



**Hochschule  
Augsburg** University of  
Applied Sciences

# Vorlesung: Betriebssysteme

## Grundlagen

Prof. Dr. Lothar Braun, Dr.-Ing. Volodymyr Brovkov  
Sommersemester 2024

Hardware und Steuerung von Hardware

Aufgaben und Bestandteile eines Betriebssystems

Grundstrukturen von Betriebssystemen

Hardware und Steuerung von Hardware

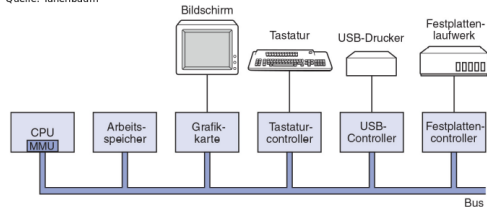
Aufgaben und Bestandteile eines Betriebssystems

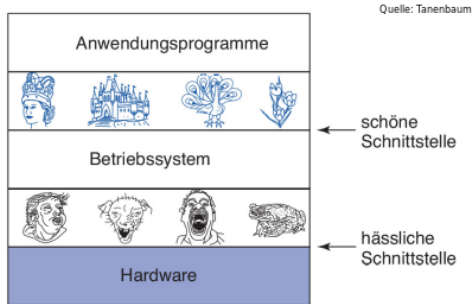
Grundstrukturen von Betriebssystemen

- Ein oder mehr Prozessoren
- Hauptspeicher
- Festplatten und/oder SSDs
- Tastatur, Maus oder Touchpad
- Grafikkarten
- Einen oder mehrere Bildschirme
- Sonstige Geräte wie Drucker

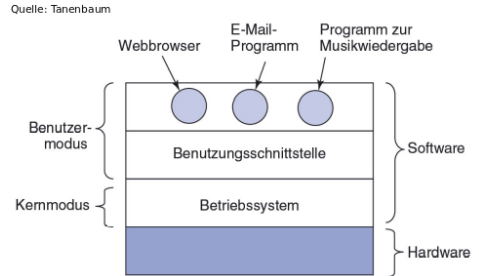
## Interaktion der einzelnen Bestandteile über Bus-Systeme

Quelle: Tanenbaum



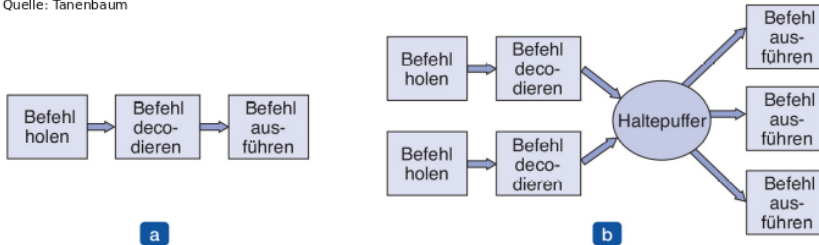


Abstraktion der Hardware durch das Betriebssystem



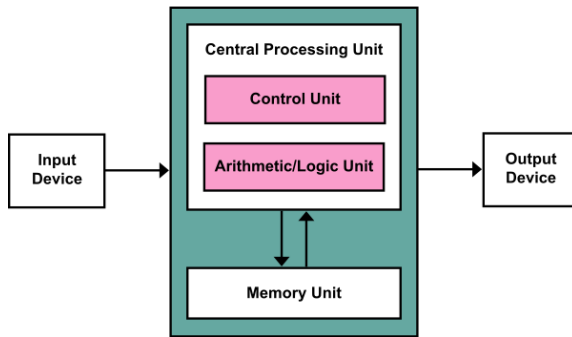
Zusammenspiel zwischen Betriebssystem, Anwendungen, und Hardware

Quelle: Tanenbaum



a) Dreistufige Pipeline, b) superskalare CPU

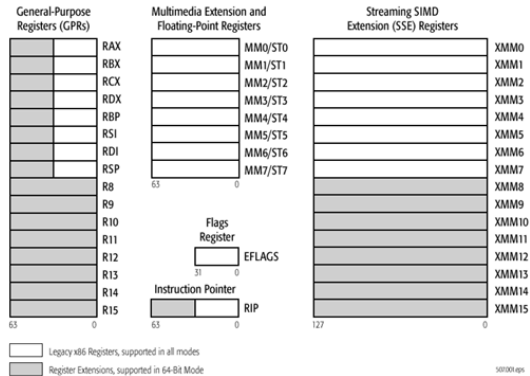
- **Frage: Woher weiß die CPU welcher Code wie auszuführen ist?**



