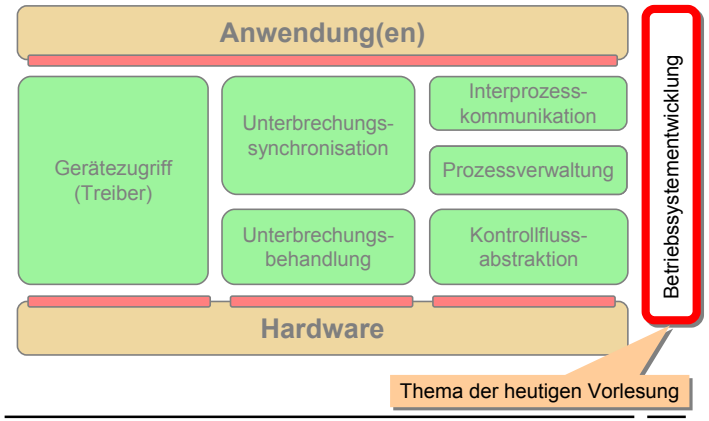
Betriebssysteme (BS)

Architekturen

Daniel Lohmann

Lehrstuhl für Informatik 4
Verteilte Systeme und Betriebssysteme

Überblick: Vorlesungen



Agenda

- Bewertungskriterien für Betriebssysteme
- Paradigmen der Betriebssystementwicklung
- Bibliotheks-Betriebssysteme
- Monolithische Systeme
- Mikrokerne
- Exokerne und Virtualisierung
- Fazit

Agenda

- **Bewertungskriterien für Betriebssysteme**
- **Paradigmen der Betriebssystementwicklung**
- Bibliotheks-Betriebssysteme
- Monolithische Systeme
- Mikrokerne
- Exokerne und Virtualisierung
- Fazit

Bewertungskriterien für Betriebssysteme

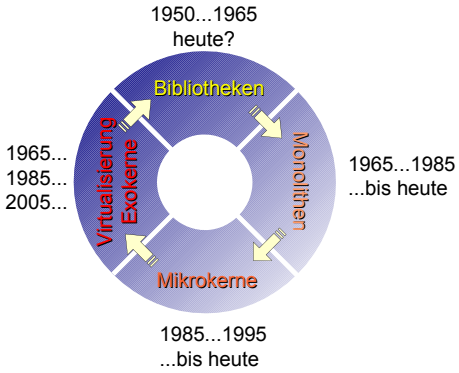
- Anwendungsorientierte Kriterien
  - **Portabilität**
  - **Erweiterbarkeit**
  - **Robustheit**
  - **Leistung**
- Technische Kriterien (Architektureigenschaften)
  - **Isolationsmechanismus**
  - **Interaktionsmechanismus**
  - **Unterbrechungsmechanismus**

Bewertungskriterien für Betriebssysteme

- 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?*

Betriebssystem-Geschichte

Paradigmen der Betriebssystementwicklung

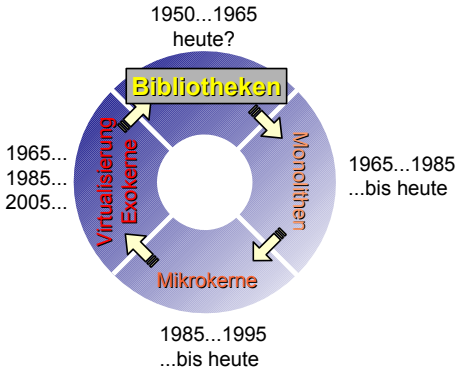


Agenda

- Bewertungskriterien für Betriebssysteme
- Paradigmen der Betriebssystementwicklung
- **Bibliotheks-Betriebssysteme**
- Monolithische Systeme
- Mikrokerne
- Exokerne und Virtualisierung
- Fazit

