

## 2. Foliensatz Betriebssysteme

Prof. Dr. Christian Baun

Frankfurt University of Applied Sciences  
(1971–2014: Fachhochschule Frankfurt am Main)  
Fachbereich Informatik und Ingenieurwissenschaften  
[christianbaun@fb2.fra-uas.de](mailto:christianbaun@fb2.fra-uas.de)

# Lernziele dieses Foliensatzes

Betriebssysteme werden nach unterschiedlichen Kriterien klassifiziert

Die wichtigen Unterscheidungskriterien sind Inhalt dieses Foliensatzes

- Am Ende dieses Foliensatzes kennen/verstehen Sie...
  - den Unterschied zwischen **Einzelprogrammbetrieb** (Singletasking) und **Mehrprogrammbetrieb** (Multitasking)
  - den Unterschied zwischen **Einzelbenutzerbetrieb** (Single-User) und **Mehrbenutzerbetrieb** (Multi-User)
  - den Grund für die **Länge der Speicheradressen**
  - was **Echtzeitbetriebssysteme** sind
  - was **Verteilte Betriebssysteme** sind
  - den **Betriebssystemaufbau** (unterschiedliche **Kernelarchitekturen**)
    - **Monolithische Kerne**
    - **Minimale Kerne**
    - **Hybride Kerne**
  - das **Schalenmodell** bzw. **Schichtenmodell**

Übungsblatt 2 wiederholt die für die Lernziele relevanten Inhalte dieses Foliensatzes

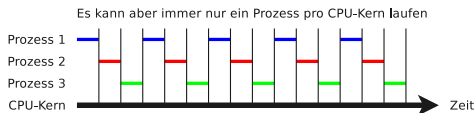
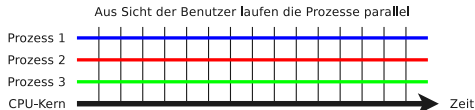
# Einzelprogrammbetrieb und Mehrprogrammbetrieb

- **Einzelprogrammbetrieb** (Singletasking)

- Zu jedem Zeitpunkt läuft nur ein einziges Programm
- Mehrere gestartete Programme werden **nacheinander** ausgeführt

- **Mehrprogrammbetrieb**  
(Multitasking)

- Mehrere Programme können **gleichzeitig** (bei mehreren CPUs/Kernen) oder **zeitlich verschachtelt** (**quasi-parallel**) ausgeführt werden



Task, Prozess, Aufgabe, Auftrag,...

Der Begriff **Task** ist gleichzusetzen mit **Prozess** oder aus Anwendersicht **Aufgabe** bzw. **Auftrag**

# Warum Mehrprogrammbetrieb (Multitasking)?

## Wir wissen...

- Bei **Mehrprogrammbetrieb** laufen mehrere Prozesse nebenläufig
  - Die Prozesse werden in kurzen Abständen, abwechselnd aktiviert  
⇒ Dadurch entsteht der **Eindruck der Gleichzeitigkeit**
  - Nachteil: Das Umschalten von einem Prozess zu anderen, erzeugt **Verwaltungsaufwand (Overhead)**
- 
- Prozesse müssen häufig auf äußere Ereignisse warten
    - Gründe sind z.B. Benutzereingaben, Eingabe/Ausgabe-Operationen von Peripheriegeräten, Warten auf eine Nachricht eines anderen Programms
    - Durch Mehrprogrammbetrieb können Prozesse, die auf ankommende E-Mails, erfolgreiche Datenbankoperationen, geschriebene Daten auf der Festplatte oder ähnliches warten, in den Hintergrund geschickt werden
      - **Andere Prozesse kommen so früher zum Einsatz**
  - **Der Overhead**, der bei der quasiparallelen Abarbeitung von Programmen durch die Programmwechsel entsteht, **ist im Vergleich zum Geschwindigkeitszuwachs zu vernachlässigen**