Schematischer Aufbau einer CPU

- Daten und Code liegen im Speicher (RAM) → es gibt keine physikalische Trennung zwischen Code und Daten
- Interpretation der Bits im RAM entscheidet ob Inhalt des Speichers Daten oder Code sind

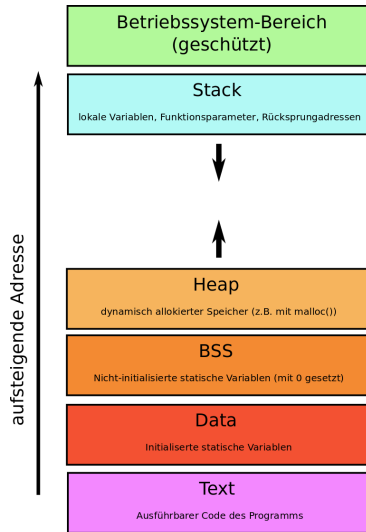
- CPU hat lokalen Datenspeicher: Die Register



Schematische Darstellung wichtiger Register in 64Bit - X86 CPUs)



- **Register: *Instruction Pointer***
  - Zeigt auf eine Adresse im Hauptspeicher
  - CPU holt Daten von dieser Adresse und führt diese aus (Fetch, Decode, Execute)
  - Instruction Pointer wird auf nächste Adresse gesetzt → Nächste Instruktion wird ausgeführt
- **Verschiedene Operationen existieren (CPU-Abhängig)**
  - Operationen zum Laden von Daten aus dem RAM in Register
  - Operationen zur Berechnung (z.B. Addition) von Daten in Registern
  - Operationen die Kontrollfluss anpassen (z.B. Sprünge)
  - ...
- **Daten und Code im Speicher können beliebig gemischt werden**
  - Die CPU führt aus worauf auch immer der Instruction Pointer zeigt
  - In der Praxis: Speicher wird in logische Segmente für unterschiedliche Daten aufgeteilt



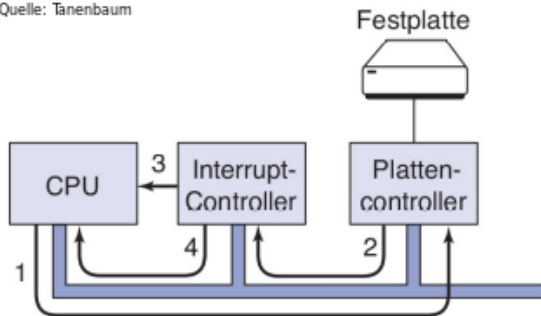
- **Assembler: Erlaubt Programmierung mit CPU-spezifischen Instruktionen**
  - Verschiedene Syntax für verschiedene CPU-Architekturen notwendig (da z.B. unterschiedliche Instruktionen für x86 und ARM existieren)
  - Teilweise verschiedene Syntax bei einer Architektur (z.B. AT&T Syntax vs. Intel Syntax)
- Beispiel für Assembler in (AT&T-Syntax)

```
# Example to move a value into a register
.section .data # Section that contains data
value:
    .int 1
.section .text          # Section that contains executable code
.globl _start            # entry point for execution
_start:                  # label that is
    nop                  # Assembler: Do nothing
    movl value, %ecx     # Move content from "value" to register ecx
    movl $1, %eax        # Move value "1" into register eax
    movl $5, %ebx        # Move value "5 " into ebx
    int $0x80             # perform interrupt
```

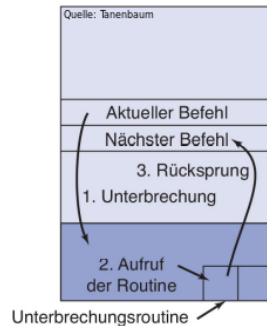
- **Option 1: Ansteuerung über spezielle CPU-Befehle**
  - CPU kann mit speziellen Befehlen auf Geräte zugreifen
  - Beispiele:
    - *int*: Spezielle Interrupts triggern, die Gerät ansprechen
    - *IN, OUT*: Schreiben von I/O Ports von Geräten
- **Option 2: Memory Mapped I/O**
  - Kommunikation bei Zugriff auf spezielle Adressen im Hauptspeicher
  - Speicher- bzw. Ladeoperationen auf diese Adressen gehen nicht in den Hauptspeicher sondern an ein Gerät
- **Hardware gibt vor wie zu kommunizieren ist**

