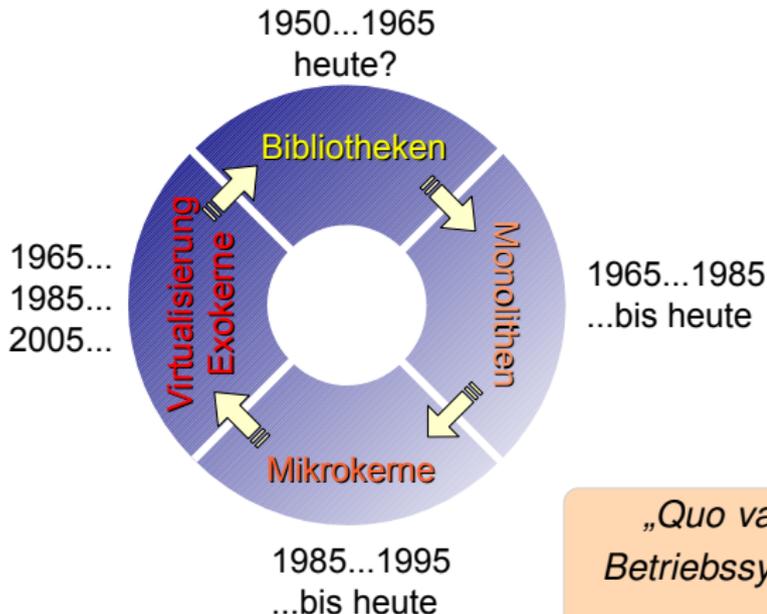
### ■ **Unterbrechungsmechanismus**

- VMM simuliert Unterbrechungen in den einzelnen VMs

Weiterleitung an VM

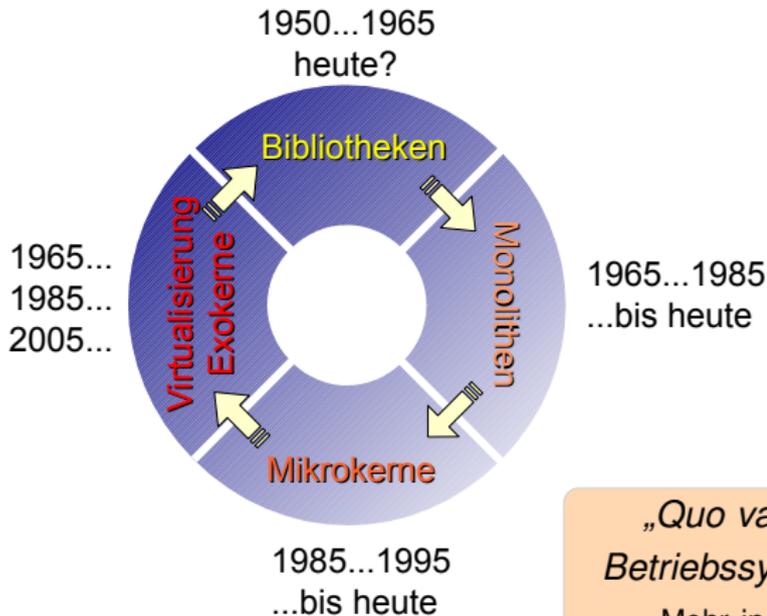
# Überblick: Paradigmen

*Back where we started?*



# Überblick: Paradigmen

*Back where we started?*



*„Quo vadis”,  
Betriebssysteme*

→ Mehr in VL 13!

# Agenda

---

Einführung

Geschichte, Mode und Trend

**Zusammenfassung**

Referenzen



- Betriebssysteme sind ein unendliches Forschungsthema
  - „alte“ Technologien (wie Virtualisierung oder Bibliotheken) finden immer wieder neue Einsatzgebiete
  - Hardwaretechnologie treibt die weitere Entwicklung
- Revolutionäre Neuerungen sind schwer durchzusetzen
  - Kompatibilität ist ein hohes Gut
    - Auf Anwendungsebene durch *Personalities* erreichbar
    - Neue Systeme scheitern jedoch meistens an fehlenden Treibern
  - Virtualisierte Hardware als Kompatibilitätsebene
- Die „ideale“ Architektur ist letztlich eine Frage der Anwendung!
  - Sensornetze, tief eingebettete Systeme  
Desktoprechner, Server, ...
  - Architektur → nichtfunktionale Eigenschaft des Betriebssystems



- [1] Mike Accetta, Robert Baron, David Golub u. a. „MACH: A New Kernel Foundation for UNIX Development“. In: *Proceedings of the USENIX Summer Conference*. USENIX Association, 1986, S. 93–113.
- [2] Paul Barham, Boris Dragovic, Keir Fraser u. a. „Xen and the Art of Virtualization“. In: *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)*. Bd. 37, 5. ACM SIGOPS Operating Systems Review. New York, NY, USA: ACM Press, Okt. 2003, S. 164–177.
- [3] Fred Brooks. *The Mythical Man Month*. Addison-Wesley, 1975. isbn: 0-201-00650-2.
- [4] Dawson R. Engler, M. Frans Kaashoek und James O'Toole. „Exokernel: An Operating System Architecture for Application-Level Resource Management“. In: *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP '95)*. ACM SIGOPS Operating Systems Review. ACM Press, 1995, S. 251–266.
- [5] Hermann Härtig, Michael Hohmuth, Jochen Liedtke u. a. „The Performance of  $\mu$ -Kernel-Based Systems“. In: *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP '97)*. New York, NY, USA: ACM Press, Okt. 1997. url: <http://l4ka.org/publications/>.

- [6] Gerwin Klein, Kevin Elphinstone, Gernot Heiser u. a. „seL4: formal verification of an OS kernel“. In: *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP '09)*. ACM Press. Big Sky, MT, USA: ACM Press, Okt. 2009, S. 207–220. isbn: 978-1-60558-752-3. doi: 10.1145/1629575.1629596.
- [7] Jochen Liedtke. „On  $\mu$ -Kernel Construction“. In: *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP '95)*. ACM SIGOPS Operating Systems Review. ACM Press, Dez. 1995.
- [8] Gerald J. Popek und Robert P. Goldberg. „Formal Requirements for Virtualizable Third Generation Architectures“. In: *Communications of the ACM* 17.7 (1974), S. 412–421. issn: 0001-0782. doi: <http://doi.acm.org/10.1145/361011.361073>.
- [9] Jim Smith und Ravi Nair. *Virtual Machines. Versatile Platforms for Systems and Processes*. Elsevier, 2005. isbn: 978-1558609105.