# Einzelbenutzerbetrieb und Mehrbenutzerbetrieb

- **Einzelbenutzerbetrieb (Single-User)**

- Der Computer steht immer nur einem einzigen Benutzer zur Verfügung

- **Mehrbenutzerbetrieb (Multi-User)**

- Mehrere Benutzer können gleichzeitig mit dem Computer arbeiten
    - Die Benutzer teilen sich die Systemressourcen (möglichst gerecht)
    - Benutzer müssen (u.a. durch Passwörter) identifiziert werden
    - Zugriffe auf Daten/Prozesse anderer Benutzer werden verhindert

	Single-User	Multi-User
<b>Singletasking</b>	MS-DOS, Palm OS	—
<b>Multitasking</b>	OS/2, Windows 3x/95/98, BeOS, MacOS 8x/9x, AmigaOS, Risc OS	Linux/UNIX, MacOS X, Server-Versionen der Windows NT-Familie

- Die Desktop/Workstation-Versionen von Windows NT/XP/Vista/7/8/10/11 sind **halbe Multi-User-Betriebssysteme**
  - Verschiedene Benutzer können nur nacheinander am System arbeiten, aber die Daten und Prozesse der Benutzer sind voreinander geschützt

## 8/16/32/64 Bit-Betriebssysteme

- Die Bit-Zahl gibt die **Länge der Speicheradressen** an, mit denen das Betriebssystem intern arbeitet
  - Ein Betriebssystem kann nur so viele Speichereinheiten ansprechen, wie der Adressraum zulässt
    - Die Größe des Adressraums hängt vom Adressbus ab  $\implies$  Foliensatz 3
- **8 Bit-Betriebssysteme** können  $2^8$  Speichereinheiten adressieren
  - z.B. GEOS, Atari DOS, Contiki
- **16 Bit-Betriebssysteme** können  $2^{16}$  Speichereinheiten adressieren
  - z.B. MS-DOS, Windows 3.x, OS/2 1.x

### Bill Gates (1989)

„We will never make a 32-bit operating system.“

- **32 Bit-Betriebssysteme** können  $2^{32}$  Speichereinheiten adressieren
  - z.B. Windows 95/98/NT/Vista/7/8/10, OS/2 2/3/4, eComStation, Linux, BeOS, MacOS X (bis einschließlich 10.7)
- **64 Bit-Betriebssysteme** können  $2^{64}$  Speichereinheiten adressieren
  - z.B. Linux (64 Bit), Windows 7/8/10 (64 Bit), MacOS X (64 Bit)

# Echtzeitbetriebssysteme (*Real-Time Operating Systems*)

- Sind Multitasking-Betriebssysteme mit zusätzlichen Echtzeit-Funktionen für die Einhaltung von Zeitschranken
- Wesentliche Kriterien von Echtzeitbetriebssystemen:
  - **Reaktionszeit**
  - Einhalten von **Deadlines**
- Unterschiedliche Prioritäten werden berücksichtigt, damit wichtige Prozesse innerhalb gewisser Zeitschranken ausgeführt werden
- 2 Arten von Echtzeitbetriebssystemen existieren:
  - **Harte Echtzeitbetriebssysteme**
  - **Weiche Echtzeitbetriebssysteme**
- Aktuelle Desktop-Betriebssysteme können **weiches Echtzeitverhalten** für Prozesse mit hoher Priorität garantieren
  - Wegen des unberechenbaren Zeitverhaltens durch Swapping, Hardwareinterrupts etc. kann aber kein **hartes Echtzeitverhalten** garantiert werden

# Harte und Weiche Echtzeitbetriebssysteme

## ● Harte Echtzeitbetriebssysteme

- Zeitschranken müssen unbedingt eingehalten werden
- Verzögerungen können unter keinen Umständen akzeptiert werden
- Verzögerungen führen zu katastrophalen Folgen und hohen Kosten
- Ergebnisse sind nutzlos wenn sie zu spät erfolgten
- Einsatzbeispiele: Schweißroboter, Reaktorsteuerung, ABS, Flugzeugsteuerung, Überwachungssysteme auf der Intensivstation