Überblick: Paradigmen

Bibliotheks-Betriebssysteme als einfache Infrastrukturen



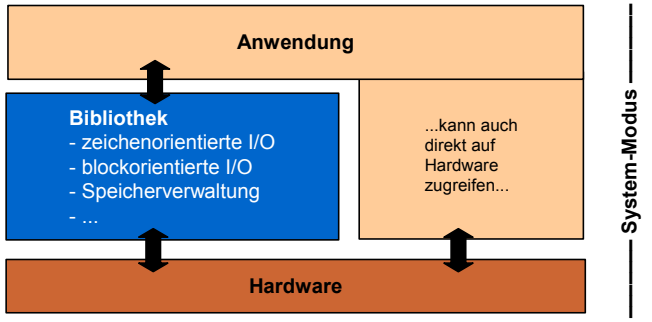
Entstehung von Bibliotheks-Betriebssystemen

- 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

Bibliotheks-Betriebssysteme

- 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 (von Anwendungen)
    → verringerte Ladezeit (von Anwendungen)
  - Fehler konnten von Experten zentral behoben werden
    → erhöhte Zuverlässigkeit

Bibliotheks-Betriebssysteme



Bibliotheks-Betriebssysteme: Bewertung

- Anwendungsorientierte Kriterien
  - **Portabilität** — **gering**
    - keine Standards, eigene Bibliotheken für jede Architektur
  - **Erweiterbarkeit** — **mäßig**
    - theoretisch gut, in der Praxis oft „Spaghetti-Code“
  - **Robustheit** — **sehr hoch**
    - *single tasking*, Kosten für Taskwechsel sehr hoch
  - **Leistung** — **sehr hoch**
    - direktes Operieren auf der Hardware, keine Privilegebenen
- Technische Kriterien
  - **Isolationsmechanismus** — **nicht notwendig**
    - Anwendung = System
  - **Interaktionsmechanismus** — **Funktionsaufrufe**
    - Betriebssystem = Bibliothek
  - **Unterbrechungsmechanismus** — **oft nicht vorhanden**
    - Kommunikation mit Geräten über *polling*

Bibliotheks-Betriebssysteme: Probleme

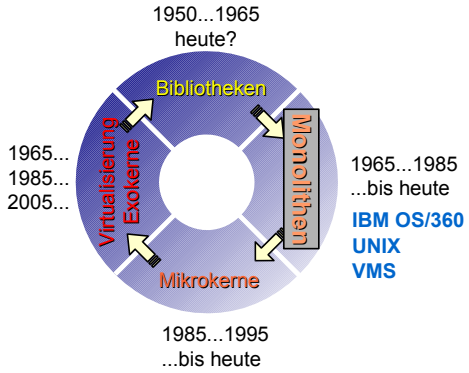
- Teure Hardware wird nicht optimal ausgelastet
  - Hoher Zeitaufwand beim Wechseln der Anwendung
  - Warten auf Ein-/Ausgabe verschwendet unnötig CPU-Zeit
- Organisatorische Abläufe sehr langwierig
  - Stapelbetrieb, Warteschlangen
  - von der Abgabe eines Programs bis zum Erhalt der Ergebnisse vergehen 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 parametrisierbar

Agenda

- Bewertungskriterien für Betriebssysteme
- Paradigmen der Betriebssystementwicklung
- Bibliotheks-Betriebssysteme
- **Monolithische Systeme**
- Mikrokerne
- Exokerne und Virtualisierung
- Fazit