- **Laden von Daten wird asynchron durchgeführt**
  - Während der Ladezeiten wird anderes Program ausgeführt
  - Hardware sendet Interrupt sobald Daten geladen wurden

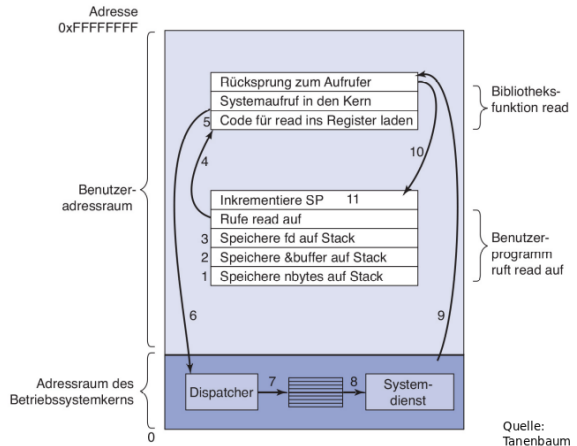
Quelle: Tanenbaum



Quelle: Tanenbaum



- **Anwendungsprogramme sprechen nicht direkt mit Hardware**
  - Das Betriebssystem übernimmt diese Aufgabe
  - Betriebssystem bietet Schnittstellen an
- **Zusammenspiel:**
  - Systemaufruf (*system call* durch die Anwendung)
  - Betriebssystem übernimmt Ausführung
  - Liefert Daten zurück und übergibt Ausführung an Anwendung zurück



Ablauf des Systemaufrufs `read(fd, &buffer, Bytes)`

- **Nutzung der Systembibliothek für Ausgabe auf Konsole:**

```
#include <unistd.h>
#include <string.h>

int main(int argc, char** argv) {
    char* hello="Hello world\n";
    write(1, hello, strlen(hello));
}
```

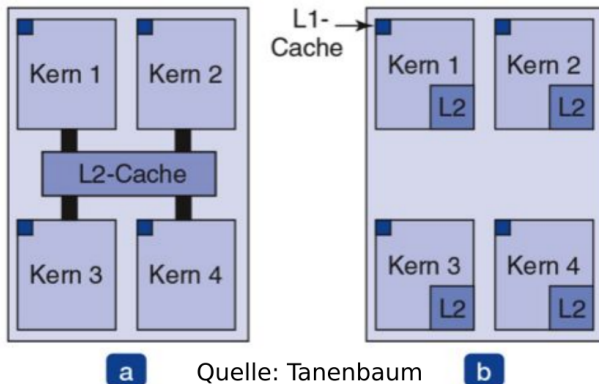
- **Was passiert:**

- `write()` ist ein Systemaufruf, der das Betriebssystem zum Schreiben aufruft
- Der erste Parameter ist *File Descriptor*: 1 ist der *Standard Output* Kanal
- Aufruf nimmt Puffer *hello* und schreibt `strlen(hello)` bytes nach *Standard Output*

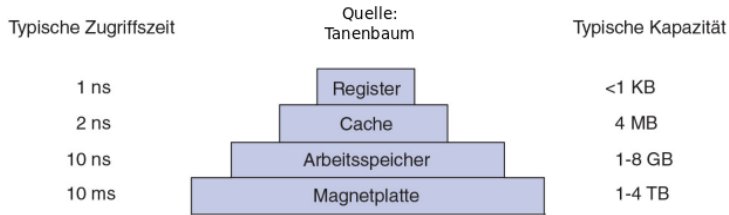
- Beispiel aus C von vorheriger Folie in Assembler mit AT&T Syntax

```
# Example: Output Hello World on Linux using
.section .data                                # Section that contains data
msg:
    .ascii "Hello, world!\n"                # Our Data String to output
    msg_len = (. - msg)                     # Length of String
.section .text                                # Start of text segment
.globl _start                                # entry point for execution
_start:                                       # label that is our start position
    nop                                     # Assembler: Do nothing
    movl $msg_len, %edx                     # Move string length to register EDX
    movl $msg, %ecx                         # Move address of string to ECX
    movl $1, %ebx                           # Move file descriptor 1 (stdout) to EBX
    movl $4, %eax                           # Move System call number for sys_write to EAX
    int $0x80                               # Call operating system to handle system call
```





