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

Lernziele dieses Foliensatzes

Betriebssysteme werden nach unterschiedlichen Kriterien klassifiziert

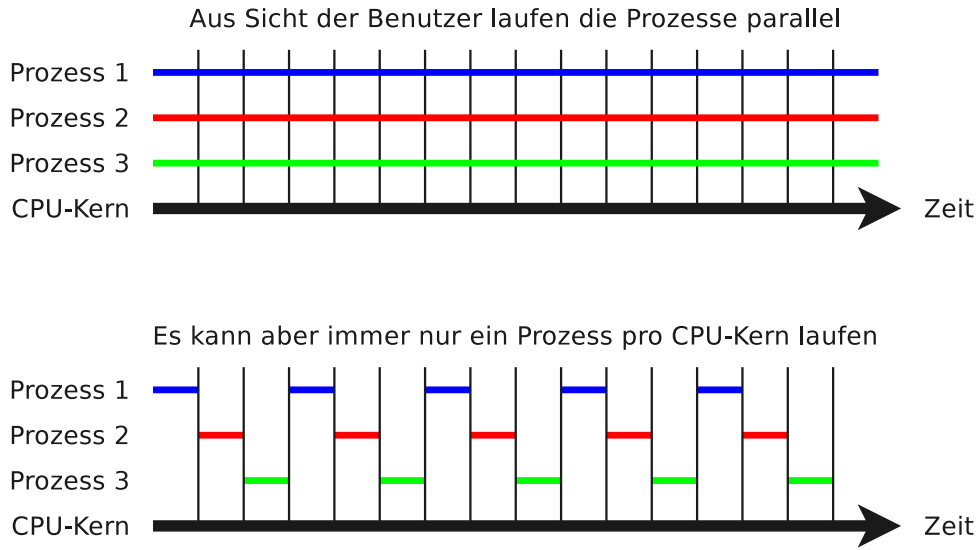
Die wichtigen Unterscheidungskriterien sind Inhalt dieses Foliensatzes

- 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**
- was die Schritte des **Bootprozess** (Bootstrap) sind

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
  $\Longrightarrow$ 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**
Singletasking	MS-DOS, Palm OS	—
Multitasking	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 $\Longrightarrow$ 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*)

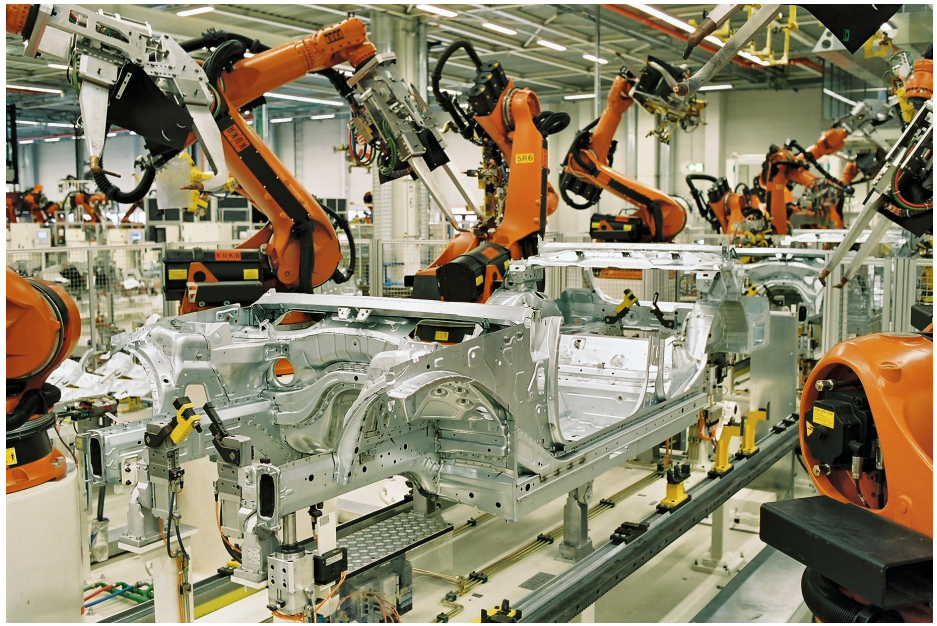
- Sind Multitasking-Betriebssysteme mit zusätzlichen Echtzeit-Funktionen für die Einhaltung von Zeitschranken
- Wesentliche Kriterien von Echtzeitbetriebssystemen:
  - **Reaktionszeit**
  - Einhalten von **Deadlines**
- Unterschiedliche Prioritäten werden berücksichtigt, damit wichtige Prozesse innerhalb 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

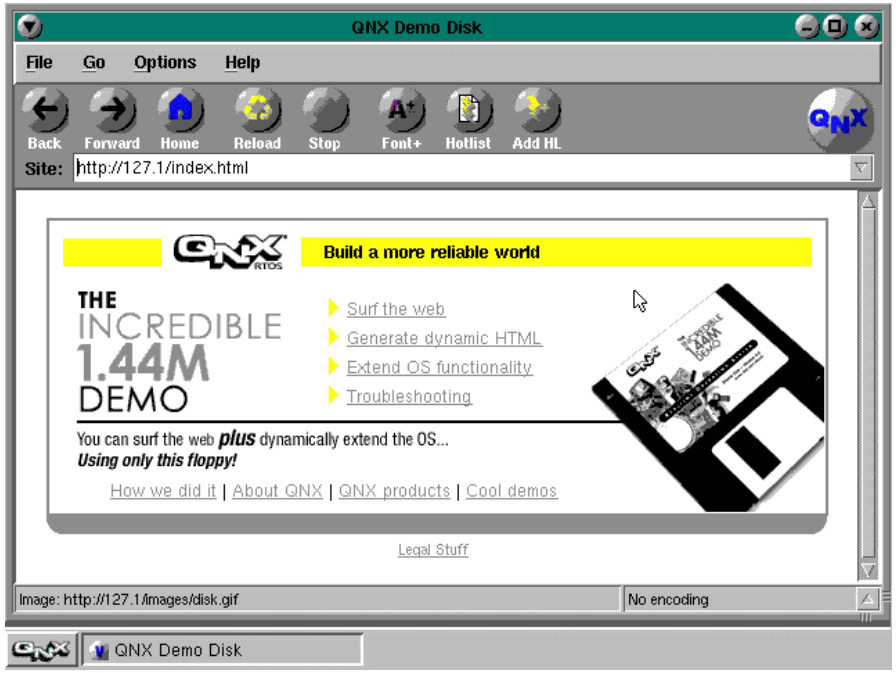
Einsatzgebiete von Echtzeitbetriebssystemen

- 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...



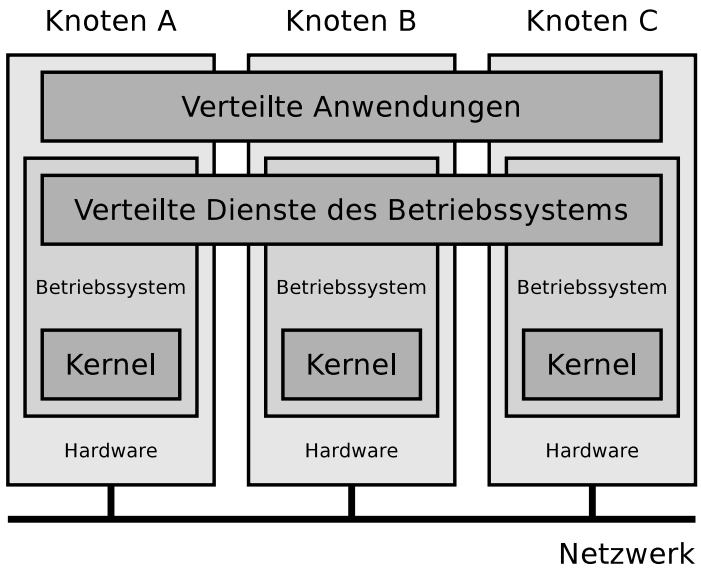
Bildquelle: `http://toastytech.com/guis/qnxdemo.html`

