TECHNISCHE UNIVERSITÄT DRESDEN

Fakultät Informatik Institut für Systemarchitektur, Professur für Betriebssysteme

BETRIEBSSYSTEME UND SICHERHEIT

mit Material von Olaf Spinczyk, Universität Osnabrück

Betriebssystemarchitekturen

https://tud.de/inf/os/studium/vorlesungen/bs

HORST SCHIRMEIER

Inhalt

- Architektur-Grundbegriffe, Unterscheidungskriterien
- Bibliotheks-Betriebssysteme
- Monolithische Systeme
- Mikrokerne
- Exokerne und Virtualisierung
- Fazit

Literatur

Silberschatz, Chap. 23,
„Influential Operating Systems“

Tanenbaum, Kap. 1.7,
„Betriebssystemstrukturen“

Inhalt

- **Architektur-Grundbegriffe, Unterscheidungskriterien**

- Bibliotheks-Betriebssysteme
- Monolithische Systeme
- Mikrokerne
- Exokerne und Virtualisierung

- Fazit

Softwarearchitektur

- Definition:

> *Die grundlegende Organisation eines Systems, dargestellt durch dessen **Komponenten**, deren **Beziehungen** zueinander und zur Umgebung sowie den **Prinzipien**, die den Entwurf und die Evolution des Systems bestimmen.*

Quelle: Gesellschaft für Informatik e.V., https://gi.de/informatiklexikon/software-architektur

- Intuitiv ausgedrückt: „Kästchen mit Pfeilen“
- Beschreibt noch *nicht* den detaillierten Entwurf
- Es geht um die Zusammenhänge zwischen **Anforderungen** und dem zu konstruierenden **System**.

Unterscheidung von BS-Architekturen

- **Isolation**
- **Interaktionsmechanismen**
- **Interruptbehandlungs-Mechanismen**

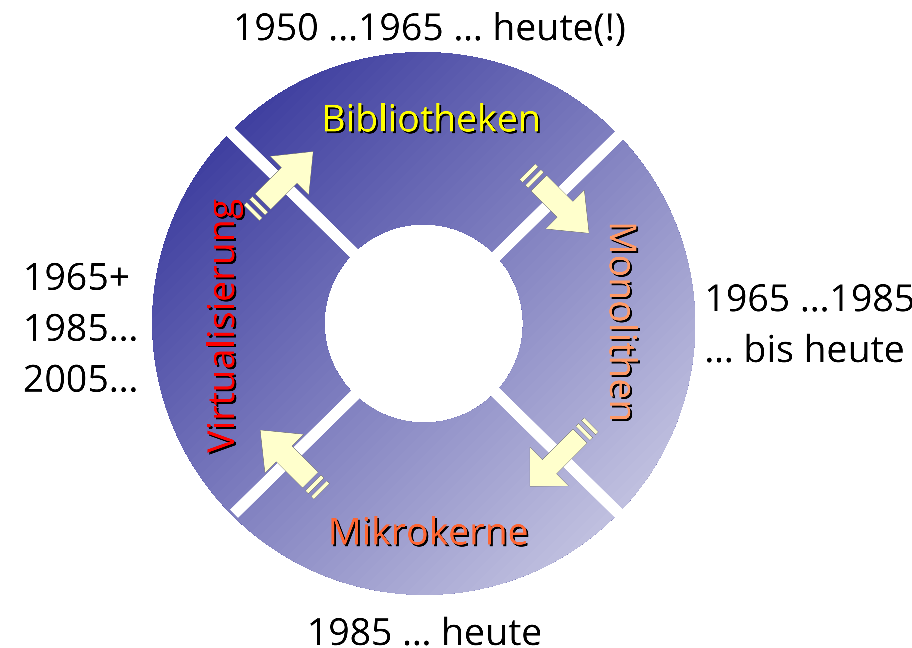
Technische Kriterien
(Prinzipien)

Beobachtbare Kriterien
(Anforderungen)

- **Anpassbarkeit**
  - Portierungen, Änderungen
- **Erweiterbarkeit**
  - Neue Funktionen und Dienste
- **Robustheit**
  - Verhalten bei Fehlern
- **Leistung**

Inhalt

- Architektur-Grundbegriffe, Unterscheidungskriterien
- Bibliotheks-Betriebssysteme
- Monolithische Systeme
- Mikrokerne
- Exokerne und Virtualisierung
- Fazit

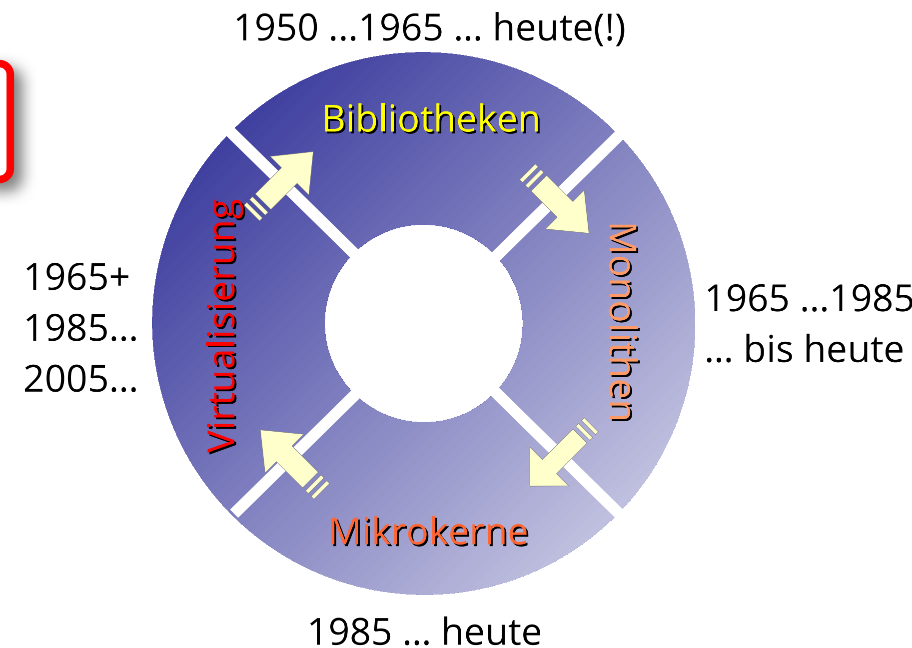


Inhalt

- Architektur-Grundbegriffe, Unterscheidungskriterien

- **Bibliotheks-Betriebssysteme**
- Monolithische Systeme
- Mikrokerne
- Exokerne und Virtualisierung