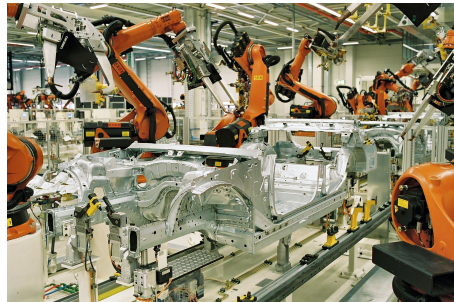
## ● Weiche Echtzeitbetriebssysteme

- Gewisse Toleranzen sind erlaubt
- Verzögerungen führen zu akzeptablen Kosten
- Einsatzbeispiele: Telefonanlage, Parkschein- oder Fahrkartenautomat, Multimedia-Anwendungen wie Audio/Video on Demand



# Einsatzgebiete von Echtzeitbetriebssystemen

- Typische Einsatzgebiete von Echtzeitbetriebssystemen:
  - Mobiltelefone
  - Industrielle Kontrollsysteme
  - Roboter
- Beispiele für Echtzeitbetriebssysteme:
  - QNX
  - VxWorks
  - LynxOS
  - RTLinux
  - Symbian (veraltet)
  - Windows CE (veraltet)



Bildquelle: BMW Werk Leipzig (CC-BY-SA 2.0)

# Die QNX Demo Disc von 1999...

web.archive.org/web/20011019174050/www.qnx.com/demodisk/

**THE INCREDIBLE 1.44M DEMO**

build a more reliable world™

Download QNX 4 Demo | Extend OS functionality | Troubleshooting | Demo Home

**THE INCREDIBLE 1.44M DEMO**

Version 4 is here!

This single bootable floppy contains:

- realtime OS
- GUI
- browser
- server
- dialer
- TCP/IP
- sample apps
- and much more ...

Surf the web. Serve HTML pages. Extend the OS - on the fly ...

**Note:**

This is a demo of the QNX 4 RTOS. To download the new QNX realtime platform, visit [get.qnx.com](http://get.qnx.com).

**Create your own demo**

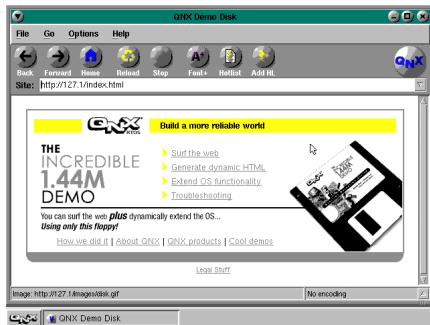
All you need is a 1.44M floppy! Just insert your disk, click on "Create a demo," and follow instructions.

Once you've downloaded the 1.44M QNX Demo, you can:

- Surf the web - Use your IP address to connect, and get ready to surf!
- Generate dynamic HTML - Even diskless systems can generate HTML pages in real time.
- Extend the OS - Download drivers on the fly - without rebooting.

**Product info**

- QNX Realtime Platform
- QNX 4 RTOS
- QNX Neutrino
- Photon microGUI
- PhAR
- Internet Suite
- IAT
- more ...



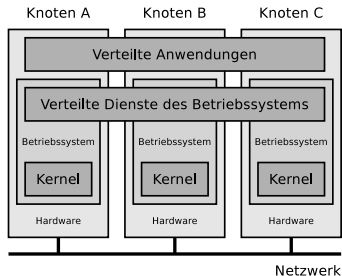
Bildquelle: <http://toastytech.com/guis/qnxdemo.html>

Beeindruckendes Video über die Demo Disc:  
[https://www.youtube.com/watch?v=K\\_Vl16IBEJO](https://www.youtube.com/watch?v=K_Vl16IBEJO)

# Verteilte Betriebssysteme

- Verteiltes System
- Steuert die Prozesse auf mehreren Rechner eines Clusters
- Die einzelnen Rechner bleiben den Benutzern und deren Prozessen transparent verborgen
  - Das System erscheint als ein einzelner großer Rechner
  - Prinzip des **Single System Image**