Beeindruckendes Video über die Demo Disc:
`https://www.youtube.com/watch?v=K_V1I6IBEJ0`

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)

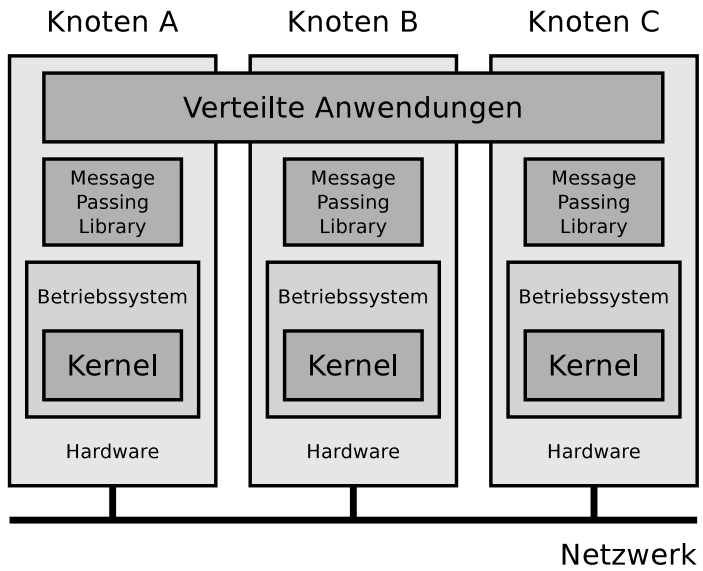
- **Sprite**
  - University of California, Berkeley (1984-1994)
  - Verbindet Workstations so, dass Sie für die Benutzer wie eine einzelnes System mit Dialogbetrieb (*Time Sharing*) erscheinen
  - `pmake`, eine parallele Version von `make`, wurde für Sprite entwickelt

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

The Sprite Network Operating System. 1988. `http://www.research.ibm.com/people/f/fdouglis/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)
      $\Longrightarrow$ Standard-Lösung
    - **Parallel Virtual Machine** (PVM)
      $\Longrightarrow$ †

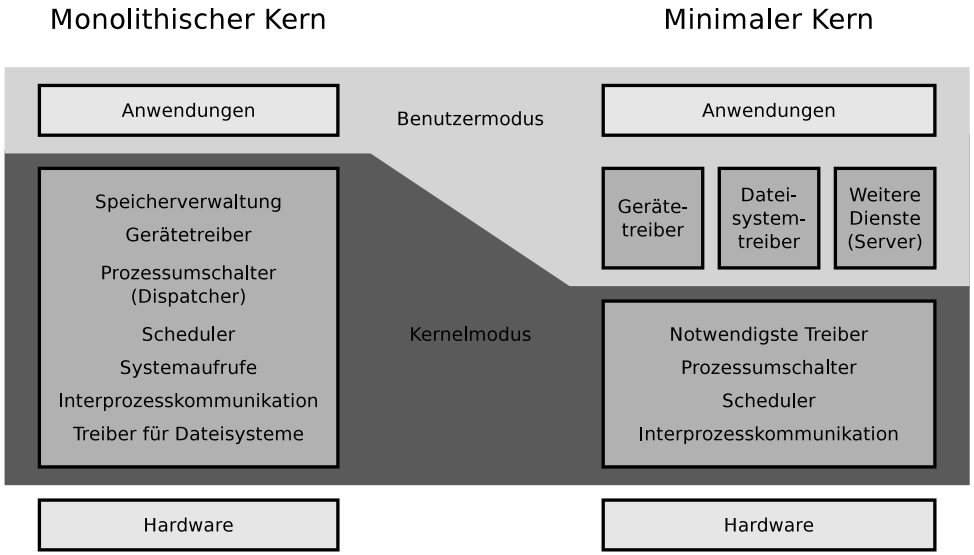


MPI tutorials

`http://mpitutorial.com/tutorials/`

Betriebssystemaufbau (Kernelarchitekturen)

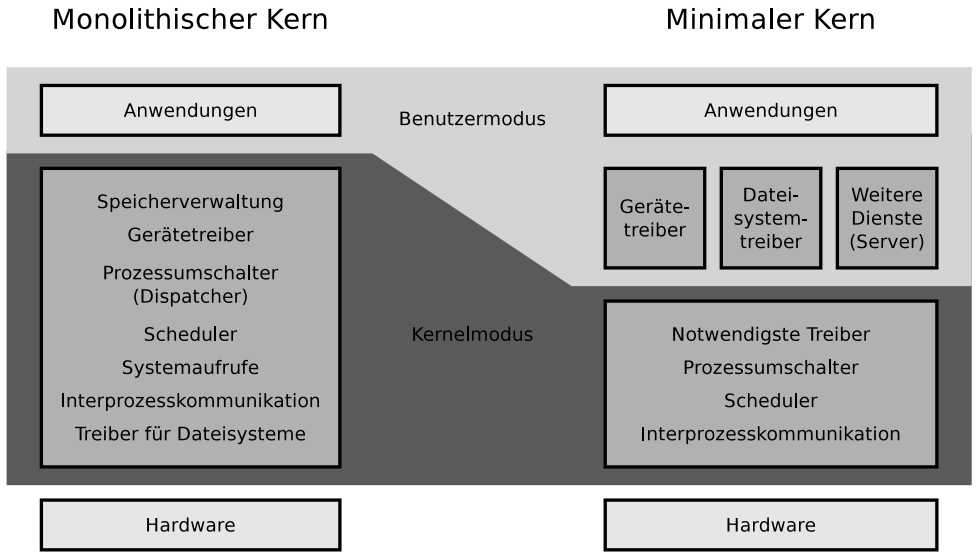
- Der **Kernel** enthält die grundlegenden Funktionen des Betriebssystems und ist die Schnittstelle zur Hardware



- 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**)
  $\Longrightarrow$ Foliensatz 5

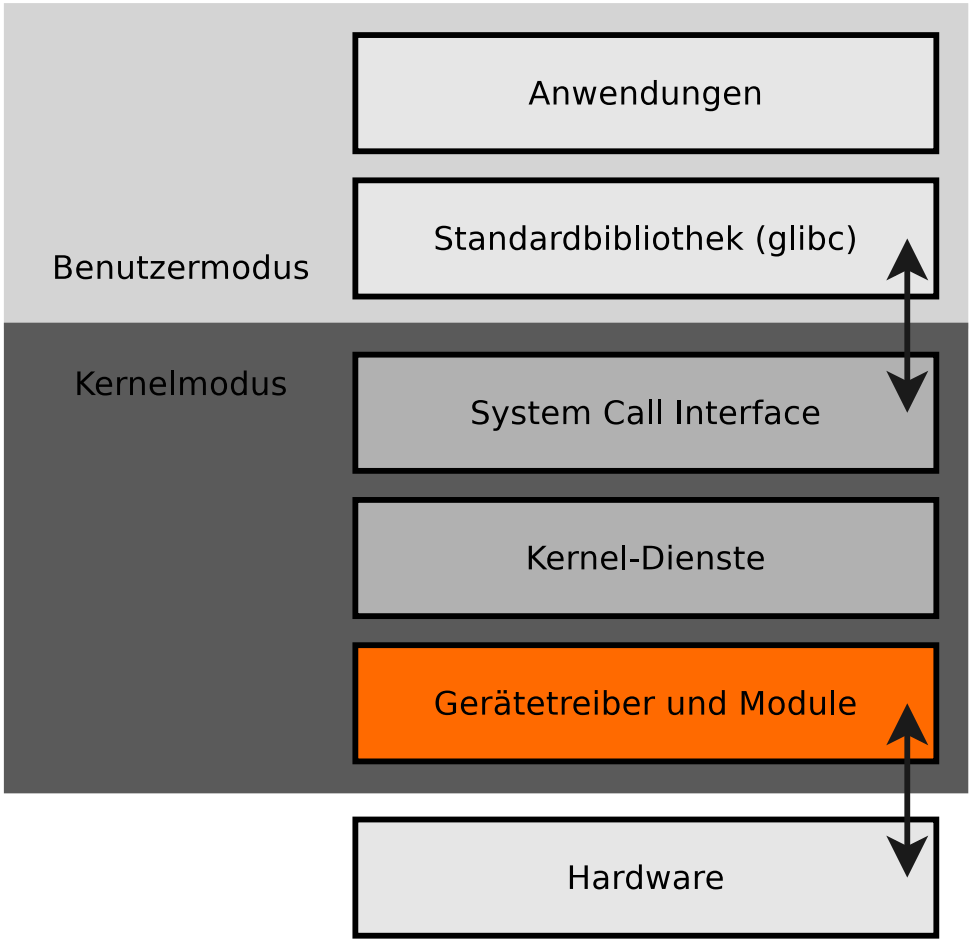
Monolithische Kerne (1/2)