- a) 4-Kern Chip mit geteiltem CPU cache
- b) 4-Kern Chip mit separaten L2-Caches



- **Betriebssystem muss Daten zwischen Speichern austauschen**
  - Ziel: Daten möglichst in schnellstem Speicher vorhalten
  - Aktionen die Transfer zwischen Speichern benötigen sind potentiell langsam

- **Prozess ist ein Programm in Ausführung**
  - Verwaltung des Zustandes laufender Programme
- **Jedem Prozess wird Adressraum zugeordnet**
  - Dort darf der Prozess lesen und schreiben
  - Beinhaltet Programm, Daten und den Stack
- **Prozesse können weitere Prozesse erzeugen**
  - Betriebssystem steuert Prozesse
  - Bestimmt welcher Prozess die CPU beanspruchen darf

- **Jeder Prozess bekommt eigenen Adressraum**
  - Mehrere Prozesse können gleichzeitig im Speicher sein
  - Prozesse können unabhängig voneinander auf Speicher zugreifen
- **Hardware bietet Schutzmechanismen**
  - Kein Zugriff auf Adressräume anderer Prozesse
  - Betriebssystem muss Speichermanagement implementieren
- **Prozesse können mehr Speicher verwenden als verbaut**
  - Virtueller Speicher und Ein-/Auslagerung
  - Betriebssystemaufgabe: Speicherverwaltung

- **Definition von Systemaufrufen notwendig**
  - Aktionen die vom Betriebssystem für eine Anwendung ausgeführt werden
  - Definition von Parametern für Systemaufrufe
  - Berechtigungskonzept: welche Anwendung darf welche Aktionen ausführen?

## Dateiverwaltung

Aufruf	Beschreibung
<code>fd = open(file, how, ...)</code>	Datei zum Lesen, Schreiben oder für beides öffnen
<code>s = close(fd)</code>	Offene Datei schließen
<code>n = read(fd, buffer, nbytes)</code>	Daten aus Datei in Puffer lesen
<code>n = write(fd, buffer, nbytes)</code>	Daten vom Puffer in Datei schreiben
<code>position = lseek(fd, offset, whence)</code>	Dateipositionszeiger bewegen
<code>s = stat(name, &amp;buf)</code>	Status einer Datei ermitteln

Beispiele für Systemaufrufe zur Dateiverwaltung, Quelle: Tanenbaum

- **Betriebssysteme können Systemaufrufe frei definieren**
  - **Auswirkung:** Keine Portabilität zwischen Betriebssystemen
- **Standardisierung**
  - Für UNIX Systeme wurde ein gemeinsamer Standard für Betriebssysteme geschaffen
  - Definiert gemeinsame Schnittstelle und Verhalten des Betriebssystems
  - Wird heute auch von nicht-UNIX Systemen unterstützt

UNIX	Win32	Beschreibung
fork	CreateProcess	Erzeugen eines neuen Prozesses
waitpid	WaitForSingleObject	Warten auf das Ende eines Prozesses
execve	(nicht vorhanden)	CreateProcess = fork + execve
exit	ExitProcess	Ausführung beenden
open	CreateFile	Erzeugen einer Datei oder Öffnen einer existierenden Datei
close	CloseHandle	Datei schließen
read	ReadFile	Daten aus einer Datei lesen
write	WriteFile	Daten in eine Datei schreiben
lseek	SetFilePointer	Dateizeiger bewegen
stat	GetFileAttributesEx	Dateiattribute erfragen
mkdir	CreateDirectory	Erzeugen eines neuen Verzeichnisses

Unterschiede Systemaufrufe. Quelle: Tanenbaum

**Beschreiben die Begriffe *Programm* und *Prozess* das Gleiche?**

- A)** Ja
- B)** Nein

Hardware und Steuerung von Hardware

Aufgaben und Bestandteile eines Betriebssystems

Grundstrukturen von Betriebssystemen



- **Ausführen von Programmen**

- **Ausführen von Programmen**
- **Verwalten von Umgebungen**
  - Prozesse
  - Benutzer