- Das Prinzip der verteilten Betriebssysteme ist tot!
- Aber: Bei der Entwicklung einiger verteilter Betriebssysteme wurden einige interessante Technologien entwickelt oder erstmals angewendet
- Einige dieser Technologien sind heute noch aktuell



# Verteilte Betriebssysteme (1/3)

## ● Amoeba

- Mitte der 1980er Jahre bis Mitte der 1990er Jahre
- Andrew S. Tanenbaum (Freie Universität Amsterdam)
- Die Programmiersprache Python wurde für Amoeba entwickelt

<http://www.cs.vu.nl/pub/amoeba/>

The Amoeba Distributed Operating System. A. S. Tanenbaum, G. J. Sharp. <http://www.cs.vu.nl/pub/amoeba/Intro.pdf>

## ● Inferno

- Basiert auf dem UNIX-Betriebssystem Plan 9
- Bell Laboratories
- Anwendungen werden in der Sprache Limbo programmiert
  - Limbo produziert wie Java Bytecode, den eine virtuelle Maschine ausführt
- Minimale Anforderungen an die Hardware
  - Benötigt nur 1 MB Arbeitsspeicher

<http://www.vitanuova.com/inferno/index.html>

# Verteilte Betriebssysteme (2/3)

## • Rainbow

- Universität Ulm
- Konzept eines gemeinsamen Speichers mit einem für alle Rechner im Cluster einheitlichen Adressraum, in welchem Objekte abgelegt werden
  - Für Anwendungen ist es transparent, auf welchem Rechner im Cluster sich Objekte physisch befinden
  - Anwendungen können über einheitliche Adressen von jedem Rechner auf gewünschte Objekte zugreifen
  - Sollte sich das Objekt physisch im Speicher eines entfernten Rechners befinden, sorgt Rainbow automatisch und transparent für eine Übertragung und lokale Bereitstellung auf dem bearbeitenden Rechner

Rainbow OS: A distributed STM for in-memory data clusters. *Thilo Schmitt, Nico Kämmer, Patrick Schmidt, Alexander Weggerle, Steffen Gerhold, Peter Schulthess*. MIPRO 2011

# Verteilte Betriebssysteme (3/3)

## • Sprite

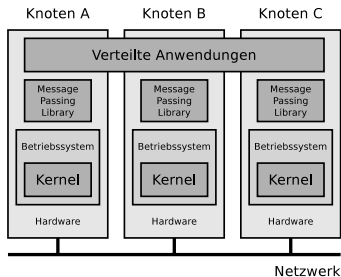
- University of California, Berkeley (1984-1994)
- Verbindet Workstations so, dass Sie für die Benutzer wie ein einzelnes System mit Dialogbetrieb (*Time Sharing*) erscheinen
- pmake, eine parallele Version von make, wurde für Sprite entwickelt

<http://www.stanford.edu/~ouster/cgi-bin/spriteRetrospective.php>

The Sprite Network Operating System. 1988. <http://www.research.ibm.com/people/f/fdougkis/papers/sprite.pdf>

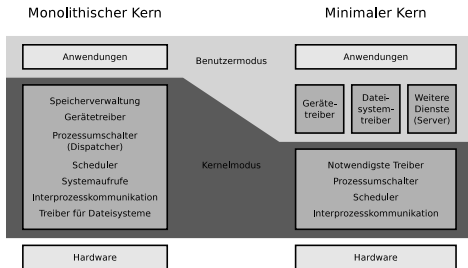
# Verteilte Betriebssysteme heute

- Das Konzept konnte sich nicht durchsetzen
  - Verteilte Betriebssysteme kamen nicht aus dem Stadium von Forschungsprojekten heraus
  - Etablierte Betriebssysteme konnten nicht verdrängt werden
- Um Anwendungen für Cluster zu entwickeln, existieren Bibliotheken, die von der Hardware unabhängiges **Message Passing** bereitstellen
  - Kommunikation via Message Passing basiert auf dem Versand von Nachrichten
  - Verbreitete Message Passing Systeme:
    - **Message Passing Interface (MPI)**  
⇒ Standard-Lösung
    - **Parallel Virtual Machine (PVM)** ⇒ †