- Enthalten Funktionen zur...
  - Speicherverwaltung
  - Prozessverwaltung
  - Prozesskommunikation
  - Hardwareverwaltung
  - Dateisysteme



- Vorteile:
  - Weniger Kontextwechsel als Mikrokernel $\Longrightarrow$ höhere Geschwindigkeit
  - Gewachsene Stabilität
    - Mikrokernel 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)



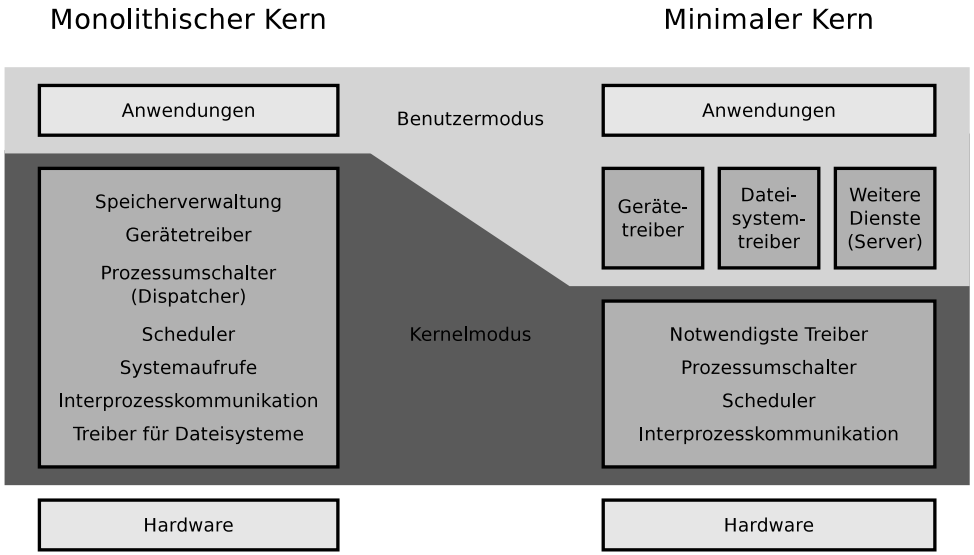
- Linux ist das populärste moderne Betriebssystem mit einem monolithischem Kern
- Hardware- und Dateisystem-Treiber im **Linux-Kernel** können in Module auslagert werden
  - Die Module laufen im *Kernelmodus* und nicht im *Benutzermodus*
    - Darum ist der Linux-Kernel ein monolithischer Kernel

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)



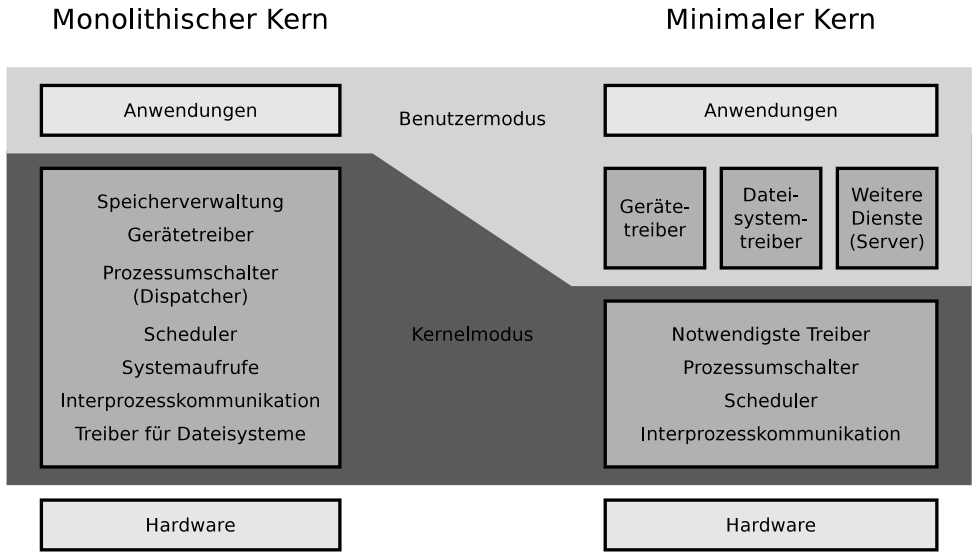
- Gerätetreiber, Dateisysteme und Dienste (Server) sind außerhalb des Kerns und laufen wie die Anwendungsprogramme im Benutzermodus

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

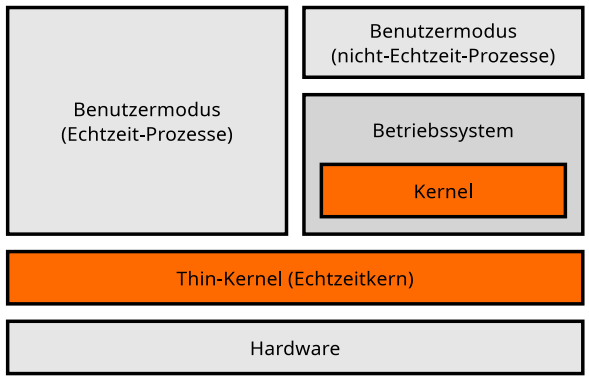


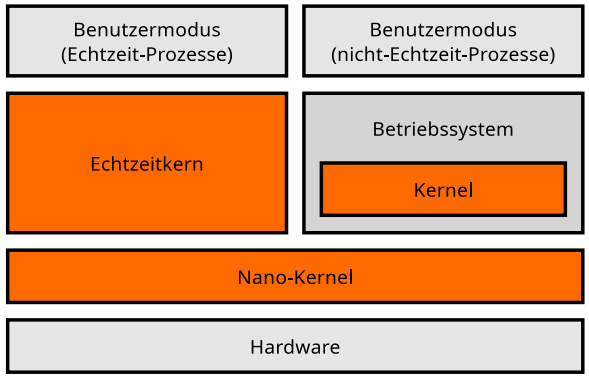
- Nachteile:
  - Langsamer wegen der größeren Zahl von Kontextwechseln
  - Entwicklung eines neuen (Mikro-)kernels ist eine komplexe Aufgabe

Der Anfang der 1990er Jahre prognostizierte Erfolg der Mikrokernelsysteme blieb aus
$\Longrightarrow$ Diskussion von Linus Torvalds vs. Andrew S. Tanenbaum (1992) $\Longrightarrow$ 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 $\Longrightarrow$ 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 ($\Longrightarrow$ Foliensatz 10)
- **Pico-Kernel, Femto-Kernel, Atto-Kernel**
  - Marketingbegriffe um die Winzigkeit von Echtzeit-Kerneln 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 NT-Familie 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 $\Longrightarrow$ Rückschritt
  - Linux ist nicht portabel, weil auf 80386er optimiert und diese Architektur wird demnächst von RISC-Prozessoren abgelöst (Irrtum!)

