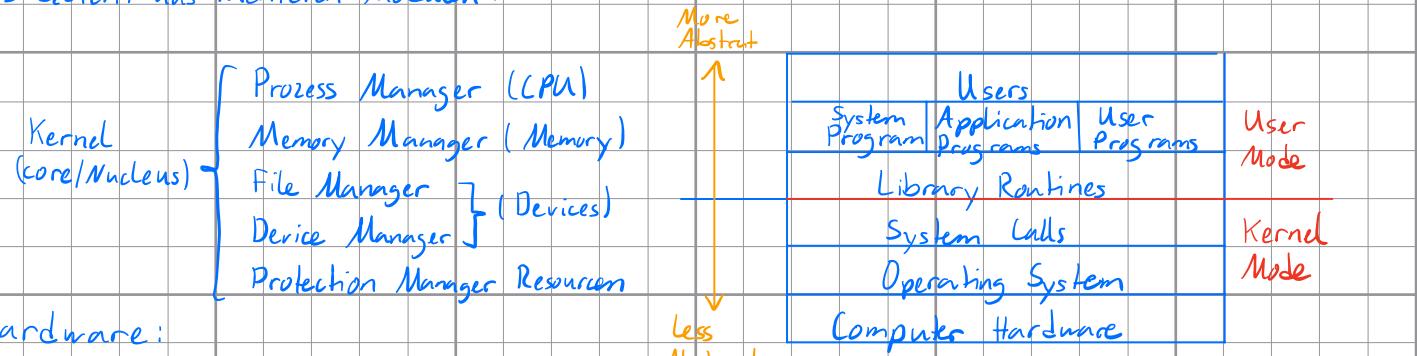


Defining OS: Betriebssystem das die Hardware eines Computers verwaltet und Ausführungen von Anwendungsprogrammen ermöglicht.

Hauptaufgaben: Prozessverwaltung, Speicherverwaltung, Dateiverwaltung, Gerätverwaltung, Benutzeroberfläche  
Os besteht aus mehreren Modulen:



Hardware:

Physische Bestandteile eines Computers:

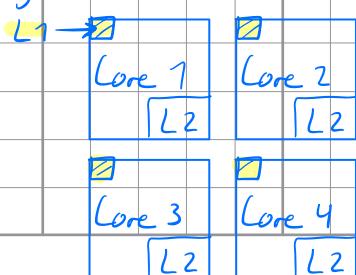
- Processor (CPU)
- Arbeitsspeicher (RAM)
- Festplatte / SSD (Speicher)
- etc.

CPU Caches

- Shared L2 Cache
  - Core teilen effizient
  - ↳ Anwendung mit hohen Daten austausch



- Separate L2 Cache
  - Alle Core haben eigenen L2 Cache + L1
  - Schneller Zugriff auf L2
  - Daten synchronisieren (synchronised)
  - Effizient bei Berechnungen, die stark parallelisiert sind und wenig datenaustausch benötigen.



Central Processing Unit (CPU):

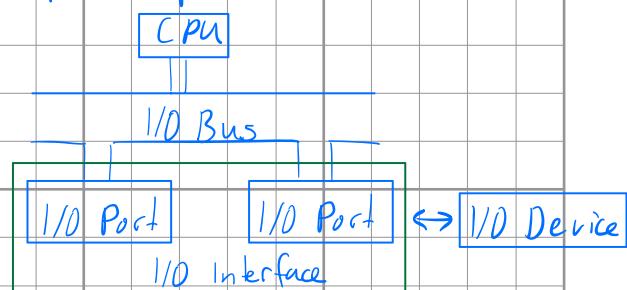
Lädt die erste Anweisung aus Speicher → dekodiert sie → führt sie aus → wiederholt diesen Vorgang für die nächste Anweisung.



CPU-Architektur & Befehlssätze

- Befehlssätze: ARM, RISC, X86
- Ausführungsprivilegien
  - Inkl: Priority (z.B. User Mode vs. Kernel Mode)
  - ARM: Privileged vs. Unprivileged Mode

Input/Output:



Software that talks to I/O Controller  
↳ called Device Driver

Operating System vs. Distributions:

- OS (Kernel) ist Grundgerüst, das Hardware und Software verbindet.
- Distributions sind komplett Betriebssysteme, die auf einem Kernel basieren und Software für den Alltags Einsatz enthalten.