Überblick: Paradigmen

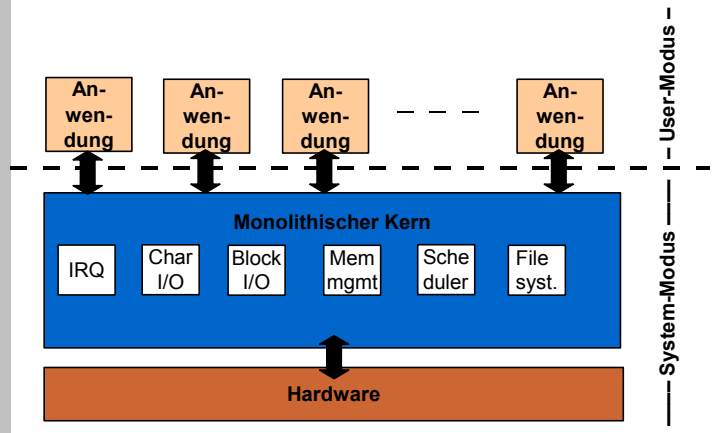
Monolithen als Herrscher über das System



Monolithische Betriebssysteme

- Motivation: **Mehrprogrammbetrieb**
- Problem: **Isolation**
- Lösung: BS als **kontrollierende Instanz**
  - Anwendungen laufen unter der Kontrolle des Systems
  - Damit erstmals sinnvoll Mehrprozess-Systeme realisierbar.
- Einführung eines Privilegiensystems
  - Systemmodus vs. Anwendungsmodus
  - Direkter Hardware-Zugriff nur im Systemmodus
    - → Gerätetreiber gehören zum System
- Einführung neuer Hard- und Software-Mechanismen
  - *Traps* in den Kern
  - Kontextumschaltung und -sicherung
  - *Scheduling* der Betriebsmittel

Monolithische Betriebssysteme

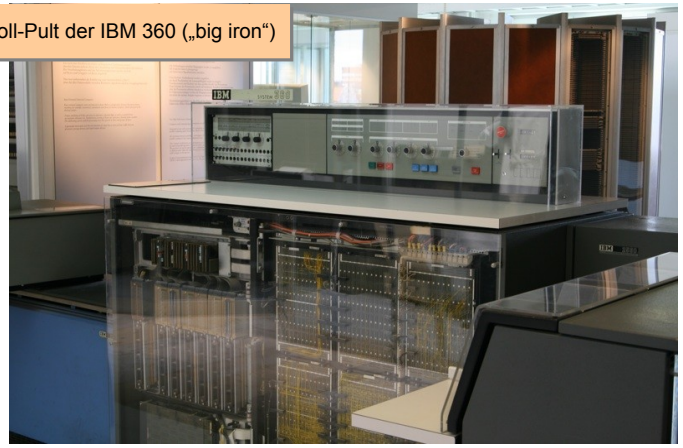


Monolithische Systeme: IBM OS/360

- 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: OS/360

Kontroll-Pult der IBM 360 („big iron“)



Monolithische Systeme: 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“ **[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 bleibt die Anzahl der Fehler konstant
- Treibender Faktor für die „Geburt der Softwaretechnik“

Monolithische Systeme: Bell Labs/AT&T UNIX

- 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

Historische PDP11

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

Monolithische Betriebssysteme: Bewertung

- Anwendungsorientierte Kriterien
  - **Portabilität** — **hoch**
    - dank „C“ konnte UNIX einfach portiert werden
  - **Erweiterbarkeit** — **mäßig**
    - von Neukompilierung → Modulkonzept, viele interne Abhängigkeiten
  - **Robustheit** — **mäßig**
    - Anwendungen isoliert, nicht jedoch BS-Komponenten (Treiber!)
  - **Leistung** — **hoch**
    - Nur Betreten / Verlassen des Kerns ist teuer
- Technische Kriterien
  - **Isolationsmechanismus** — **Privilegebenen, Addressräume**
    - Jede Anwendung bekommt einen Addressraum, Kern läuft auf Systemebene
  - **Interaktionsmechanismus** — **Funktionsaufrufe, *Traps***
    - Anwendung → Kern durch *Traps*, innerhalb des Kerns durch Funktionsaufrufe
  - **Unterbrechungsmechanismus** — **Bearbeitung im Kern**
    - interne Unterteilung in UNIX: *bottom half*, *top half*