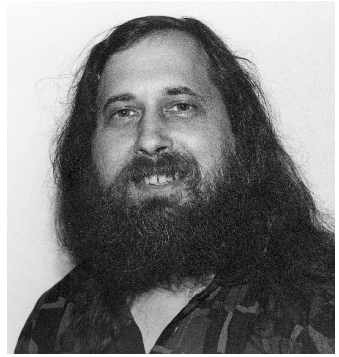
Es folgte eine mehrere Tage dauernde, zum Teilemotional geführte Diskussion über die Vor- und Nachteile von monolithischen Kernen, minimalen Kernen, Portabilität und freie Software

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

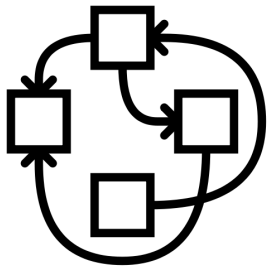
Ein trauriges Kernel-Beispiel – HURD

- 1984: Richard Stallman gründet das GNU-Projekt
- Ziel: Entwicklung eines freien UNIX-Betriebssystems
  $\Longrightarrow$ **GNU HURD**
- GNU HURD besteht aus:
  - GNU Mach, der Mikrokernel
  - 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 Mikrokernel



Bildquelle:
`stallman.org`

Wikipedia
(CC-BY-SA-2.0)



Wikipedia
(CC-BY-SA-3.0)

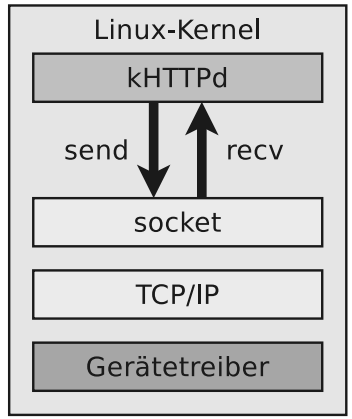
Ein extremes Kernel-Beispiel – kHTTPd

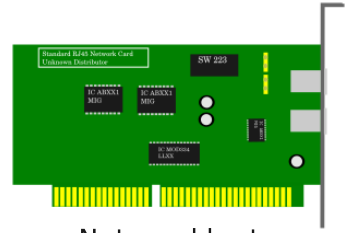
http://www.fenrus.demon.nl

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

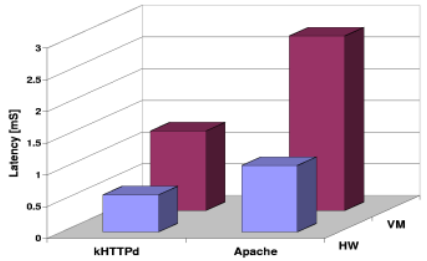
The Design of kHTTPd: https://www.linux.it/~rubini/docs/khttpd/khttpd.html
Announce: kHTTPd 0.1.0: http://static.lwn.net/1999/0610/a/khttpd.html

- 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





Netzwerkkarte

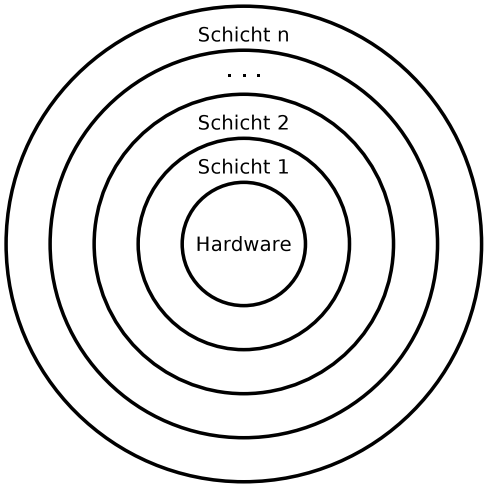


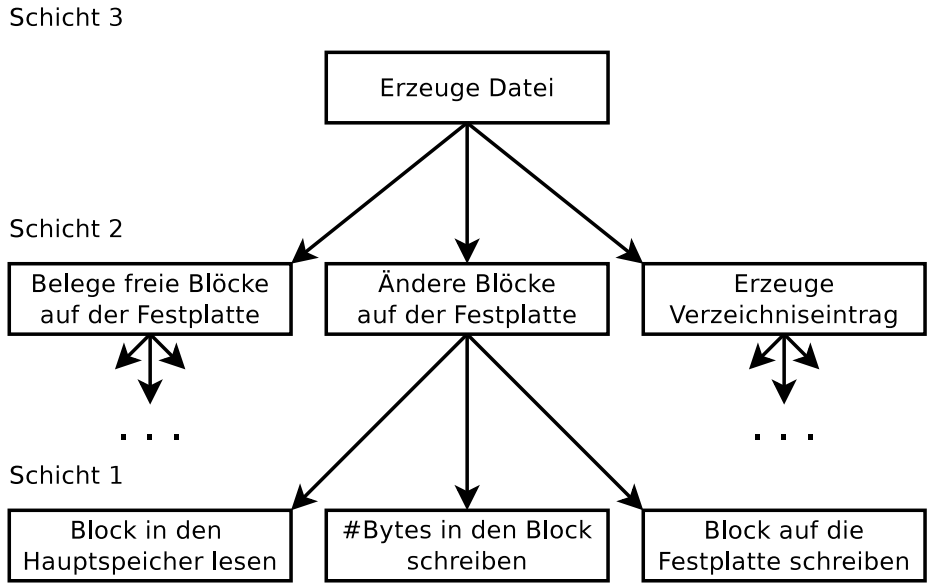
Bildquelle:
Kernel Plugins: When A VM Is Too Much. *Ivan Ganev, Greg Eisenhauer, Karsten Schwan*. 2004

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

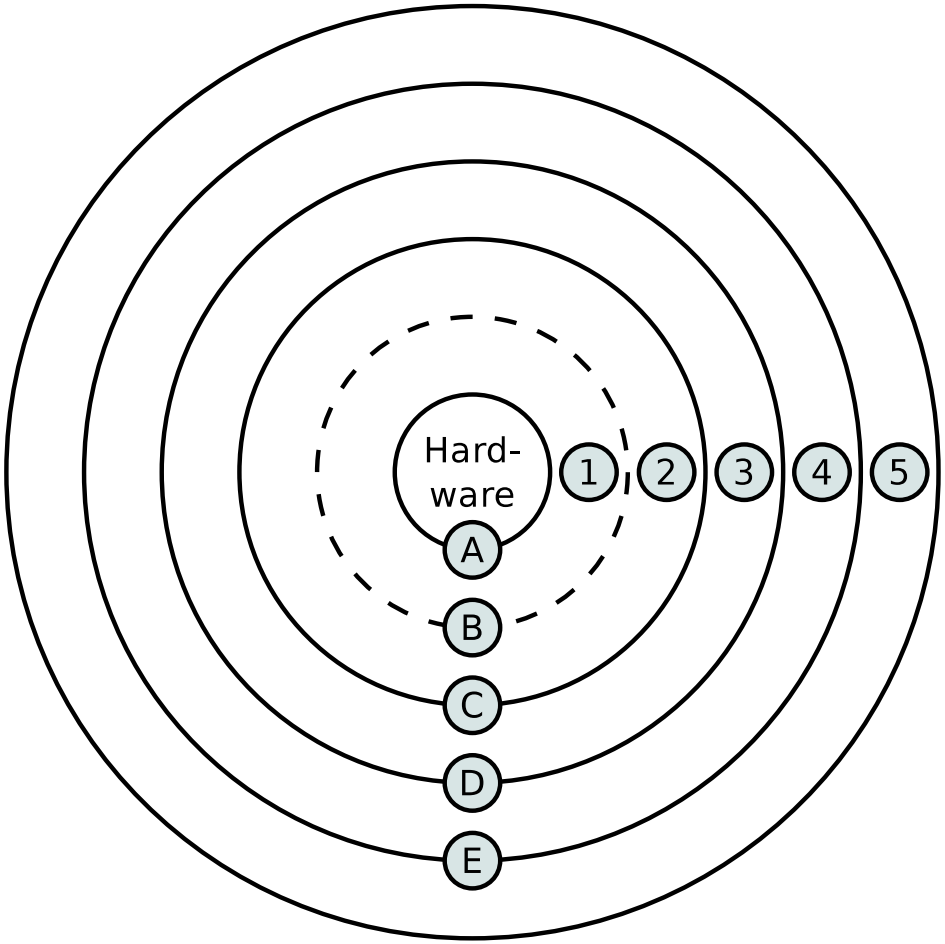
Schichtenmodell (2/2)