- Fazit



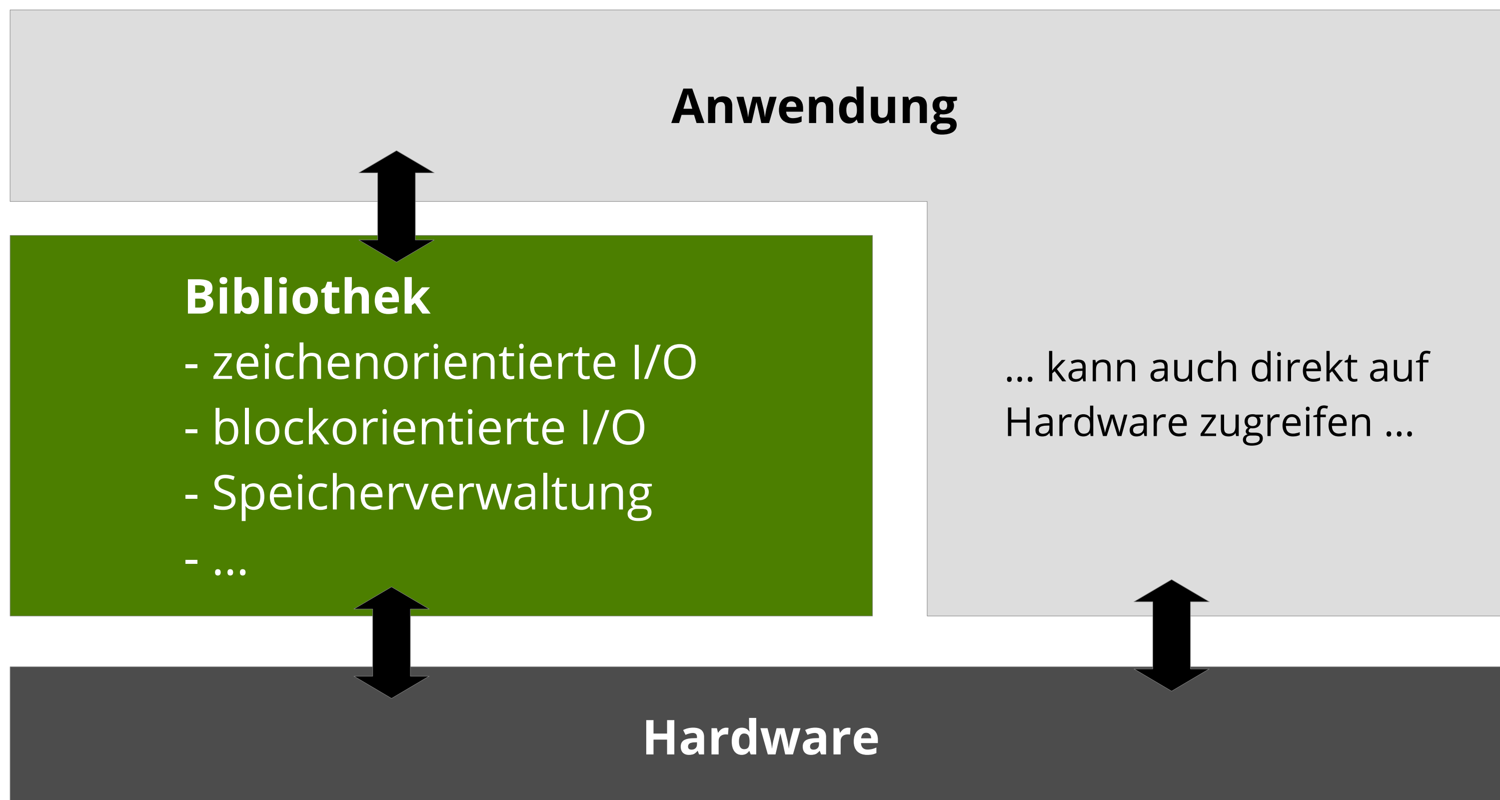
Entstehung von Betriebssystemen

- Erste Rechnersysteme besaßen keinerlei Systemsoftware
  - **Jedes Programm** musste die gesamte Hardware selbst ansteuern
  - Systeme liefen Operator-gesteuert im *Batch*-Betrieb
    - *Single tasking*, Lochkartenbetrieb
  - Peripherie war vergleichsweise einfach
    - Seriell angesteuerte Bandlaufwerke, Lochkartenleser und -schreiber, Drucker
- **Replikation von Code** zur Geräteansteuerung in jedem Anwendungsprogramm
  - Verschwendung von Entwicklungs- und Übersetzungszeit sowie Speicherplatz
  - Fehleranfällig

Bibliotheks-Betriebssysteme

- Zusammenfassung von häufig benutzten Funktionen zur Ansteuerung von Geräten in **Software-Bibliotheken (*Libraries*)**, die von allen Programmen genutzt werden konnten
  - ➔ Aufruf von Systemfunktionen als **normale Funktionsaufrufe**
- Bibliothek konnte im Speicher des Rechners bleiben
  - ➔ verringerte Programmladezeiten, **„Residenter Monitor“**
- Funktionen der Bibliothek waren **dokumentiert und getestet**
  - ➔ geringerer Entwicklungsaufwand für Anwendungsprogrammierer
- Fehler konnten **zentral** behoben werden
  - ➔ gesteigerte Zuverlässigkeit

Bibliotheks-Betriebssysteme



Bibliotheks-BS: Bewertung

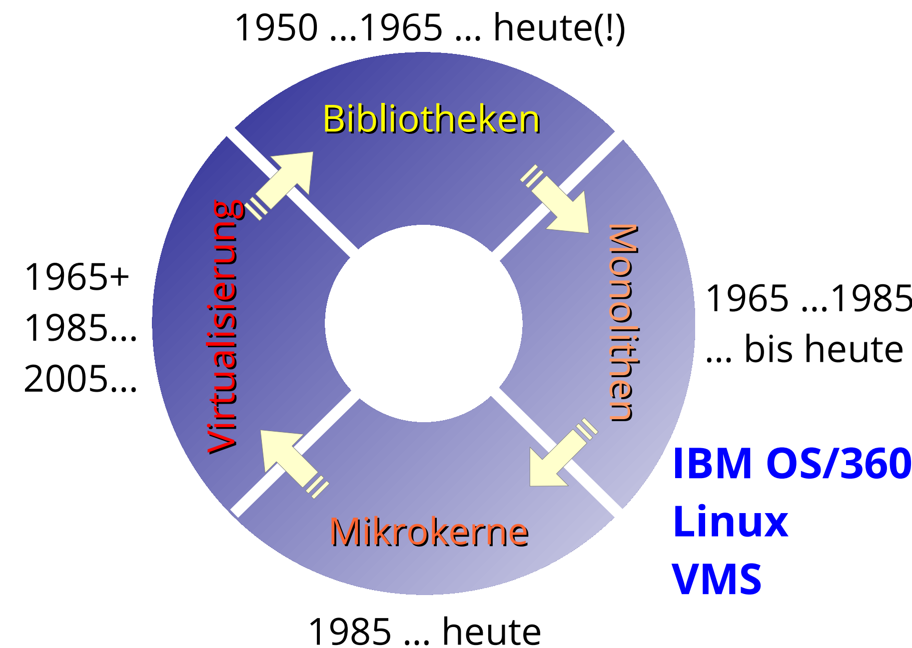
- **Isolation**
  - Ideal: ***Single Tasking***-System – aber hohe „Task-Wechselzeiten"
- **Interaktionsmechanismen**
  - Direkt (Funktionsaufrufe)
- **Interruptbehandlungs-Mechanismen**
  - Teilweise keinerlei Interrupts → ***Polling***
- **Anpassbarkeit**
  - Eigene Bibliotheken für jede Hardware-Architektur, keine Standards
- **Erweiterbarkeit**
  - Abhängig von Bibliotheks-Struktur: Globale Strukturen, **„Spaghetti-Code"**
- **Robustheit**
  - Direkte Kontrolle über die gesamte Hardware: Fehler → Systemstillstand
- **Leistung**
  - Sehr hoch, durch direktes Operieren auf der Hardware ohne Privilegierungsmechanismen