Monolithische Betriebssysteme: Probleme

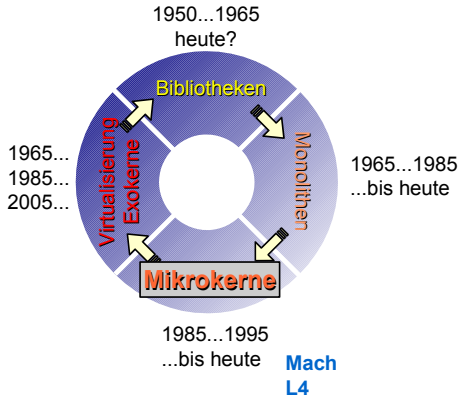
- Monolithen sind schwer handhabbar
  - Hinzufügen oder Abändern von Funktionalität betrifft oft mehr Module, als der Entwickler vorhergesehen hat
- Eingeschränkte Synchronisationsmechanismen
  - Oft nur ein *„Big Kernel Lock“*, d.h. nur ein Prozess kann zur selben Zeit im Kernmodus ausgeführt werden, 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

Agenda

- Bewertungskriterien für Betriebssysteme
- Paradigmen der Betriebssystementwicklung
- Bibliotheks-Betriebssysteme
- Monolithische Systeme
- **Mikrokerne**
- Exokerne und Virtualisierung
- Fazit

Überblick: Paradigmen

Mikrokerne als Reduktion auf das Notwendige



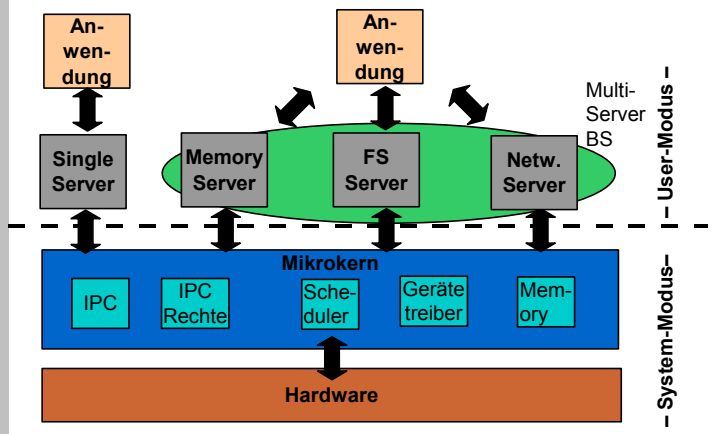
Mikrokern-Betriebssysteme

- Ziel: Reduktion der *Trusted Computing Base* (TCB)
  - Minimierung der im privilegierten Modus der CPU 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++ Linux: 1 Mloc C (ohne Treiber)
  - Ermöglicht Ansätze zur formalen Verifikation des Mikrokerns

Mikrokerne erster Generation: CMU Mach

- 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 → 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