- **Ausführen von Programmen**
- **Verwalten von Umgebungen**
- **Verwalten von Ressourcen**

**Aufteilung von:**

CPUs  
Speicher  
Ein-/Ausgabegeräten  
Dateien

**... auf**

Benutzer  
laufende Prozesse

**... unter Berücksichtigung von**

Effizienz, Fairness, Sicherheit und ggf. speziellen Anforderungen wie Echtzeit

- **Ausführen von Programmen**
- **Verwalten von Umgebungen**
- **Verwalten von Ressourcen**
- **Ein- und Ausgabe**
  - Steuerung der Hardware (Treiber und Modelle)
  - Bereitstellen eines Dateisystems
  - Schnittstellen zur Kommunikation über Netzwerk

- **Ausführen von Programmen**
- **Verwalten von Umgebungen**
- **Verwalten von Ressourcen**
- **Ein- und Ausgabe**
- **Fehlerbehandlung**
  - Fehler in Anwendungsprogrammen
  - Behandlung von Hardware-Fehlern
  - Fehler im Betriebssystem selbst

- **Ausführen von Programmen**
- **Verwalten von Umgebungen**
- **Verwalten von Ressourcen**
- **Ein- und Ausgabe**
- **Fehlerbehandlung**
- **Accounting**
  - Sammeln von Statistiken zur Auslastung
  - Messungen der Performance
  - Messungen des Verbrauchs von Ressourcen

- **Ausführen von Programmen**
- **Verwalten von Umgebungen**
- **Verwalten von Ressourcen**
- **Ein- und Ausgabe**
- **Fehlerbehandlung**
- **Accounting**
- **Start/Stop des Systems**

- **Ablaufsteuerung**
  - Prozess- und Thread-Modelle
  - Scheduling (auf Ein- oder Mehr-Prozessorsystemen)
  - Interrupts und Exceptions
  - Synchronisation zwischen Prozessen und Threads



- **Ablaufsteuerung**
- **(Virtuelle) Speicherverwaltung**
  - Verwaltung des vorhandenen Speichers
  - Bereitstellung eines geschützten Addressraums für Prozesse
  - Optimierung der Speichernutzung
    - Swapping
    - Caching von Dateien
    - Vermeidung von Kopien in Speicher (z.B. Anwendungen die mehrfach geladen werden, sollten nur einmal Speicher benötigen)

- **Ablaufsteuerung**
- **(Virtuelle) Speicherverwaltung**
- **Schnittstelle für Systemaufrufe**
  - Implementierung eigener oder standardisierter Interfaces (z.B. POSIX)

- **Ablaufsteuerung**
- **(Virtuelle) Speicherverwaltung**
- **Schnittstelle für Systemaufrufe**
- **Datei-Systeme**
  - Anbieten eines oder mehrerer Dateisysteme für Systeme
  - Typisches Modell: Schichtenmodell
    - Gemeinsame Module für Plattenzugriff oder Caching
    - Einzelne Module für spezielle Datei-Systemstandards (z.B. NTFS, vfat, ext4, ...)

- **Ablaufsteuerung**
- **(Virtuelle) Speicherverwaltung**
- **Schnittstelle für Systemaufrufe**
- **Datei-Systeme**
- **Treiber**
  - Treiber sind für die Steuerung einzelner Geräte zuständig
  - Jede Hardware benötigt einen eigenen Treiber (z.B. auch für ähnliche Hardware anderer Hersteller)
  - Schnittstellen zur Entwicklung von Treibern

- **Ablaufsteuerung**
- **(Virtuelle) Speicherverwaltung**
- **Schnittstelle für Systemaufrufe**
- **Datei-Systeme**
- **Treiber**
- **Sicherheitsmechanismen**
  - Trennung von Prozessen
  - Rechteverwaltung auf Dateisystemen
  - Verschlüsselung von Dateisystemen
  - ...