- 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

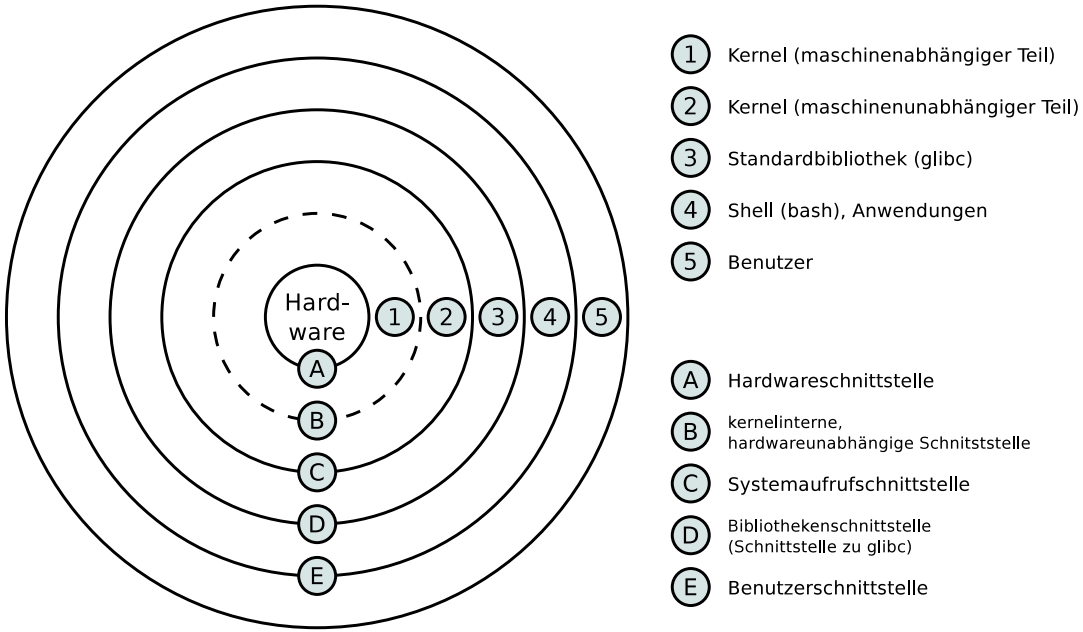


1. Kernel (maschinenabhängiger Teil)
2. Kernel (maschinenunabhängiger Teil)
3. Standardbibliothek (glibc)
4. Shell (bash), Anwendungen
5. Benutzer

- A: Hardwareschnittstelle
- B: kernelinterne, hardwareunabhängige Schnitststelle
- 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 ($\Longrightarrow$ siehe Foliensatz 7)

Inhalte der Vorlesung Betriebssysteme



- Dieses Modul orientiert sich an den Ebenen des Schichtenmodells
  - Ebene 0 $\Longrightarrow$ **Hardware:** Foliensätze 3+4
  - Ebene 1 $\Longrightarrow$ **Kernel-Architektur:** Foliensatz 2
  - Ebene 2 $\Longrightarrow$ **Kernel-Funktionen:** Foliensätze 5+6+7+8+9
  - Ebene 3 $\Longrightarrow$ **Standardbibliothek:** Foliensätze 7+9
  - Ebene 4 $\Longrightarrow$ **Shell:** Übungsblätter und Beispiele auf den Foliensätzen
  - Ebene 5 $\Longrightarrow$ **Benutzer:** Sie :-)

Start des Betriebssystems

- Der Start eines Computers und seines Betriebssystems heißt **Bootvorgang**, **Bootprozess** oder **Bootstrapping**
  - Er umfasst Schritte von der Initialisierung der Hardwarekomponenten des Computers bis zur Übergabe der Kontrolle an das Betriebssystem und seine Benutzer bzw. deren Prozesse und die Bereitstellung einer Benutzerschnittstelle
- Die einzelnen Schritte des Bootvorgangs in Linux/UNIX-Betriebssystemen sind:
  1. Computer einschalten
  2. Firmware starten und Selbsttest durchführen
  3. Bootloader starten
  4. Betriebssystemkern starten und temporäres Root-Dateisystem einbinden
  5. Echtes Root-Dateisystem einbinden
  6. init/systemd und Systemprozesse starten
  7. Kontrolle an die Benutzer übergeben

Die folgenden Folien beschreiben die einzelnen Schritte des Bootvorgangs

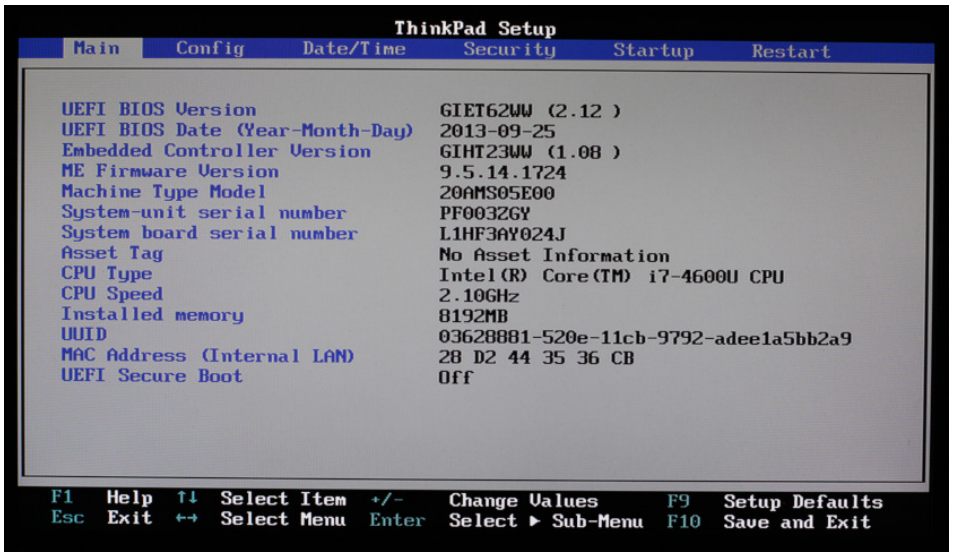
(1) Einschalten des Computers

- Bei älteren Computern (bis in die 2000er Jahre) wird das Mainboard durch Einschalten des Computers mit Strom versorgt und der Prozessor beginnt mit dem **Von-Neumann-Zyklus** ($\Longrightarrow$ Foliensatz 3)
- Bei moderneren Computern läuft üblicherweise dauerhaft ein **autonomes Subsystem** wie:
  - **Intel Management Engine** (seit 2008)
  - **AMD Platform Security Processor** (seit 2013)
  - Solche Subsysteme sind eigenständige Mikrocontroller auf dem Mainboard oder im Chipsatz ($\Longrightarrow$ Foliensatz 3) oder alternativ spezielle Prozessorkerne im Hauptprozessor mit eigenem Betriebssystem
    - Sie laufen üblicherweise immer dann, wenn ein ausreichend geladener Akku oder eine permanente Stromquelle vorhanden ist
    - Sie ermöglichen ein Überwachen und Aufwecken eines Rechners über das Netzwerk (**Wake-on-LAN**) und realisieren Möglichkeiten zur Fernadministration (**Remote-Management**)