Bibliotheks-BS: Diskussion

- Produktive Nutzung der teuren Hardware nur zu einem relativ **kleinen Teil der Zeit**
  - Hoher Zeitaufwand für Wechsel der Anwendung
  - Warten auf Ein-/Ausgabe verschwendet unnötig Zeit, da nur ein „Prozess“ auf dem System läuft
- Lange Wartezeiten auf Ergebnisse
  - Warteschlange, Stapelverarbeitung
- Keine Interaktivität
  - Betrieb durch Operatoren, kein direkter Zugang zur Hardware
  - Programmabläufe nicht zur Laufzeit beeinflussbar

Inhalt

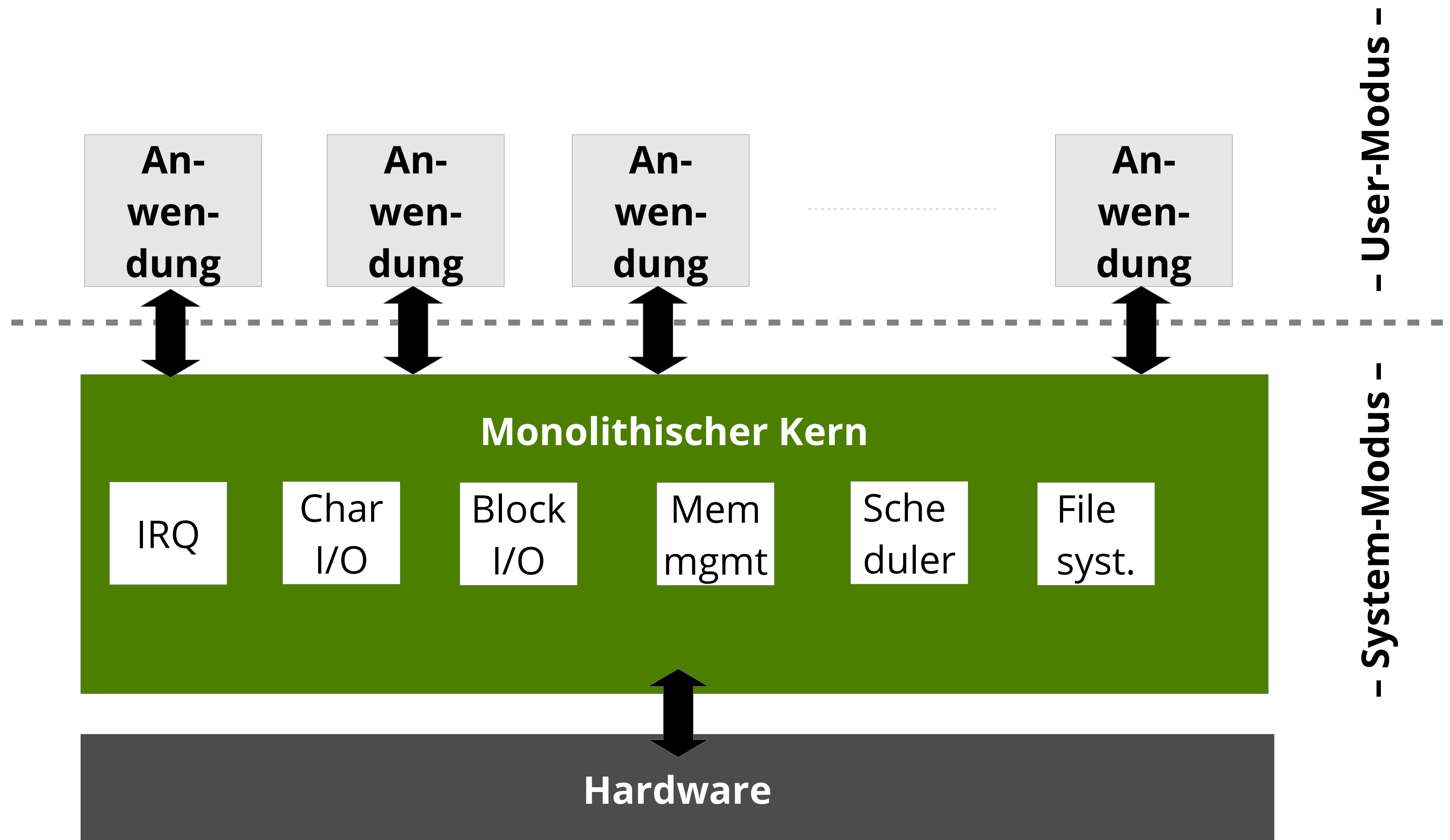
- Architektur-Grundbegriffe, Unterscheidungskriterien
- Bibliotheks-Betriebssysteme
- **Monolithische Systeme**
- Mikrokerne
- Exokerne und Virtualisierung
- Fazit



Monolithische Systeme

- Verwaltungssysteme für Rechnerhardware
  - Standardisiertes ***Accounting*** von Systemressourcen
- **Vollständige** Kontrolle der Hard- und Software
  - Anwendungen liefen nun unter Kontrolle des Systems
  - Damit erstmals Mehrprozess-Systeme realisierbar: ***Multiprogramming***
- Einführung eines Privilegiensystems
  - System-Modus und Anwendungs-Modus
  - Unterscheidung und Umschaltung von Hardware unterstützt
  - Direkter Hardware-Zugriff nur im System-Modus
- Aufruf von Systemfunktionen über spezielle Mechanismen (***Software Traps***)
  - Erfordert Kontextumschaltung und -sicherung

Monolithische Betriebssysteme



Monolithische Systeme: OS/360

- Eines der ersten monolithischen Systeme: IBM OS/360, 1966
- Ziel: gemeinsames *Batch*-BS für alle IBM-Großrechner
  - Leistung und Speicher differierten aber um Zehnerpotenzen!
- Verfügbarkeit des Systems in diversen Konfigurationen:
  - **PCP** (*Primary Control Program*): Einprozessbetrieb, kleine Systeme
  - **MFT** (*Multiprogramming with Fixed number of Tasks*): mittlere Systeme (256 kB RAM!), feste Speicherpartitionierung zwischen Prozessen, feste Anzahl an Tasks
  - **MVT** (*Multiprogramming with Variable number of Tasks*): high end, Swapping, optional *Time Sharing Option* (TSO) für interaktive Nutzung
- Richtungsweisende Eigenschaften
  - Hierarchisches Dateisystem
  - Prozesse können Unterprozesse erzeugen
  - MFT und MVT sind von API und ABI her kompatibel

z/OS unterstützt noch heute OS/360 Applikationen

Monolithische Systeme: OS/360

- Probleme im Bereich der Betriebssystem-Entwicklung
  - Fred Brooks „*The Mythical Man-Month*“ beschreibt die Probleme der Entwicklung des Systems
  - **Konzeptuelle Integrität**
    - **Separation von Architektur und Implementierung** war schwierig. Entwickler neigen dazu, die technischen Möglichkeiten auszuschöpfen und die Bedürfnisse der Benutzer zu beachten. → beeinflusst Verständlichkeit und damit Entwicklungs-Produktivität
  - ***„Second System Effect“***
    - Entwickler wollten alle Fehler des Vorgängersystems beseitigen und alle fehlenden Eigenschaften einbauen → wird nicht fertig.
  - Zu **komplexe Abhängigkeiten** zwischen Komponenten des Systems
    - Ab einer gewissen Codegröße ist eine bestimmte Menge an Fehlern unvermeidlich!
- Entwicklungen in der Softwaretechnik waren getrieben von Entwicklungen in Betriebssystemen