Hardware und Steuerung von Hardware

Aufgaben und Bestandteile eines Betriebssystems

Grundstrukturen von Betriebssystemen

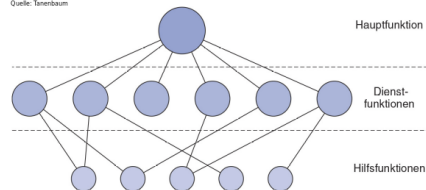
- **Das gesamte Betriebssystem ist ein Programm**

- Funktionen werden als Einzelfunktionen implementiert
- Jeder Teil des Betriebssystem kann jede Funktion aufrufen
- Kein Hardware-Schutz zwischen Teilen des Betriebssystem
- Systemaufruf durch Spezialbefehle (*traps*)

- **Auswirkungen:**

- Sicherheit und Zuverlässigkeit nur falls gesamtes System korrekt
- Wartung und Erweiterung aufwändig
- Aber: Effizient

Quelle: Tanenbaum



**Gibt es heutzutage überhaupt noch Monolithische Betriebssysteme?**

- A)** Ja
- B)** Nein

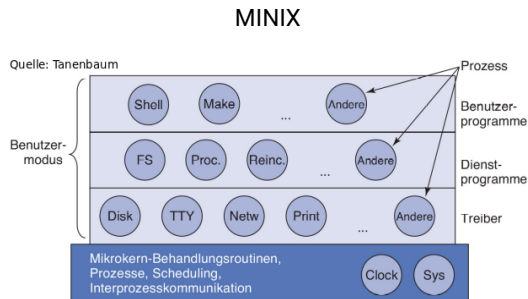


- **Kern übernimmt nur zentrale Aufgaben:**

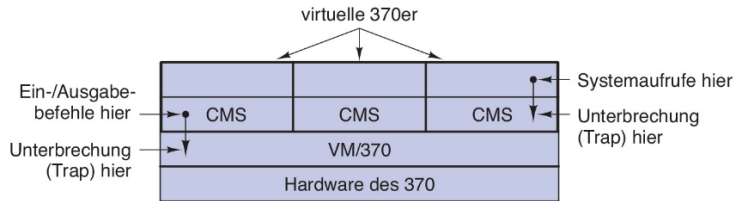
- Prozess-/Thread-Verwaltung
- Botschaftenverwaltung
- Interrupts und Exceptions
- Virtuelle Speicherverwaltung
- Physikalische Ein-/Ausgabe

- **Andere Aufgaben in Module**

- Prozesse im Benutzer-Modus
- Nur Mikro-Kern im Kern-Modus
- Systemaufrufe über Botschaften
- Größere Stabilität da Prozesse im Fehlerfall neu gestartet werden können



- **Virtualisierung bietet Hardware für parallelen Betrieb mehrerer Betriebssysteme**
- **Aufteilung der Betriebssystem-Funktionalitäten:**
  - Starke Trennung der Umgebung einzelner virtueller Maschinen
  - Jeder Maschine wird "eigene" *virtuelle* Hardware zur Verfügung gestellt



Virtualisierung bei IBM VM/370 mit CMS (*Conversational Monitor System*). Quelle: Tanenbaum

- **Einsparung von Kosten durch effizientere Nutzung von Hardware**
  - Viele Dienste (z.B. FTP-Server, Web-Server, ...) lasten Server-Systeme nicht aus
  - Virtuelle Maschinen auf einem Server erlauben Teilung von Ressourcen
  - Virtualisierung sorgt für starke Trennung der Dienste
- **Cloud-Dienste und Web-Hosting**
  - Anbieter können kostengünstige Angebote zum Hosting von Diensten anbieten
  - Starke Trennung der Daten verschiedener Kunden durch starke Trennung der virtuellen Maschinen
- **Mehrere Betriebssysteme auf einem PC**
  - Zum Beispiel eigene virtuelle Maschine für Praktika in Vorlesungen

- **Emulatoren**

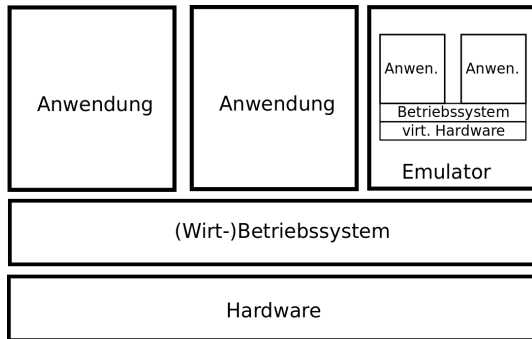
- Laufen als Prozesse im Betriebssystem
- Emulieren die gesamte Hardware eines Systems