Mikrokerne erster Generation



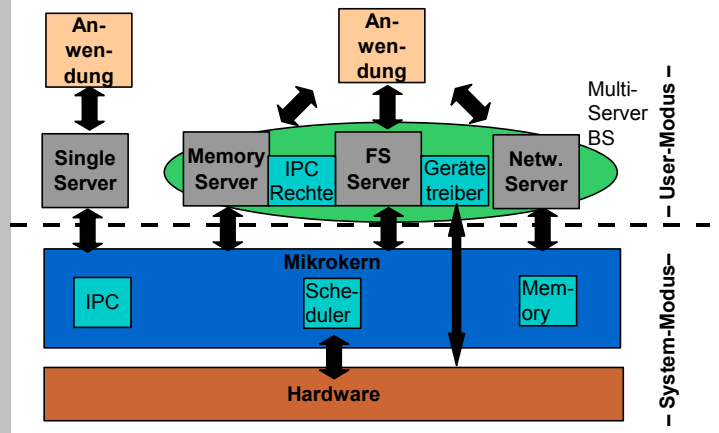
Mikrokerne erster Generation: Probleme

- 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 Mikrokern
    - die eigentlichen Probleme nicht gelöst
  - Führte zu schlechtem Ruf von Mikrokernen allgemein
    - Einsetzbarkeit in der Praxis wurde bezweifelt
- Die Mikrokern-Idee galt Mitte der 90er Jahre als tot
  - Praktischer Einsatz von Mach erfolgte nur in hybriden Systemen
    - Separat entwickelte Komponenten für Mikrokern und Server
    - Kolokation der Komponenten in einem Adressraum, Ersetzen von in-kernel IPC durch Funktionsaufrufe
    - Beispiel Apple OS X: Mach 3 Mikrokern + FreeBSD

Mikrokerne zweiter Generation: L4

- 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

Mikrokern-System (2. Generation)



Mikrokerne: Bewertung

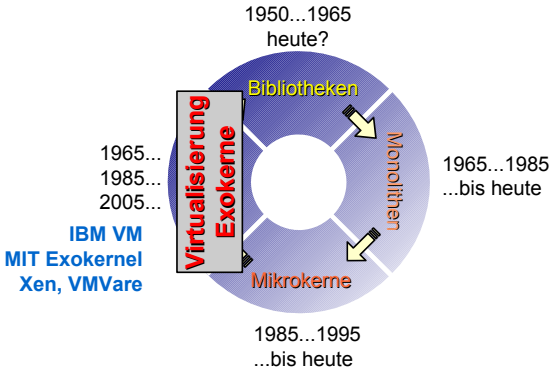
- Anwendungsorientierte Kriterien
  - **Portabilität** — **mäßig**
    - ursprünglich rein in Assembler, aktuell in C++ entwickelt
  - **Erweiterbarkeit** — **sehr hoch**
    - durch neue Server im Benutzermodus, auch zur Laufzeit
  - **Robustheit** — **sehr hoch**
    - durch strikte Isolierung
  - **Leistung** — **mäßig – gut**
    - IPC-Performance ist ***der*** kritsche Faktor
- Technische Kriterien
  - **Isolationsmechanismus** — **Adressräume**
    - Ein Adressraum pro Anwendung, ein Adressraum pro Systemkomponente
  - **Interaktionsmechanismus** — **IPC**
    - Anwendungen und Systemkomponente interagieren über Nachrichten
  - **Unterbrechungsmechanismus** — **IPC an Server-Prozess**
    - Unterbrechungsbehandlung erfolgt durch Faden im Benutzermodus

Agenda

- Bewertungskriterien für Betriebssysteme
- Paradigmen der Betriebssystementwicklung
- Bibliotheks-Betriebssysteme
- Monolithische Systeme
- Mikrokerne
- **Exokerne und Virtualisierung**
- Fazit

Überblick: Paradigmen

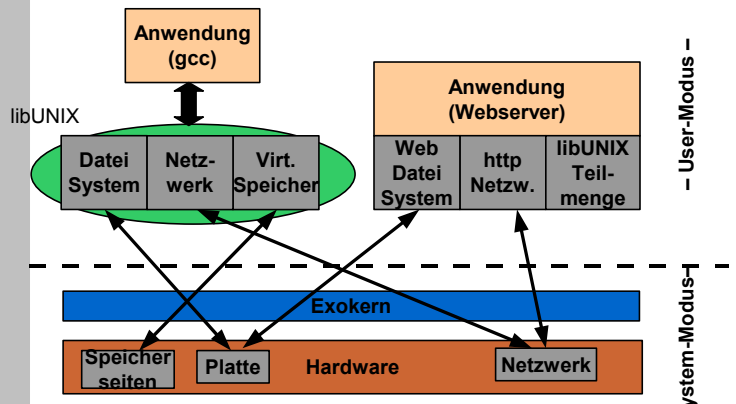
Exokernel und Virtualisierung als weitere Reduktion