# Betriebssystemaufbau (Kernelarchitekturen)

- Der **Kernel** enthält die grundlegenden Funktionen des Betriebssystems und ist die Schnittstelle zur Hardware



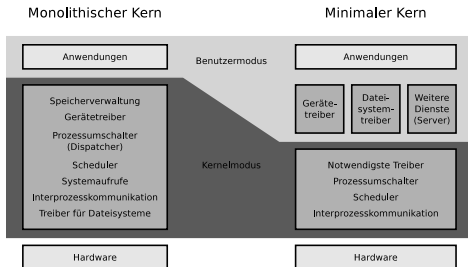
- Unterschiedliche Kernelarchitekturen existieren
    - Sie unterscheiden sich darin, welche Funktionen **im Kern** enthalten sind und welche sich **außerhalb des Kerns** als Dienste (Server) befinden
  - Funktionen im Kern, haben vollen Hardwarezugriff (**Kernelmodus**)
  - Funktionen außerhalb des Kerns können nur auf ihren virtuellen Speicher zugreifen (**Benutzermodus**)
- ⇒ Foliensatz 5



# Monolithische Kerne (1/2)

## • Enthalten Funktionen zur...

- Speicherverwaltung
- Prozessverwaltung
- Prozesskommunikation
- Hardwareverwaltung
- Dateisysteme



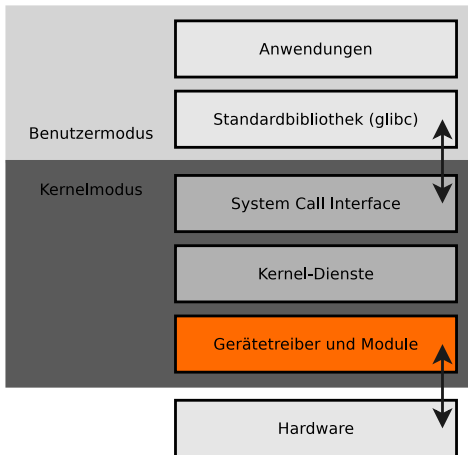
## • Vorteile:

- Weniger Kontextwechsel als Mikrokern  $\Rightarrow$  höhere Geschwindigkeit
- Gewachsene Stabilität
  - Mikrokern sind in der Regel nicht stabiler als monolithische Kerne

## • Nachteile:

- Abgestürzte Komponenten des Kerns können nicht separat neu gestartet werden und das gesamte System nach sich ziehen
- Hoher Entwicklungsaufwand für Erweiterungen am Kern, da dieser bei jedem Kompilieren komplett neu übersetzt werden muss

## Monolithische Kerne (2/2)



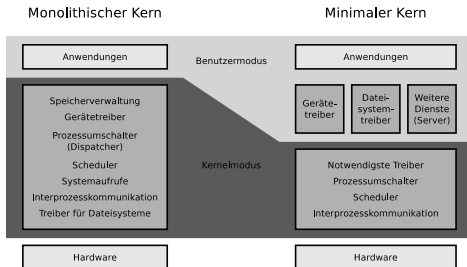
- Linux ist das populärste moderne Betriebssystem mit einem monolithischem Kern
- Hardware- und Dateisystem-Treiber im **Linux-Kernel** können in Module ausgelagert werden
  - Die Module laufen im *Kernelmodus* und nicht im *Benutzermodus*
  - Darum ist der Linux-Kernel ein monolithischer Kernel