- Carikli D. (2018). **The Intel Management Engine: an attack on computer users' freedom.** Free Software Foundation. `https://static.fsf.org/nosvn/blogs/Intel_ME_Carikli_article_PRINT_2.pdf`
- Ermolov M, Goryachy M. (2017). **How to Hack a Turned-Off Computer, or Running Unsigned Code in Intel Management Engine.** Black Hat Europe. London

(2) Start der Firmware und Selbsttest (1/2)

- Nach dem Einschalten des Computers wird die Firmware gestartet
  - Ältere Computer mit x86-kompatiblen Prozessoren von Anfang der 1980er Jahre bis Ende der 2000er Jahre haben als Firmware ein **BIOS** (Basic Input/Output System)
  - Neuere Computer haben ein **UEFI** (Unified Extensible Firmware Interface)

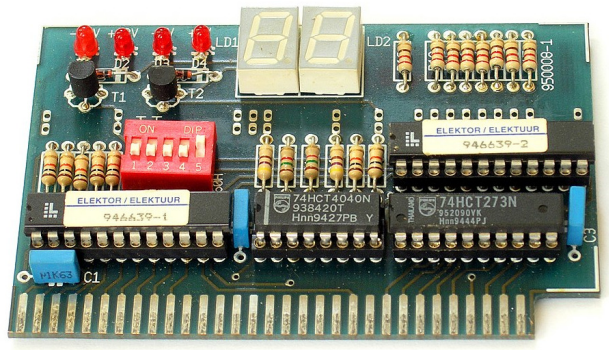


BIOS eines Thinkpad X240

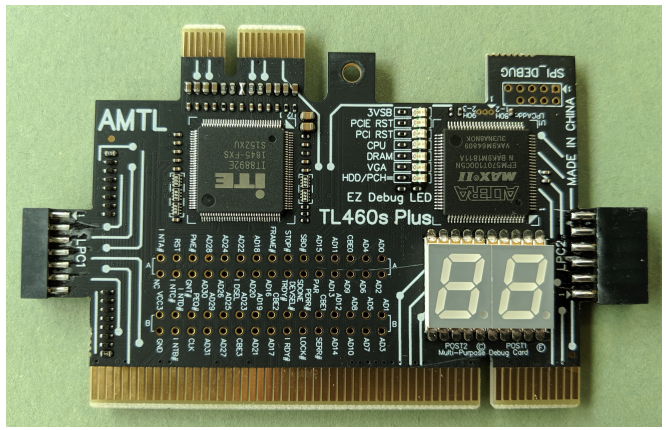
UEFI eines ASUS Z87-C

(2) Start der Firmware und Selbsttest (2/2)

- Die Firmware...
  - führt den Selbsttest **POST** (Power-on self-test) durch
    - Dabei werden u.a. die korrekte Funktion des Prozessors, des Pufferspeichers (Cache) und das Hauptspeichers überprüft
    - Es wird auch das Vorhandensein einer Hardware zur grafischen Ausgabe sowie von Ein-/Ausgabegeräten und Speicherlaufwerken überprüft
- Status- und Fehlermeldungen werden während des POST auf dem Bildschirm oder durch akustische Signale (Pieptöne) mitgeteilt
- Für die meisten Computer gibt es Diagnosekarten (**POST cards**) um den Ablauf des Selbsttests zu überwachen



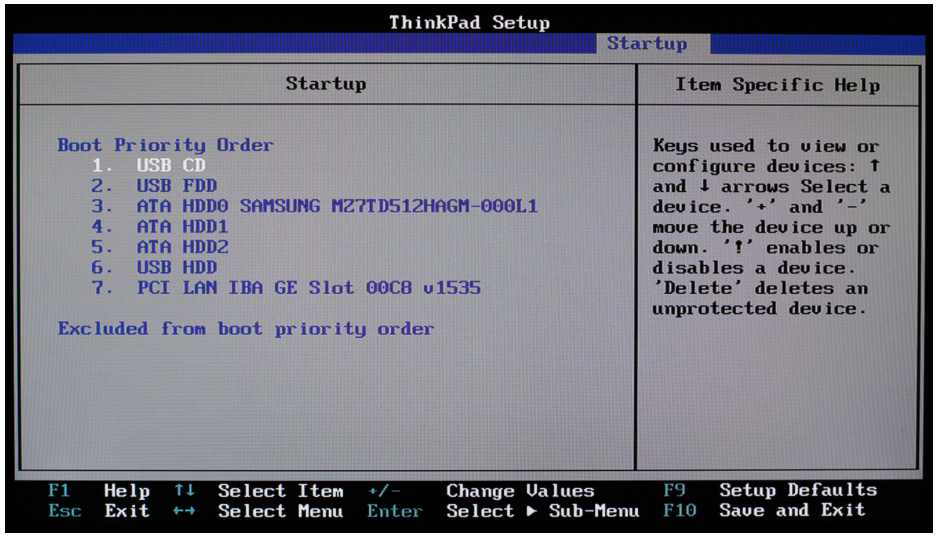
POST card for PCI
Bildquelle: Jahoe. Wikimedia (CC0)



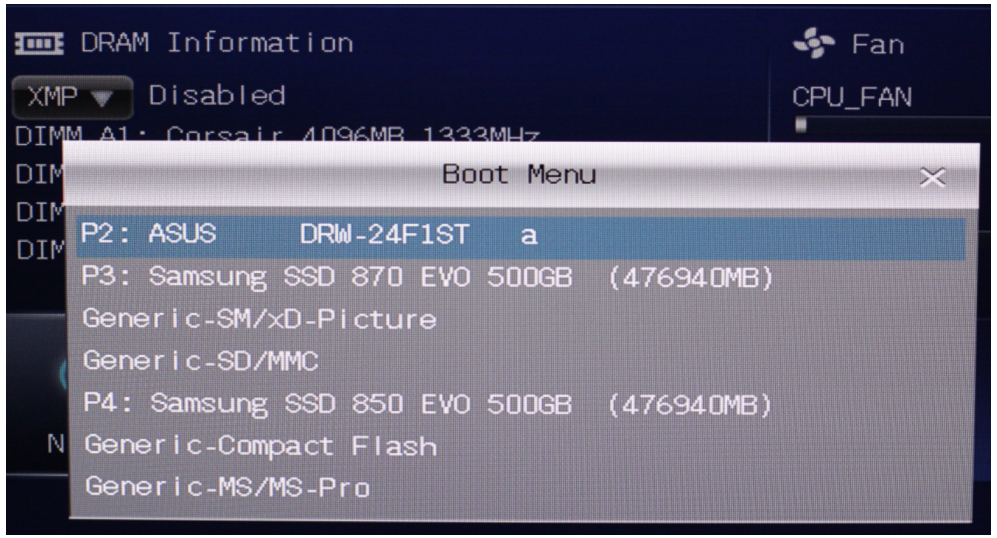
POST card for PCI, PCIe and LPC
Bildquelle: Markus Kuhn. Wikimedia (CC0)

(3) Start des Bootloaders (1/4)

- Nach dem Start des Computers und dem erfolgreichen Selbsttest sucht die Firmware nach dem ersten **Bootgerät** (Bootlaufwerk)
  - Die Reihenfolge der Laufwerke ergibt sich aus der vom Benutzer in der Firmware definierten Boot-Reihenfolge oder entspricht der standardmäßigen Boot-Reihenfolge
  - Das Bootgerät kann ein Laufwerk (z.B. SSD, Festplatte, USB-Speicherstick) oder eine Netzwerkressource sein ($\Longrightarrow$ Protokoll PXE = **Preboot Execution Environment**)