Exokerne: MIT exokernel

- 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 (Mikrokerne) oder weiterer Abstraktionen (Monolithen)
  - Anwendungen können die für sie idealen Abstraktionen, Komponenten und Strategien verwenden

Exokerne



Exokerne: Bewertung

- Anwendungsorientierte Kriterien
  - **Portabilität** — **sehr hoch**
    - Exokerne sind sehr klein
  - **Erweiterbarkeit** — **sehr hoch**
    - aber auch erforderlich – der Exokern stellt kaum Funktionalität bereit
  - **Robustheit** — **gut**
    - Schutz wird durch den Exokern bereitgestellt
  - **Leistung** — **sehr gut**
    - Anwendungen operieren nahe an der Hardware, wenig Abstraktionsebenen
- Technische Kriterien
  - **Isolationsmechanismus** — **Adressräume**
    - Ein Adressraum pro Anwendung + von ihr gebrauchter Systemkomponenten
  - **Interaktionsmechanismus** — **nicht vorgegeben**
    - wird von der Anwendung bestimmt
  - **Unterbrechungsmechanismus** — **nicht vorgegeben**
    - Exokern verhindert allerdings die Monopolisierung der CPU

Exokerne: Probleme

- 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 sind noch offen

Virtualisierung

- 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 Betriebssysteme) müssen nicht angepasst werden

Beispiel: IBM VM

- Für IBM 360-Großrechner existierten eine Menge verschiedener Betriebssysteme
  - DOS/360, MVS: Stapel-orientierte Bibliotheks-Betriebssysteme
  - OS/360: Interaktives Mehrbenutzersystem
  - Kundenspezifische Entwicklungen
- Problem: wie kann man Anwendungen für all diese Systeme gleichzeitig verwenden?
  - Hardware war teuer (Millionen von US$)
- Entwicklung der ersten Systemvirtualisierung „VM“ durch Kombination aus Emulation und Hardware-Unterstützung
  - Harte Partionierung der Betriebsmittel
  - Gleichzeitiger Betrieb von stapelverarbeitenden und interaktiven Betriebssystemen wurde ermöglicht

Virtualisierungskriterien (Popek und Goldberg 1974)

- Um virtualisierbar zu sein, muss eine *Instruction Set Architecture* (ISA) folgende Kriterien erfüllen:
  - Äquivalenz
    - Eine Instruktion, die unter einem VMM ausgeführt wird, soll ein Verhalten zeigen, das essentiell identisch zu dem Verhalten bei direkter Ausführung auf einer äquivalenten Maschine ist.
  - Ressourcenverwaltung
    - Der VMM muss die vollständige Kontrolle über alle virtualisierten Ressourcen besitzen.
  - Effizienz
    - Ein hoher Anteil an Maschineninstruktionen muss ohne Intervention des VMM ausgeführt werden können.
- IA-32 hat Probleme mit Äquivalenz
  - Nicht alle privilegierten Befehle *trappen* im Benutzermodus (Ring 3)
  - Aktueller Modus (Ring 0,1,2,3) ist auslesbar

Ansatz: x86 Paravirtualisierung

- VMM läuft auf Ring 0
  - also mit Systemprivilegien
  - hat volle Kontrolle über alle virtualisierbaren Ressourcen
- VMs laufen auf Ring 3
  - also ohne Systemprivilegien
- Gast-Betriebssystem (in der VM) wird modifiziert
  - „kritische Befehle“ werden ersetzt
    - so dass Äquivalenz sichergestellt ist
    - Ersetzung zur Übersetzungszeit (Xen) oder zur Laufzeit (VMWare)
  - Ringmodell wird durch VMM simuliert
    - z.B. durch Adressräume
    - die meisten Betriebssysteme verwenden eh nur Ringe 0 und 3