### Beispiele für Betriebssysteme mit monolithischem Kern

Linux, BSD, MS-DOS, FreeDOS, Windows 95/98/ME, MacOS (bis 8.6), OS/2

# Minimale Kerne (1/2)

- Im Kern sind primär. . .
  - Funktionen zur Speicher- und Prozessverwaltung
  - Funktionen zur Synchronisation und Interprozesskommunikation
  - Notwendigste Treiber (z.B. zum Systemstart)
- Gerätetreiber, Dateisysteme und Dienste (Server) sind außerhalb des Kerns und laufen wie die Anwendungsprogramme im Benutzermodus



## Beispiele für Betriebssysteme mit Mikrokernel

AmigaOS, MorphOS, Tru64, QNX Neutrino, Symbian, GNU HURD (siehe Folie 24)

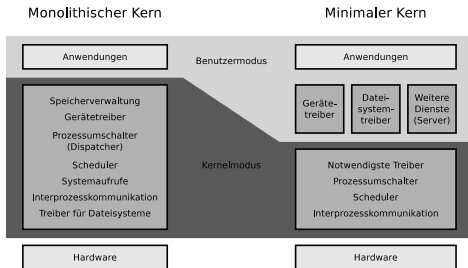
## Minimale Kerne (2/2)

- Vorteile:

- Einfache Austauschbarkeit der Komponenten
- Theoretisch beste Stabilität und Sicherheit
  - Grund: Es laufen weniger Funktionen im Kernelmodus

- Nachteile:

- Langsamer wegen der größeren Zahl von Kontextwechseln
- Entwicklung eines neuen (Mikro-)kernels ist eine komplexe Aufgabe

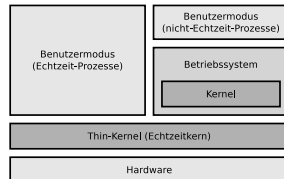


Der Anfang der 1990er Jahre prognostizierte Erfolg der Mikrokernelsysteme blieb aus  
⇒ Diskussion von Linus Torvalds vs. Andrew S. Tanenbaum (1992) ⇒ siehe Folie 23

# Kernelarchitekturen von Echtzeitbetriebssystemen

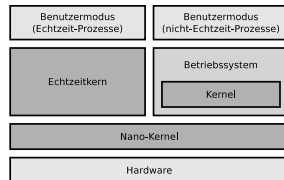
## • **Thin-Kernel** (ähnlich wie Mikrokernel)

- Der Betriebssystemkern selbst läuft als Prozess mit niedrigster Priorität im Hintergrund
- Der Echtzeit-Kernel übernimmt das Scheduling
  - Er ist eine abstrahierende Schnittstelle zwischen Hardware und Linux-Kernel
- Echtzeit-Prozesse haben die höchste Priorität  
⇒ minimale Reaktionszeiten (Latenzzeiten)



## • **Nano-Kernel** (auch ähnlich wie Mikrokernel)

- Neben dem Echtzeit-Kernel kann eine beliebige Anzahl anderer Betriebssystem-Kerne laufen
- Ein Nano-Kernel arbeitet so ähnlich wie ein Typ-1-Hypervisor (⇒ Foliensatz 10)



## • **Pico-Kernel, Femto-Kernel, Atto-Kernel**

- Marketingbegriffe um die Winzigkeit von Echtzeit-Kernen hervorzuheben

Quelle: Anatomy of real-time Linux architectures (2008): <http://www.ibm.com/developerworks/library/l-real-time-linux/>

# Hybride Kerne / Hybridkernel / Makrokernel

- Kompromiss zwischen monolithischen Kernen und minimalen Kernen
  - Enthalten aus Geschwindigkeitsgründen Komponenten, die bei minimalen Kernen außerhalb des Kerns liegen
- Es ist nicht festgelegt, welche Komponenten bei Systemen mit hybriden Kernen zusätzlich in den Kernel einkompiliert sind
- Die Vor- und Nachteile von hybriden Kernen zeigt Windows NT 4
  - Das Grafiksystem ist bei Windows NT 4 im Kernel enthalten
    - Vorteil: Steigerung der Performance
    - Nachteil: Fehlerhafte Grafiktreiber führen zu häufigen Abstürzen