Monolithische Systeme: Unix

- Unix wurde als Betriebssystem für Rechner mit recht beschränkten Ressourcen entwickelt (Bell Labs)
  - Kernelgröße im Jahr 1979 (*7th Edition Unix*, PDP11): ca. 10.000 Zeilen Code (überschaubar, handhabbar!), compiliert ca. 50kB
  - Von ursprünglich 2-3 Entwicklern geschrieben
- Einführung von einfachen Abstraktionen
  - Jedes Objekt im System lässt sich als **Datei** darstellen
  - Jede Datei ist nur ein einfacher, unformatierter Strom von Bytes
  - Komplexe Funktionalität wird durch **Kombination einfacher Systemprogramme** realisiert (***Shell Pipelines***)
- Neues Ziel der Systementwicklung: **Portabilität**
  - Einfache Adaptierbarkeit des Systems auf unterschiedliche Hardware
  - Entwicklung von Unix in „C" — als domänenspezifische Sprache zur Betriebssystem-Entwicklung entworfen

Monolithische Systeme: Unix

- Weitere Entwicklung von Unix war nicht vorhersehbar
  - Systeme mit großem Adressraum (VAX; RISC-Systeme)
  - Der Kernel ist „mitgewachsen“ (System III, System V, BSD) – ohne wesentliche Strukturänderungen
  - Gleichzeitig wurden extrem komplexe Subsysteme integriert
    - TCP/IP war ungefähr so umfangreich wie der Rest des Kernels
- Linux orientiert(e) sich an der Struktur von System V Unix
- Einfluss im akad. Bereich: ***„Open Source“***-Politik der *Bell Labs*
  - Schwachpunkte von Unix führten zu neuen Forschungsansätzen
  - Viele Projekte (z.B. Mach) versuchten aber **kompatibel** zu bleiben

Monolithische BS: Bewertung

- **Isolation**
  - Keinerlei Isolation von Komponenten im Kernel-Modus, nur zwischen Anwendungsprozessen
- **Interaktionsmechanismen**
  - Direkte Funktionsaufrufe (im Kern), *Traps* (Anwendung–Kern)
- **Interruptbehandlungs-Mechanismen**
  - Direkte Behandlung von Hardware-Interrupts durch IRQ-Handler
- **Anpassbarkeit**
  - Änderungen einer Komponente beeinflussen andere Komponenten
- **Erweiterbarkeit**
  - Ursprünglich: Neuübersetzung erforderlich; Aktuell: Modulsystem
- **Robustheit**
  - Schlecht – Fehler in einer Komponente „tötet" gesamtes System
- **Leistung**
  - Hoch – wenige Kopieroperationen notwendig, da alle Kernkomponenten im gleichen Adressraum laufen. Die Ausführung eines Systemdienstes erfordert aber einen *Trap*.

Monolithische BS: Diskussion

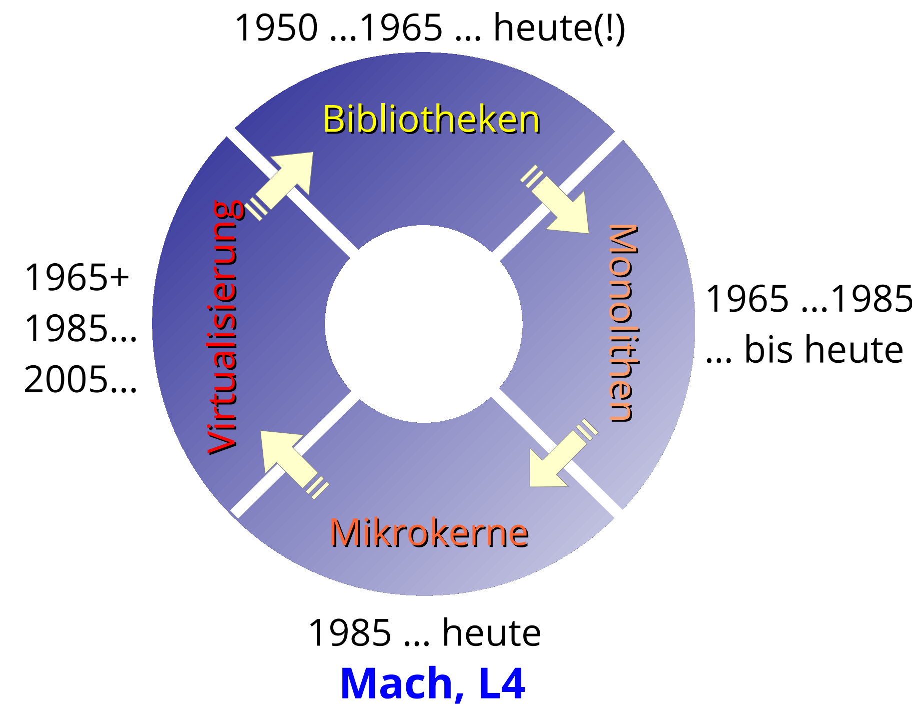
- Komplexe monolithische Kerne sind schwer handhabbar
  - Hinzufügen oder Abändern von Funktionalität betrifft oft mehr Module, als der Entwickler vorhergesehen hat
- Gemeinsamer Adressraum
  - Sicherheitsprobleme in einer Komponente (z.B. ***Buffer Overflow***) führen zur Kompromittierung des gesamten Systems
  - Viele Komponenten laufen überflüssigerweise im Systemmodus
- Eingeschränkte Synchronisationsmechanismen
  - Oft nur ein ***„Big Kernel Lock“***, d.h., nur ein Prozess kann zur selben Zeit im Kernel-Modus ausgeführt werden, alle anderen warten
  - Insbesondere bei Mehrprozessor-Systemen leistungsreduzierend

Inhalt

- Architektur-Grundbegriffe, Unterscheidungskriterien

- Bibliotheks-Betriebssysteme
- Monolithische Systeme
- **Mikrokerne**
- Exokerne und Virtualisierung