- **Hardware Virtualisierung**

- Die komplette Hardware, d.h. CPU und Peripherie-Geräte, werden durch sogenannte Hypervisoren an verschiedene Systeme virtualisiert
- CPU-Instruktionen der virtuellen Maschinen werden direkt ausgeführt
- Privilegierte Instruktionen werden vom Hypervisor abgefangen und emuliert
- **Unterstützung im Prozessor notwendig**

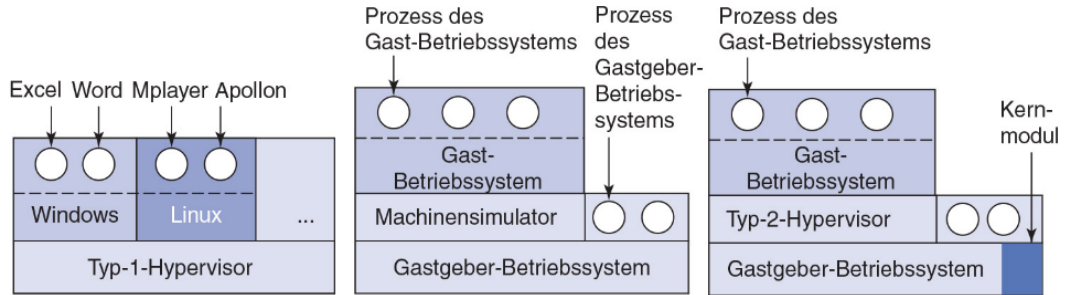
- **Paravirtualisierung**

- Ähnlich zu Hardware-Virtualisierung
- Unterschied: Gast-Betriebssystem muss modifiziert werden, so dass bei Peripherie-Hardware nicht auf Hardware zugegriffen wird, sondern auf Funktionen des Hypervisors



- Vollständige Emulation der Hardware
- **Vorteil:**
  - Keine Prozessor-Unterstützung notwendig
  - Keine Anpassung des Gast-Betriebssystems notwendig
- **Nachteil:**
  - Schlechtere Performance als mit Hardware-Virtualisierung

Quelle: Tanenbaum



Darstellung verschiedener Hypervisoren:  
Typ-1 Hypervisor, klassischer Typ2-Hypervisor, Typ2-Hypervisor in der Praxis.  
Quelle: Tanenbaum

- **Klarerer Strukturierung des Systems**

- Jede Schicht erfüllt gewisse Aufgaben
- Es bestehen klare Aufrufbeziehungen
- Es besteht die Möglichkeit die Funktion der Schichten gegenseitig unter Verwendung von Hardware-Schutz abzusichern
- Systemaufrufe über *Traps* auch innerhalb des Betriebssystems

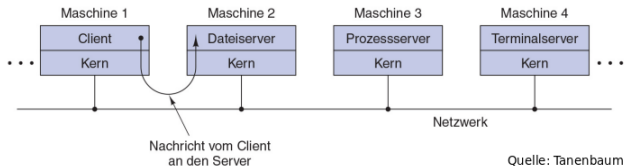
Beispiel eines Schichtensystems im Betriebssystem THE

Schicht	Funktion
5	Der Operator
4	Benutzerprogramme
3	Ein-/Ausgabeverwaltung
2	Operator-Prozess-Kommunikation
1	Speicherverwaltung
0	Prozessorzuteilung und Multiprogrammierung

Quelle: Tanenbaum

- **Leichte Variation des Mikrokern-Ansatzes:**

- Dienst-Prozesse des Betriebssystems werden in *Client* und *Server* eingeteilt
- *Clients* nutzen Dienste der *Server* mittels Nachrichten (*message passing*)
- Erlaubt Auslagerung einzelner Dienste auf andere Systeme des Netzwerkes