Beispiel: Xen

- Ähnliche Problematik wie IBM in den 60er Jahren
  - Ablauf mehrerer Betriebssystem-Instanzen gleichzeitig
  - Serverkonsolidierung, Kompatibilität zu Anwendungen
- PC-Hardware ist mittlerweile leistungsfähig genug
  - Ausreichend CPU-Leistung und Hauptspeicher für mehrere Instanzen von Betriebssystemen
- IA-32 unterstützte aber nicht die Virtualisierungskriterien von Popek und Goldberg
  - Paravirtualisierung erforderlich
  - Windows-Kernel-Sourcecode ist aber nicht (frei) verfügbar
  - Beschränkung auf Open Source OS
- Neue IA-32 CPUs unterstützen Hardwarevirtualisierung
  - AMD: VT-x „Pacifica“
  - Intel: VT „Vanderpool“

Virtualisierung: Bewertung

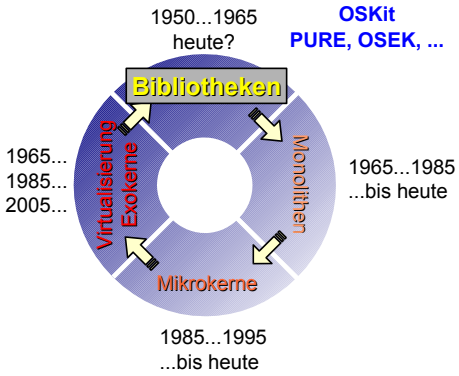
- Anwendungsorientierte Kriterien
  - **Portabilität** — **gering**
    - sehr hardware-spezifisch, Paravirtualisierung aufwändig
  - **Erweiterbarkeit** — **keine**
    - in den üblichen VMMs bislang nicht vorgesehen
  - **Robustheit** — **gut**
    - grobgranular auf der Ebene von VMs
  - **Leistung** — **mäßig – gut**
    - stark abhängig vom Einsatzszenario (CPU-lastig, IO-lastig, ...)
- Technische Kriterien
  - **Isolationsmechanismus** — **VM, Paravirtualisierung**
    - Jede Instanz bekommt einen eigenen Satz an Hardwaregeräten
  - **Interaktionsmechanismus** — **nicht vorgesehen**
    - Anwendungen in den VMs kommunizieren miteinander über TCP/IP
  - **Unterbrechungsmechanismus** — **Weiterleitung an VM**
    - VMM simuliert Unterbrechungen in den einzelnen VMs

Agenda

- Bewertungskriterien für Betriebssysteme
- Paradigmen der Betriebssystementwicklung
- Bibliotheks-Betriebssysteme
- Monolithische Systeme
- Mikrokerne
- Exokerne und Virtualisierung
- **Fazit**

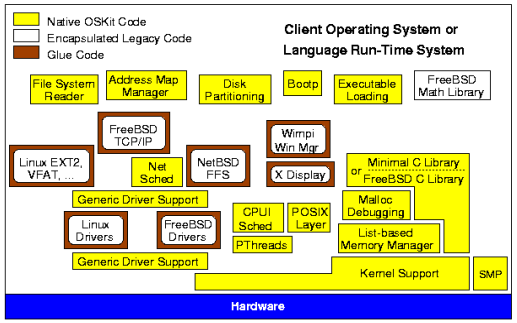
Überblick: Paradigmen

„Back where we started?“



BS-Baukästen

- Beispiel: Utah OSKit
  - *„best of“* verschiedener Betriebssystem-Komponenten
  - An gemeinsamen Standard angepasste Schnittstellen
  - Verbindung durch (automatisch generierten) *„glue code“*



Betriebssystem-Architektur: Fazit

- Betriebssysteme sind ein unendliches Forschungsthema!
  - „alte“ Technologien wie Virtualisierung finden 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
- Anforderungen an Systeme (und als Folge deren Architektur) sind abhängig von der Anwendung
  - Sensornetze, tief eingebettete Systeme
  - Desktoprechner, Server
  - High-Performance Computing