Quelle: **MS Windows NT Kernel-mode User and GDI White Paper**. <https://technet.microsoft.com/library/cc750820.aspx>

- Vorteile:
  - Bessere Geschwindigkeit als minimale Kerne da weniger Kontextwechsel
  - Höhere Stabilität (theoretisch!) als monolithische Kerne

## Beispiele für Betriebssysteme mit hybriden Kernen

Windows seit NT 3.1, ReactOS, MacOS X, BeOS, ZETA, Haiku, Plan 9, DragonFly BSD

# Linus Torvalds vs. Andrew Tanenbaum (1992)

Bildquelle: unbekannt

- 26. August 1991: Linus Torvalds kündigt das Projekt Linux in der Newsgroup `comp.os.minix` an
  - 17. September 1991: Erste interne Version (0.01)
  - 5. Oktober 1991: Erste offizielle Version (0.02)
- 29. Januar 1992: Andrew S. Tanenbaum schreibt in der Newsgroup `comp.os.minix`: „**LINUX is obsolete**“
  - Linux hat einen monolithischen Kernel  $\implies$  Rückschritt
  - Linux ist nicht portabel, weil auf 80386er optimiert und diese Architektur wird demnächst von RISC-Prozessoren abgelöst (Irrtum!)



Es folgte eine mehrere Tage dauernde, zum Teilmotional geführte Diskussion über die Vor- und Nachteile von monolithischen Kernen, minimalen Kernen, Portabilität und freie Software

A. Tanenbaum (30. Januar 1992): „*I still maintain the point that designing a monolithic kernel in 1991 is a fundamental error. Be thankful you are not my student. You would not get a high grade for such a design :-)*“.

Quelle: <http://www.oreilly.com/openbook/opensources/book/appa.html>

Die Zukunft kann nicht vorhergesagt werden

# Ein trauriges Kernel-Beispiel – HURD

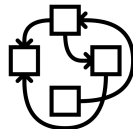
- 1984: Richard Stallman gründet das GNU-Projekt
- Ziel: Entwicklung eines freien UNIX-Betriebssystems  
⇒ **GNU HURD**
- GNU HURD besteht aus:
  - GNU Mach, der Mikrokern
  - Dateisysteme, Protokolle, Server (Dienste), die im Benutzermodus laufen
  - GNU Software, z.B. Editoren (GNU Emacs), Debugger (GNU Compiler Collection), Shell (Bash),...
- GNU HURD ist *so weit* fertig
  - Die GNU Software ist seit Anfang der 1990er Jahre weitgehend fertig
  - Nicht alle Server sind fertig implementiert
- Eine Komponente fehlt noch: Der Mikrokern



Bildquelle:  
stallman.org



Wikipedia  
(CC-BY-SA-2.0)



Wikipedia  
(CC-BY-SA-3.0)



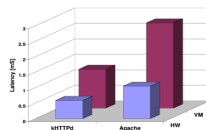
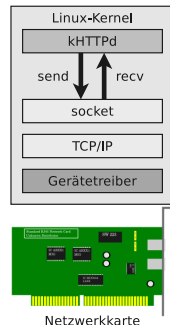
# Ein extremes Kernel-Beispiel – kHTTPd

<http://www.fenrus.demon.nl>

- 1999: Arjan van de Ven entwickelt für Linux Kernel 2.4.x den **kernel-basierten Web-Server kHTTPd**

The Design of kHTTPd: <https://www.linux.it/~rubini/docs/khttpd/khttpd.html>  
Announce: kHTTPd 0.1.0: <http://static.lwn.net/1999/0610/a/khttpd.html>

- Vorteil: Beschleunigte Auslieferung statischer(!) Web-Seiten
  - Weniger Moduswechsel zwischen Benutzer- und Kernelmodus
- Nachteil: Sicherheitsrisiko
  - Komplexe Software wie ein Web-Server sollten nicht im Kernelmodus laufen
  - Bugs im Web-Server könnten zu Systemabstürzen oder zur vollständigen Kontrollübernahme durch Angreifer führen
- Im Linux Kernel  $\geq 2.6.x$  ist kHTTPd nicht enthalten