- Fazit



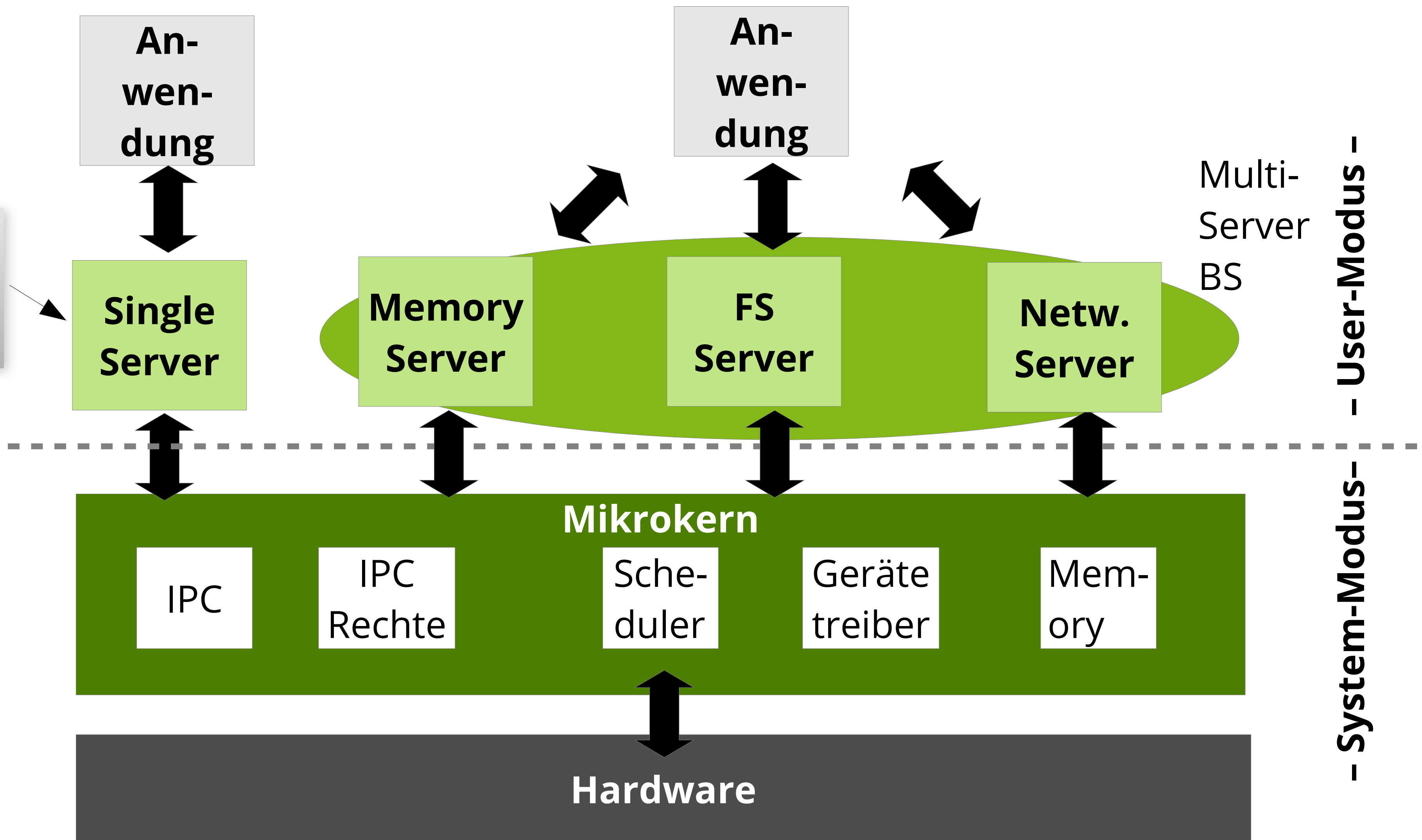
Mikrokern-Systeme

- Ziel: Verkleinerung der ***Trusted Computing Base***
  - Minimierung der im privilegierten Modus der CPU implementierten Funktionalität
  - Isolation der restlichen Komponenten voneinander im nichtprivilegierten Modus
- Prinzip der geringstmöglichen Privilegisierung
  - Systemfunktionen müssen nur so viele Privilegien besitzen, wie zur Ausführung ihrer Aufgabe erforderlich sind
- Aufruf von Systemfunktionen und Kommunikation zwischen Prozessen via Nachrichtenaustausch (IPC – ***Inter-Process Communication***)
- Reduzierte Funktionalität im Mikrokern
  - Geringere Codegröße
    (10.000 Zeilen C++ in L4 vs. 6,2 Millionen Zeilen C in Linux exkl. Gerätetreiber)
  - Ermöglicht Ansätze zur formalen Verifikation des Mikrokerns (seL4)

Mikrokerne erster Generation

- **Beispiel**: CMU Mach
- Ausgangspunkt: Separation der Fähigkeiten von (BSD) Unix in Funktionalität, die einen privilegierten Modus der CPU erfordern und solche, die ohne auskommen
- Ziel: Schaffung eines extrem portablen Systems
- Verbesserung der Unix-Konzepte
  - Neue Kommunikationsmechanismen via IPC und Ports
    - Ports sind sichere IPC-Kommunikationskanäle
    - IPC ist optional Netzwerk-transparent: Unterstützung für verteilte Systeme
  - Parallele Aktivitäten innerhalb eines Prozessadressraums
    - Unterstützung für *Threads* → neuer Prozessbegriff als „Container“
    - Bessere Unterstützung für Mehrprozessorsysteme

Mikrokern-System (1. Generation)