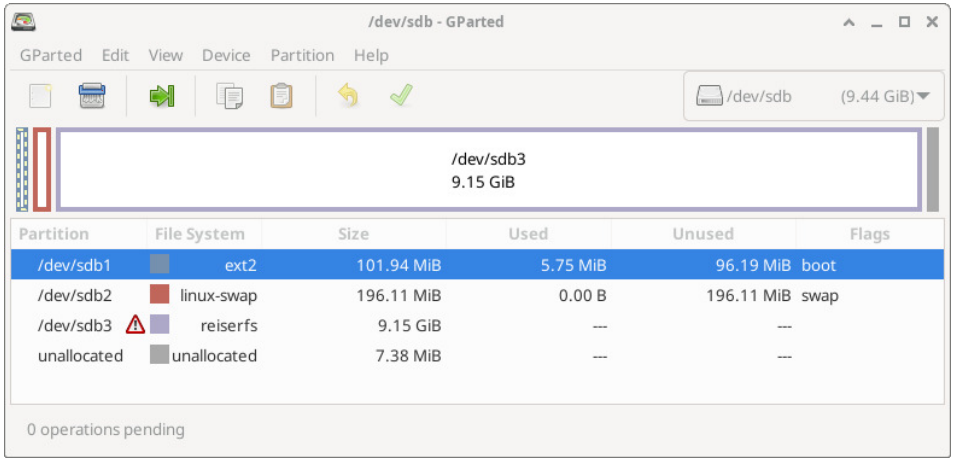
BIOS eines Thinkpad X240



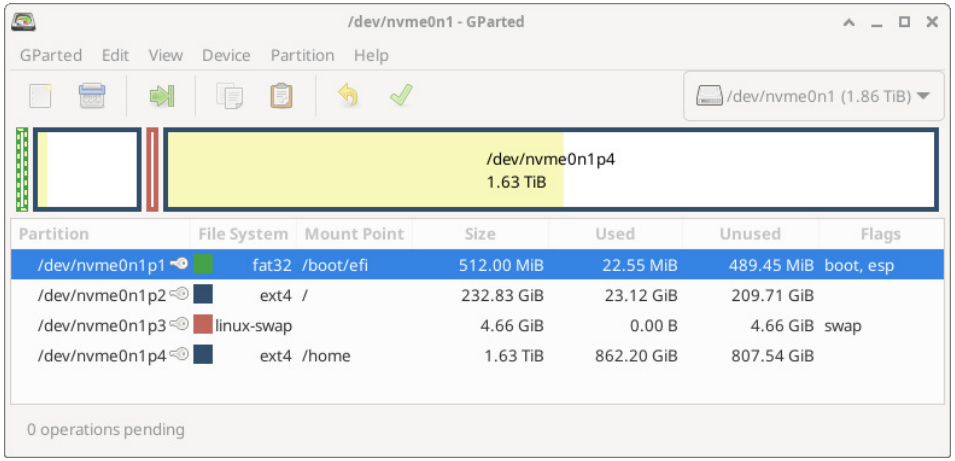
UEFI eines ASUS Z87-C

(3) Start des Bootloaders (2/4)

- Die Firmware startet den **Bootloader** vom ausgewählten Bootgerät
  - Dabei handelt es sich um ein Programm, das das Betriebssystem lädt
  - In welchem Bereich des Bootlaufwerks der Bootloader liegt, hängt vom verwendetem **Partitionsschema**
    - Bei Nutzung einer klassischen PC-Partitionstabelle liegt der Bootloader innerhalb des 512 Bytes großen *Master Boot Record* (MBR)
    - Bei Nutzung einer GUID Partition Table (GPT) liegt der Bootloader in der ESP (EFI System Partition)



Der MBR ist hier nicht sichtbar!

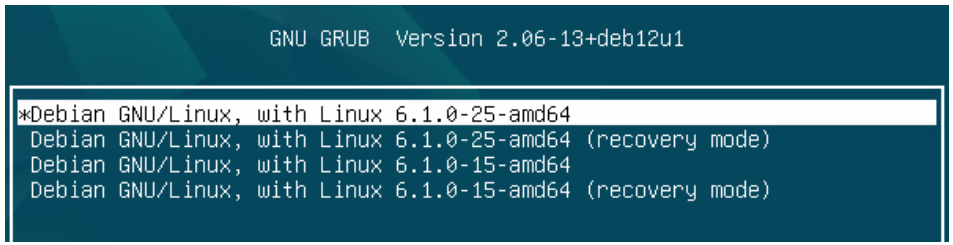


Der ESP ist hier sichtbar

Informationen zum MBR: `https://knowitlikepro.com/understanding-master-boot-record-mbr/`
Informationen zum ESP (S.117+118): `https://uefi.org/sites/default/files/resources/UEFI%202_5.pdf`

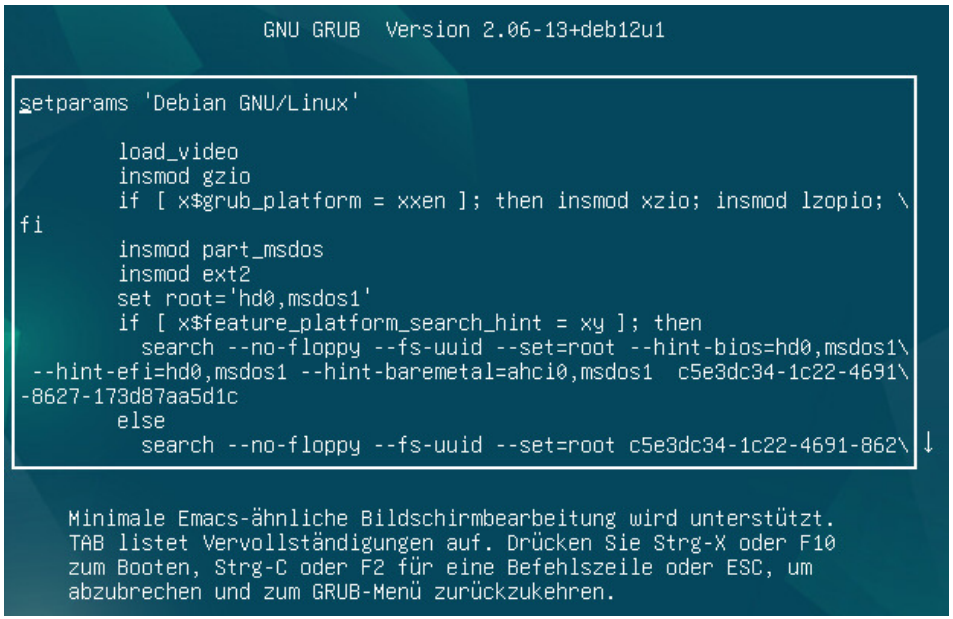
(3) Start des Bootloaders (3/4)

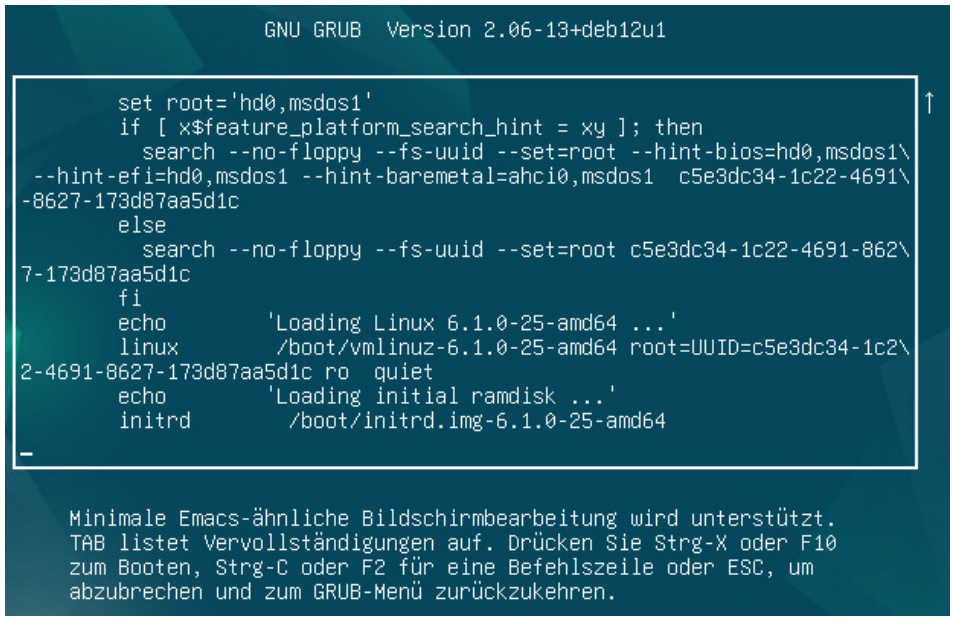
- Die Firmware lädt den Bootloader vom Bootlaufwerk und schreibt ihn in den Arbeitsspeicher
  - Einige Bootloader: **GRUB**, **Windows Boot Manager**, **Clover**,. . .
- Moderne Bootloader wie GRUB und Clover ermöglichen den Benutzern den Start des Betriebssystems mit verschiedenen Parametern oder in einem abgesicherten Modus
- Beim Vorhandensein mehrerer Betriebssysteme auf dem Computer erlauben moderne Bootloader auch die Auswahl eines dieser Betriebssysteme