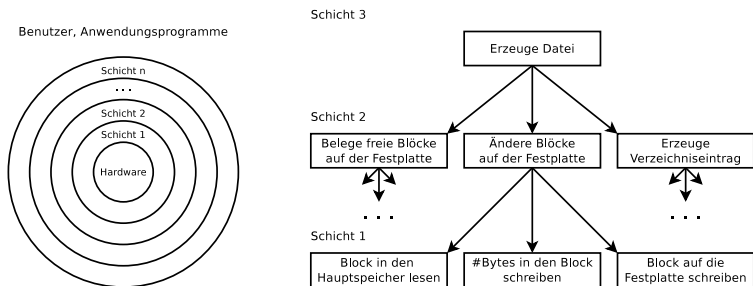


Bildquelle:  
Kernel Plugins: When A VM  
Is Too Much. *Ivan Ganey,  
Greg Eisenhauer, Karsten  
Schwan. 2004*

# Schichtenmodell (1/2)

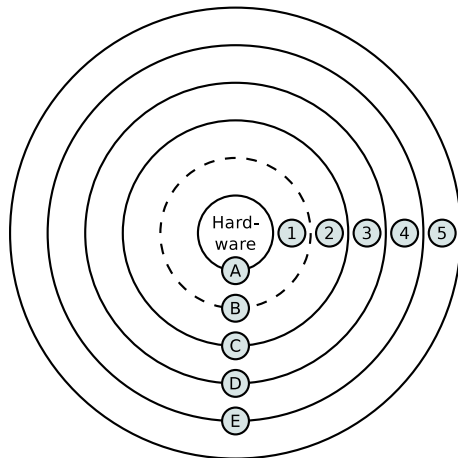
- Betriebssysteme werden mit ineinander liegenden Schichten logisch strukturiert
  - Die Schichten umschließen sich gegenseitig
  - die Schichten enthalten von innen nach außen immer abstraktere Funktionen
- Das Minimum sind 3 Schichten:
  - Die **innerste Schichten** enthält die hardwareabhängigen Teile des Betriebssystems
    - So können Betriebssysteme (theoretisch!) leicht an unterschiedliche Rechnerarchitekturen angepasst werden
  - Die **mittlere Schichten** enthält grundlegende Ein-/Ausgabe-Dienste (Bibliotheken und Schnittstellen) für Geräte und Daten
  - Die **äußerste Schichten** enthält die Anwendungsprogramme und die Benutzerschnittstelle
- In der Regel stellt man Betriebssysteme mit mehr als 3 logischen Schichten dar

## Schichtenmodell (2/2)



- Jede Schicht ist mit einer **abstrakten Maschine** vergleichbar
- Die Schichten kommunizieren mit benachbarten Schichten über **wohldefinierte Schnittstellen**
- Schichten können Funktionen der nächst inneren Schicht aufrufen
- Schichten stellen Funktionen der nächst äußeren Schicht zur Verfügung
- Alle Funktionen (**Dienste**), die eine Schicht anbietet, und die Regeln, die dabei einzuhalten sind, heißen **Protokoll**

# Schichtenmodell von Linux/UNIX



- ① Kernel (maschinenabhängiger Teil)
- ② Kernel (maschinenunabhängiger Teil)
- ③ Standardbibliothek (glibc)
- ④ Shell (bash), Anwendungen
- ⑤ Benutzer

- A Hardwareschnittstelle
- B kernelinterne, hardwareunabhängige Schnittstelle
- C Systemaufrufschnittstelle
- D Bibliothekenschnittstelle (Schnittstelle zu glibc)
- E Benutzerschnittstelle

In der Realität ist das Konzept aufgeweicht. Anwendungen der Benutzer können z.B. Bibliotheksfunktionen der Standardbibliothek glibc oder direkt die Systemaufrufe aufrufen (⇒ siehe Foliensatz 7)