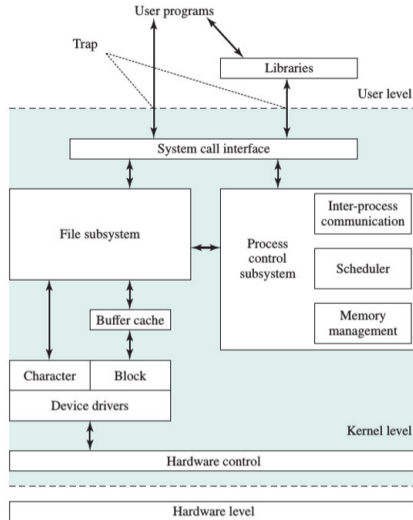


Client-Server-Modell

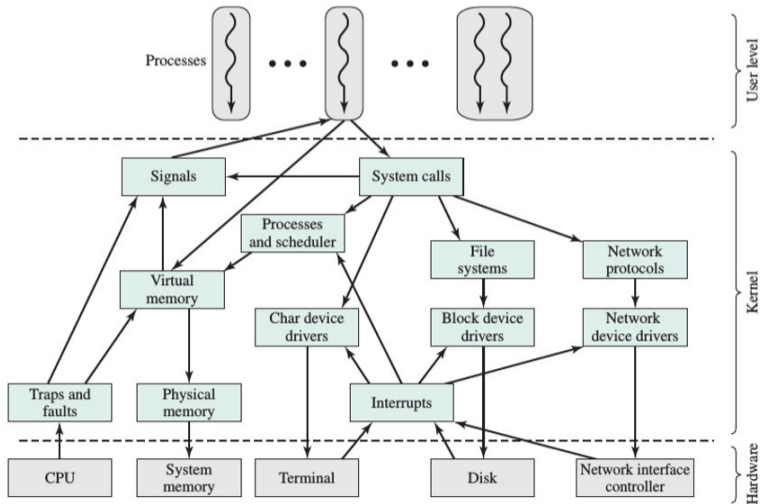
- **Exokerne**

- Exokern stellt Ressourcen bereit und verhindert Zugriff durch andere

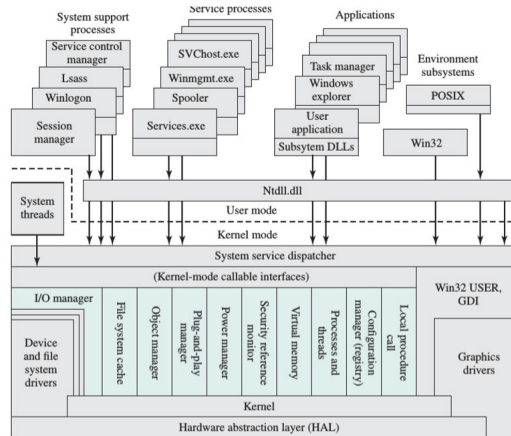




Quelle: Stallings



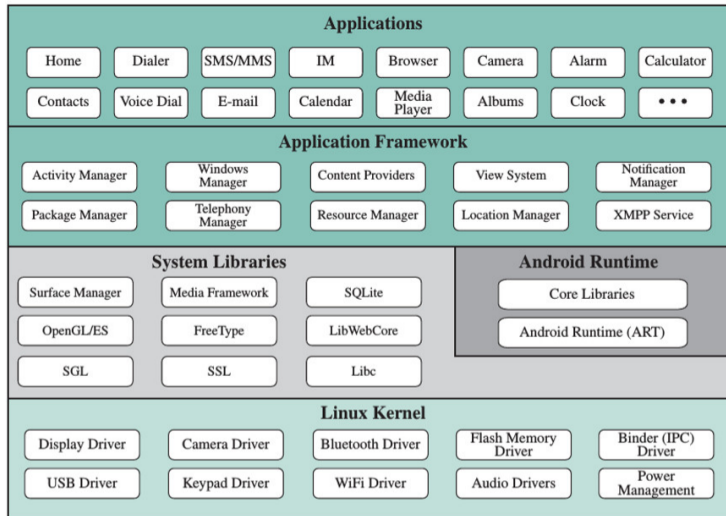
Quelle: Stallings



Lsass = local security authentication server  
POSIX = portable operating system interface  
GDI = graphics device interface  
DLL = dynamic link library

Colored area indicates Executive

Quelle: Stallings



Quelle: Stallings

- **Aufgaben des Betriebssystems**

- Verwaltung der Hardware und Bereitstellung von Ressourcen
- Bereitstellen von Diensten die für viele Anwendungen relevant sind

- **Architekturen von Betriebssystemen**

- Unterschiedliche Architekturen für Betriebssysteme existieren
- Architekturen unterscheiden sich in innerer Strukturierung und Organisation der Betriebssysteme
- Verschiedene Architekturen erlauben es eine starke oder schwache Trennung der einzelnen Bestandteile des Betriebssystems zu implementieren

Fragen?