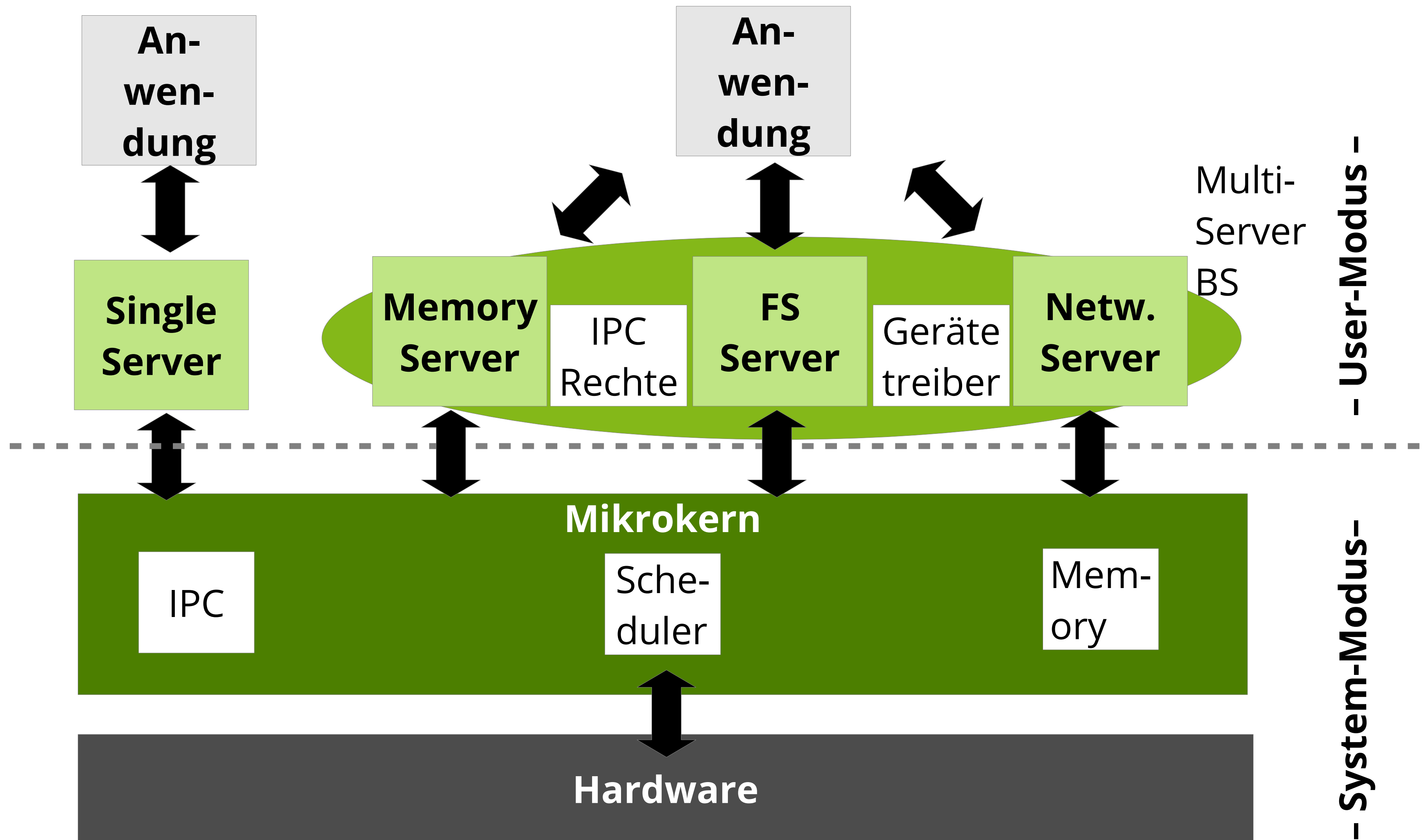
Mikrokerne erster Generation

- Probleme von Mach:
  - hoher *Overhead* für IPC-Operationen
    - Systemaufrufe **Faktor 10 langsamer** gegenüber monolithischem Kern
  - Ungünstige Entscheidung, welche Komponenten im Mikrokern implementiert werden sollten: große Code-Basis
    - Gerätetreiber und Rechteverwaltung für IPC im Mikrokern
  - Führte zu schlechtem Ruf von Mikrokernen allgemein
    - Einsetzbarkeit in der Praxis wurde bezweifelt
- Die Mikrokern-Idee war **Mitte der 90er Jahre tot**
- Praktischer Einsatz von Mach erfolgte meist in hybriden Systemen
  - Separat entwickelte Komponenten für Mikrokern und Server
  - Kolokation der Komponenten in einem Adressraum, Ersetzen von in-kernel IPC durch Funktionsaufrufe
  - Apple MacOS X: Mach-3-Mikrokern + FreeBSD

Mikrokerne zweiter Generation

- Ziel: Schwachpunkte der Mikrokerne 1. Generation ausmerzen
  - Beschleunigung von IPC-Operationen
  - Jochen Liedtke: L4 (1996)
    - Ein Konzept ist nur dann innerhalb des Mikrokerns toleriert, wenn eine Auslagerung aus dem Kern die Implementierung der für das System notwendigen Funktionalität verhindern würde.
- Vier grundlegende Mechanismen:
  - Abstraktion des **Adressraums**
  - Ein Modell für ***Threads***
  - **Synchrone Kommunikation** zwischen *Threads*
  - ***Scheduling***
- Viele von Mikrokernen der 1. Generation noch im Systemmodus implementierte Funktionalität ausgelagert
  - z.B. Überprüfung von IPC-Kommunikationsrechten

Mikrokern-System (2. Generation)



Mikrokern-BS: Bewertung

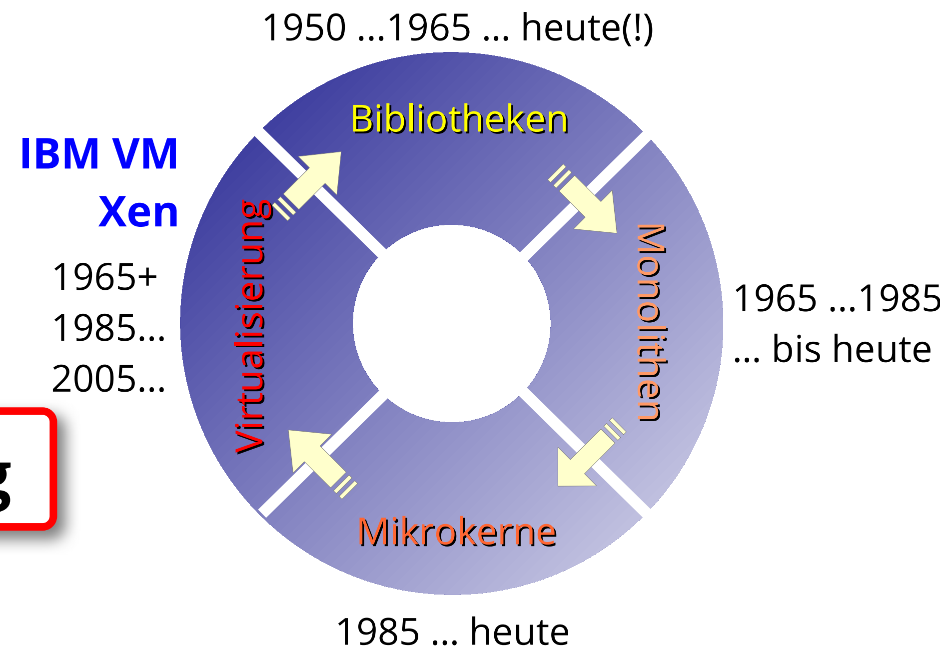
- **Isolation**
  - Sehr gut – separater Adressraum für jede Komponente
- **Interaktionsmechanismen**
  - Synchrone IPC
- **Interruptbehandlungs-Mechanismen**
  - Interrupts werden vom Mikrokern in IPC-Nachrichten umgesetzt
- **Anpassbarkeit**
  - Ursprünglich schwer anpassbar – x86-Assembler, später in C/C++
- **Erweiterbarkeit**
  - Sehr gut und einfach als Komponenten im *User*-Modus
- **Robustheit**
  - Gut – aber abhängig von Robustheit der Server
- **Leistung**
  - Im Wesentlichen abhängig von IPC-*Performance*

Inhalt

- Architektur-Grundbegriffe, Unterscheidungskriterien

- Bibliotheks-Betriebssysteme
- Monolithische Systeme
- Mikrokerne
- **Exokerne und Virtualisierung**

- Zusammenfassung und Ausblick



Exokerne: Noch kleiner als Mikro-...

Idee zur Vereinfachung:

- Die unterste Systemsoftwareschicht implementiert **keine Strategien oder Abstraktionen**. Sie virtualisiert auch keine Betriebsmittel.
- Einzige Aufgabe: **Ressourcenpartitionierung**
  - Jede Anwendung bekommt eigene Ressourcen zugewiesen
  - Die Einhaltung der Zuordnung wird durchgesetzt
  - Alles Weitere implementieren **anwendungsspezifische Bibliotheksbetriebssysteme** innerhalb der Ressourcen-Container bei Bedarf.
- Problem: Bibliotheksbetriebssysteme sind Exokern-spezifisch