(3) Start des Bootloaders (4/4)

- GRUB bietet sogar einen einen Kommandozeileninterpreter, die sogenannte GRUB-Shell
  - Diese stellt Informationen und Kommandozeilenwerkzeug zur Verfügung, um aus GRUB heraus die Konfiguration des Bootloaders zu reparieren und den Bootprozess manuell zu steuern





(4) Kernel und temporäres Root-Dateisystem starten

- Hat der Benutzer manuell oder der Bootloader automatisch ein Betriebssystem bzw. einen Kern ausgewählt, entpackt der Bootloader den Kern und lädt ihn in den Arbeitsspeicher

Der Kern liegt bei Linux typischerweise als komprimierte Datei mit dem Dateinamen `vmlinuz-<Version>-<Architektur>` im Order `/boot`. Bei der Microsoft Windows NT-Familie heißt die Datei `Ntoskrnl.exe` und sie liegt im Ordner `\Windows\System32`

- Danach lädt der Bootloader die initiale RAM-Disk (`initrd`) oder das initiale RAM-Dateisystem (`initramfs`) in den Hauptspeicher
  - Es ist ein temporäres, in den Arbeitsspeicher geladenes Root-Dateisystem
  - Das Root-Dateisystem (Wurzelverzeichnis) ist unter Linux/UNIX mit dem Schrägstrich bzw Slash (/) gekennzeichnet
    - Das durch `initrd` oder `initramfs` geladene temporäre Root-Dateisystem realisiert eine minimale Linux-Umgebung im Arbeitsspeicher
    - Sie dient in erster Linie dazu, dem Kern weitere Gerätetreiber, Treiber für Dateisysteme und Programme bereitzustellen, um das echte Root-Dateisystem des Betriebssystems in den Arbeitsspeicher zu laden

Die initiale RAM-Disk liegt bei Linux-Betriebssystemen typischerweise ebenfalls als komprimierte Datei im Order `/boot` und hat den Dateinamen `initrd.img-<Version>-<Architektur>`

(5) Einbindung des echten Root-Dateisystems

- Aus dem **temporären Root-Dateisystem** heraus greift der Kern auf das Laufwerk (in der Regel SSD oder HDD) mit dem **echten Root-Dateisystem** zu
  - Der Kern überprüft die Konsistenz (Fehlerfreiheit) des Root-Dateisystems und korrigiert ggf. Fehler im Dateisystem
- Danach bindet der Kern...
  - das richtige Root-Dateisystem ein (*mount*-Vorgang) und ersetzt damit das temporäre Root-Dateisystem
  - weitere eventuell vorhandene Dateisysteme ein (z.B. `/home`)

(6) Start von init/systemd und der Systemprozesse

- Nach dem Start des Kerns und der Einbindung des Root-Dateisystems startet der Kern `init` als ersten Prozess im Benutzermodus
  - Der Prozess `init` hat die Prozess-ID 1
  - Von `init` stammen alle weiteren Prozesse im Benutzermodus ab ($\Longrightarrow$ Foliensatz 7)
- In dieser Phase des Bootvorgangs werden auch zahlreiche Systemprozesse und Dienste (Cron-Daemon, SSH-Server, Webserver,...) automatisiert gestartet
  - Bis Ende der 2000er Jahre verwendeten Linux-Betriebssysteme wie viele andere UNIX-Betriebssysteme eine Implementierung von `init` im Stil des Standards System V, die auch **sysvinit** heißt
  - Seit Anfang der 2010er Jahre verwenden die meisten populären Linux-Distribution das moderne **systemd**

Vorteile von systemd sind u.a. ein schnellerer Systemstart durch den parallelisierten Start der Systemprozesse (Dienste), ein integriertes Logging-System namens `journald` zur einheitlichen Protokollierung und Analyse von Ereignissen, und die Fähigkeit ausgefallene Dienste automatisch neu zu starten

(7) Übergabe der Kontrolle an die Benutzer (1/2)

- Als abschließenden Schritt des Bootvorgangs übergibt der Kern die Kontrolle an die Benutzer und deren Prozesse im Benutzermodus
  - Der Kern läuft weiter im Hauptspeicher im **Kernelmodus** ($\Longrightarrow$ Foliensatz 5) und verwaltet die Hardwareressourcen und Systemaufrufe ($\Longrightarrow$ Foliensatz 7)
- In diesem Schritt werden auch die `getty`-Prozesse gestartet
  - Diese ermöglichen eine textbasierte Anmeldung der Benutzer über eine oder mehr **(virtuelle) Konsolen**
    - Für die virtuellen Konsolen (TTY1 bis TTY6) startet das Betriebssystem jeweils eine eigene Instanz des Prozesses `getty`

```
$ ps ax | grep getty
   1533 tty1     Ss+    0:00 /sbin/agetty -o -p -- \u --noclear - linux
  32078 tty2     Ss+    0:00 /sbin/agetty -o -p -- \u --noclear - linux
  32095 tty3     Ss+    0:00 /sbin/agetty -o -p -- \u --noclear - linux
  32098 tty4     Ss+    0:00 /sbin/agetty -o -p -- \u --noclear - linux
  32100 tty5     Ss+    0:00 /sbin/agetty -o -p -- \u --noclear - linux
  32102 tty6     Ss+    0:00 /sbin/agetty -o -p -- \u --noclear - linux
  32510 pts/9    S+     0:00 grep getty
```

- Typische Linux-System haben sechs virtuelle Konsolen, die über die Tasten `Strg+Alt+F1` bis `Strg+Alt+F6` aufrufbar sind
- Die virtuellen Konsolen heißen aus historischen Gründen **TTY** (Teletypewriter = Fernschreiber)

(7) Übergabe der Kontrolle an die Benutzer (2/2)

- In jeder virtuellen Konsole startet das Betriebssystem nach der erfolgreichen Anmeldung eine **Shell**, als Benutzerschnittstelle, um auf das System über die Kommandozeile zuzugreifen
  - Beispiele für Shells sind `bash`, `fish`, `ksh`, `csh`, `tcsh` oder `zsh`
- Verwendet das Betriebssystem einen **grafischen Login-Manager** (z.B. GDM, LightDM oder XDM), wird auch dieser in diesem letzten Schritt des Bootvorgangs gestartet

Üblicherweise läuft der grafischen Login-Manager auf TTY7 und kann dementsprechend auf vielen Linux-Distribution mit der Tastenkombination `Strg+Alt+F7` aufgerufen werden

- Nach der Anmeldung über den grafischen Login-Manager wird die Desktop-Umgebung geladen, die dem Benutzer eine grafische Benutzerschnittstelle bereitstellt
  - Beispiele für Desktop-Umgebungen sind: XFCE, GNOME, KDE, Window Maker oder Enlightenment