Virtualisierung

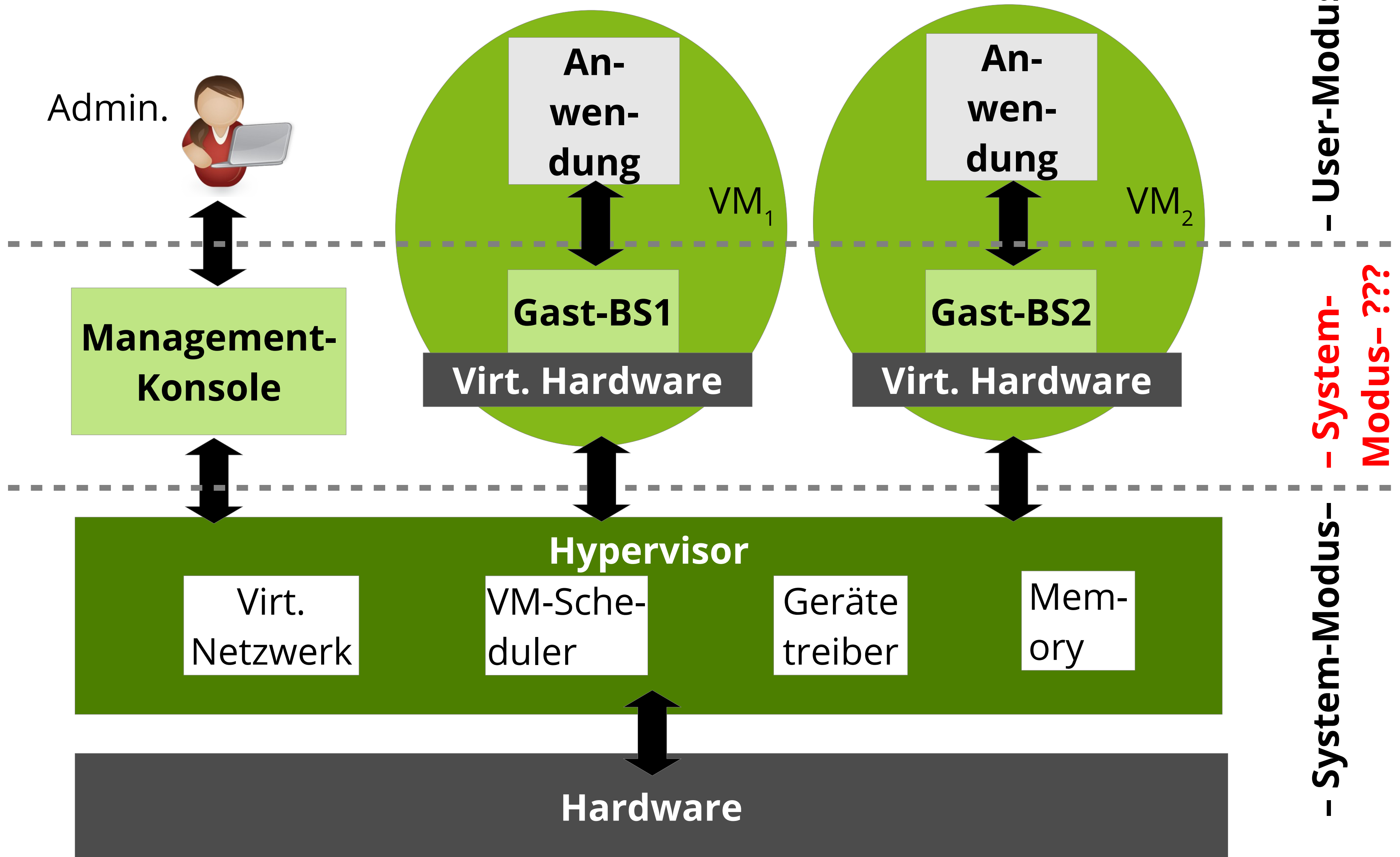
- Ziel: Isolation und *Multiplexing* von Ressourcen unterhalb der Betriebssystemebene
  - Gleichzeitige Verwendung mehrerer **Gast-Betriebssysteme**
- Virtuelle Maschinen (VMs) auf Systemebene virtualisieren die gesamten Hardware-Ressourcen, inkl.
  - Prozessor(en), Hauptspeicher und Massenspeicher-Ressourcen sowie Peripheriegeräten
- Ein ***Virtual Machine Monitor*** (VMM) oder ***Hypervisor*** ist die Softwarekomponente, die die virtuelle Maschinen-Abstraktion zur Verfügung stellt

Beispiel: IBM VM

- IBM360-Großrechner: viele verschiedene Betriebssysteme
  - DOS/360, MVS: *Batch*-orientierte *Library*-Betriebssysteme
  - OS/360+TSO: Interaktives Mehrbenutzer-System
  - Kundenspezifische Entwicklungen
- Problem: wie alle Systeme gleichzeitig verwenden?
  - Hardware war teuer (Millionen von US$)
  - BS erwartet Kontrolle über die gesamte Hardware
    → Schaffung dieser Illusion für jedes Betriebssystem
- Entwicklung der ersten Systemvirtualisierung „VM“ durch Kombination aus Emulation und Hardware-Unterstützung
  - Gleichzeitiger Betrieb von stapelverarbeitenden und interaktiven Betriebssystemen wurde ermöglicht

TECHNISCHE UNIVERSITÄT DRESDEN

Virtualisierung (mit Typ1-*Hypervisor*)



Hardware-unterstützte Virtualisierung

- Beispiel x86: Privilegierte Instruktionen in Ring 0 abfangbar
  - Intel „Vanderpool“ (**Intel VT-x**), AMD „Pacifica“ (**AMD-V**)
  - Zusätzlicher logischer Privilegmodus: oft „Ring -1“ genannt
- Gast-Betriebssystem-Kernel läuft wie bisher in Ring 0
- „Kritische“ Instruktionen in Ring 0
  - *Trap* in den *Hypervisor*
  - Dieser emuliert kritische Instruktionen
  - oder hält zugehöriges Betriebssystem an
- Verwendung mehrerer gänzlich unveränderter Betriebssystem-Instanzen gleichzeitig auf einem System möglich
  - Peripherie der jeweiligen VMs muss weiterhin emuliert werden, da die einzelnen BS keine Kenntnis voneinander haben

Paravirtualisierung

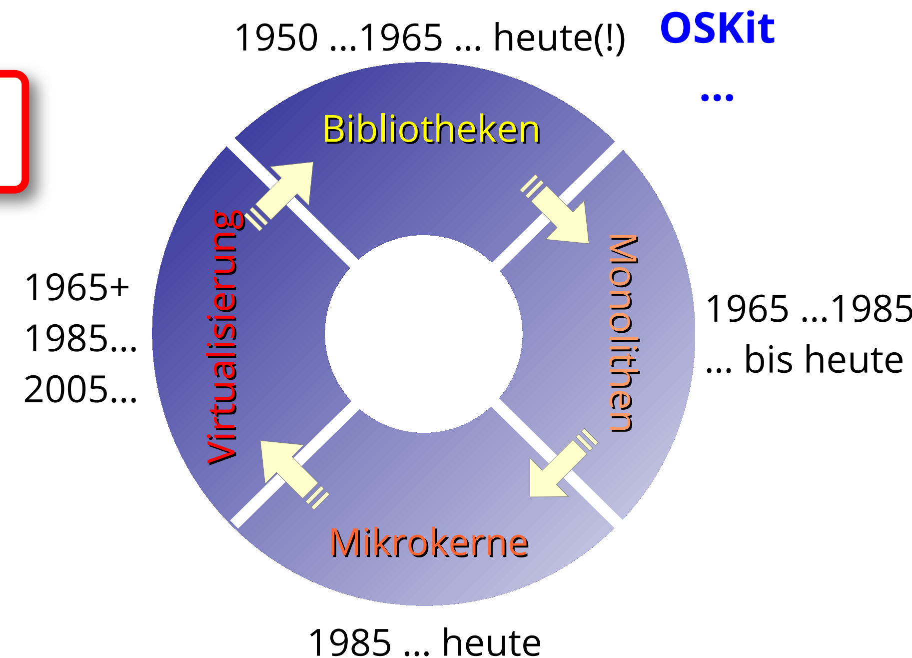
- Anwendungen der virtualisierten BS laufen unverändert, **spezieller Kernel** in virtualisierten BS erforderlich
- Gast-Kernel läuft (auf x86) in einem Ring > 0 (z.B. Ring 3)
  - also im nicht-System-Modus
- Umsetzung:
  - Ersetzung „kritischer" Instruktionen (Interrupt-Behandlung, Speicherverwaltung etc.) im Gast-Kernel durch ***Hypercalls*** (explizite Aufrufe des Hypervisors)
    - VMware-Ansatz: Binärcode-Anpassung beim Laden des Gast-BS
    - Xen-Ansatz: Änderung des Gast-BS-Quelltextes
  - Leistungverbesserung: *Hypercalls* auch für Zugriff auf Peripherie und Netzwerk – vermeidet langsame Hardware-Emulation

Virtualisierung: Bewertung

- **Isolation**
  - Sehr gut – aber grobgranular (zwischen VMs)
- **Interaktionsmechanismen**
  - Kommunikation zwischen VMs nur via TCP/IP (virtuelle Netzwerkkarten!)
- **Interruptbehandlungs-Mechanismen**
  - Weiterleitung von IRQs an Gast-Kernel in der VM (simulierte HW-Interrupts)
- **Anpassbarkeit**
  - Spezifisch zu CPU-Typ erforderlich, Paravirtualisierung aufwendig
- **Erweiterbarkeit**
  - Schwierig – in üblichen VMMs nicht vorgesehen
- **Robustheit**
  - Gut – aber grobgranular (ganze VMs sind von Abstürzen betroffen)
- **Leistung**
  - Gut – Verlust 5-10% vs. Ausführung direkt auf der Hardware

Inhalt

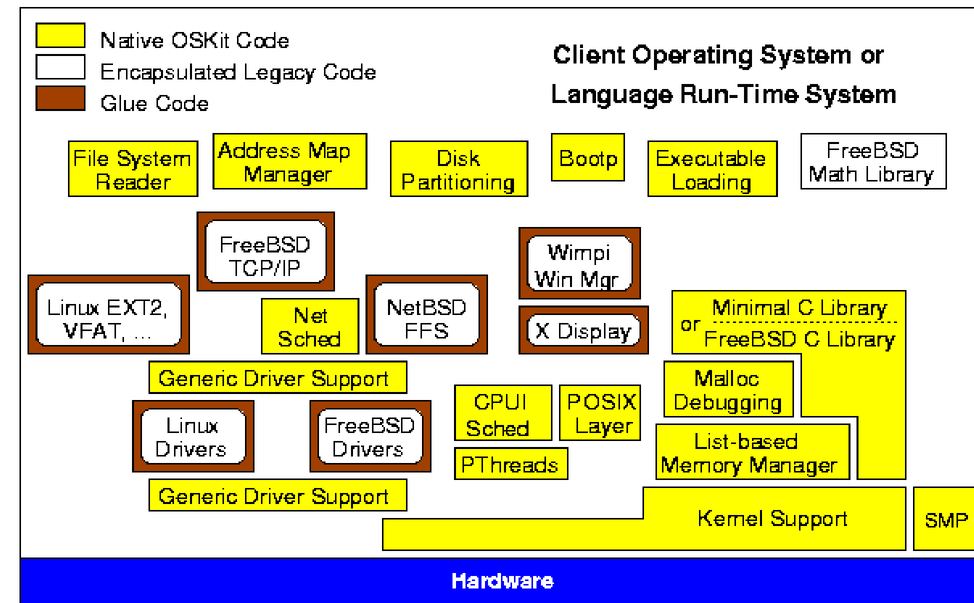
- Architektur-Grundbegriffe, Unterscheidungskriterien
- **Bibliotheks-Betriebssysteme**
- Monolithische Systeme
- Mikrokerne
- Exokerne und Virtualisierung
- Fazit



wieder zurück am Anfang?!

Bibliotheken von BS-Funktionalitäten

- ***„Unikernels“*** dienen der effizienten Ausführung einer Applikation innerhalb einer VM
  - mirageOS, Mini-OS, Unikraft, ...
- Beispiel: Utah OSKit
  - „Best Of“ verschiedener Betriebssystem-Komponenten
  - An gemeinsamen Standard angepasste Schnittstellen
  - Sprachunterstützung (Interface-Generator) erleichtern die Integration

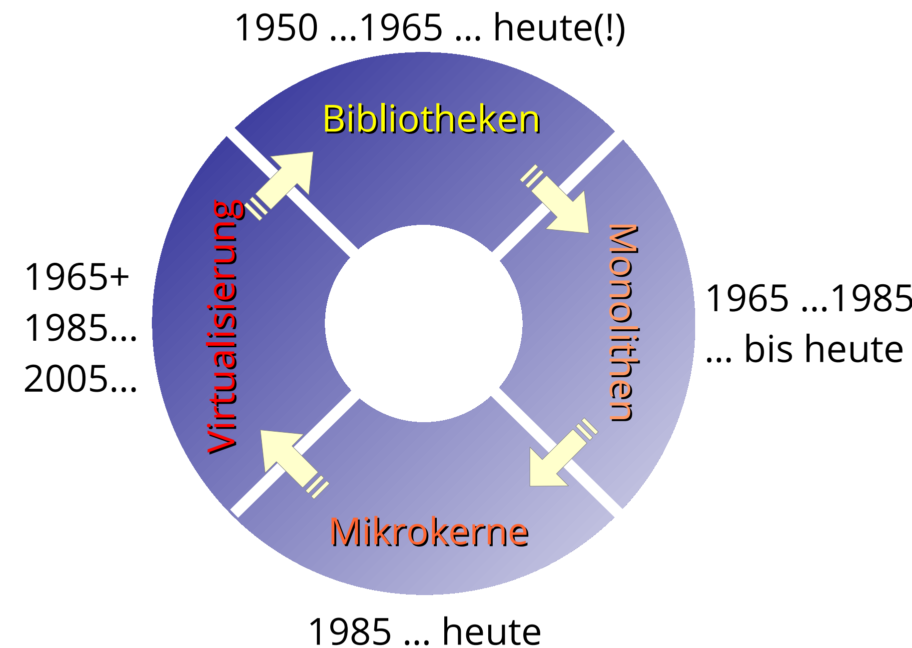


Inhalt

- Architektur-Grundbegriffe, Unterscheidungskriterien

- Bibliotheks-Betriebssysteme
- Monolithische Systeme
- Mikrokerne
- Exokerne und Virtualisierung

- **Fazit**



Betriebssystem-Architektur: Fazit

- BS-Architekturen weiterhin aktuelles Forschungsthema
  - „alte“ Technologien wie Virtualisierung finden neue Einsatzgebiete, zum Beispiel im *Cloud-Computing*
  - Hardware und Anwendungen verändern sich kontinuierlich, z. B.
    - Energiesparen (*Energy Harvesting*)
    - Skalierbarkeit (*Multi-/Manycore*-Prozessoren)
    - Umgang mit Heterogenität (ARM big.LITTLE, GPUs, ...)
    - Adaptierbarkeit (Mobile Systeme, ressourcenbeschränkte Systeme)
    - Persistente Hauptspeicher (TI FRAM, Intel DCPMMs)
- Kompatibilitätsanforderungen und hohe Entwicklungskosten verhindern schnelle Akzeptanz neuer Entwicklungen
  - Virtualisierung als Kompatibilitätsebene