<h3>BIOS (Basic Input Output System)</h3> <ul style="list-style-type: none"> <li>1) wird beim Booten aus ROM geladen</li> <li>• enthält hardware spez. I/O Software</li> <li>• aufgaben: POST (Geräteprüfung + initialisiert Gerät auswählen)</li> <li>MBR auslesen &amp; ausführen</li> </ul>	<h3>Bootloader (GRUB)</h3> <ul style="list-style-type: none"> <li>• MBR-Code lädt den Bootloader</li> <li>• Der Bootloader: findet das OS auf der Boot-Partition</li> <li>• Ladt das OS in den Speicher &amp; führt es aus</li> </ul>
<h3>OS-Initialisierung</h3> <ul style="list-style-type: none"> <li>• fragt BIOS/Bus nach HW-Infos</li> <li>• Initialisierung OS-Strukturen</li> <li>• Erstellt Systemdienste</li> <li>• Startet Benutzeroberfläche</li> </ul>	<h3>EFI Boot Manager</h3> <ul style="list-style-type: none"> <li>• verwalten</li> </ul>
<p>BIOS vs UEFI Boot</p> <p>BIOS → MBR → Bootloader → Kernel(OS)</p> <p>UEFI → EFI-Bootloader → Kernel(OS)</p>	<h3>System Initialisation</h3> <ol style="list-style-type: none"> <li>1) System startup/HW initialisation (BIOS)</li> <li>2) Bootloader (GRUB)</li> <li>3) Kernel (Linux)</li> <li>4) INIT Process (Run levels)</li> <li>5) User Prompt (Shell or GUI)</li> </ol>
<h3>Process Model</h3> <ul style="list-style-type: none"> <li>• run sequentially or parallel or both (multi programming / tasking)</li> <li>• are selected by Scheduler of the OS</li> <li>• run in UserMode / run in Foreground / Background</li> </ul>	<h3>Process Creation</h3> <ul style="list-style-type: none"> <li>• MemoryMap • Programm Counter &amp; Stack Pointer</li> <li>• OS load executable (binary) into RAM and set Scheduler, Process Table</li> <li>• From SystemMode (OS) to UserMode</li> <li>• MemoryMap ⇒ Text is area with CPU instructions, Stack is segment of memory, heap is not set to a constant size before compilation</li> </ul>
<h3>Process Creation Events</h3> <ul style="list-style-type: none"> <li>• System Boots</li> <li>• User Request ↳ cron Job</li> <li>• System Request ↳ Process creating another process</li> <li>↳ Interrupts ↳ fork() / exec() system call</li> <li>• Process hierarchy</li> <li>↳ OS creates Process with PID 1</li> <li>↳ May create "Zombies" (no parents anymore)</li> <li>• Copy-on-write</li> <li>↳ Memory Map of Parents shared by Parent or Child</li> </ul>	<h3>Process States</h3> <ul style="list-style-type: none"> <li>• running ↳ CPU assigned</li> <li>• ready ↳ Runnable waiting for first go or temp stopped (idle)</li> <li>• blocked ↳ unable to run (sleeping)</li> </ul>
<h3>Process State Change Reasons</h3> <ul style="list-style-type: none"> <li>• Timer (CPU allocation expired)</li> <li>• Interrupt</li> <li>• Page Fault</li> <li>• System Call</li> </ul>	<h3>Process State Change</h3> <ul style="list-style-type: none"> <li>• User Mode <ul style="list-style-type: none"> <li>- Instruction for Application Logic (Math)</li> <li>- Application data manipulation (read, write in Ram)</li> </ul> </li> <li>• System Mode <ul style="list-style-type: none"> <li>- Instruction for system management (CPU dispatching)</li> <li>- HW management (Interrupt handling / Device access)</li> </ul> </li> </ul>
<h3>Multi Processing</h3>	<h3>Kernel Thread</h3> <ul style="list-style-type: none"> <li>• equivalent to a process.</li> <li>• Each thread has <ul style="list-style-type: none"> <li>↳ Program counter</li> <li>↳ Registers</li> <li>↳ A Stack</li> </ul> </li> <li>• has PID, a stack, etc</li> <li>• is scheduled by scheduler</li> <li>• is uninterrupted</li> <li>• Listen to "signals" ("Start", "Stop", etc.)</li> </ul>
	<h3>Multi-Process-Multi-Threading</h3>

Scheduling  $\Rightarrow$  based on queuing theory

long CPU burst: Weniger Unterbrechungen, intensive Berechnungen, CPU-bound

Short CPU burst: Häufige Unterbrechungen durch I/O

Wartezeiten, interaktive Prozesse, I/O-bound

### Scheduling Metrics



### Batch-Systeme

- Throughput (Maximiert Anzahl erledigte Jobs/h)

- Turnaround Time (Minimierung Zeit zwischen Job-Einreichung & Abschluss)

- CPU Utilisation (CPU sollte möglichst ausgelastet sein)

### Queues

1. Scheduler sorts a queue

2. Dispatcher moves the task

from the head of the ready queue to execution on CPU  
 $\hookrightarrow$  performs the context switch

- . OS maintains several queues

### Scheduling Policies

- Allg. Prinzipien

- Fairness

- Policy Enforcement

- Balance

### Interactive Systeme

- Response time (schnell)

- Proportionality (Erfüllung Benutzererwartung)

### Echtzeitssysteme

- Meeting Deadlines (Fristen einhalten)

- Predictability (Vorhersagbares Verhalten)

### Schedulers

- Uni-processing

- Multi-processing

- Multitasking

- FIFO

### Real-Time Schedulers

- need responsiveness to I/O

- fulfil deadlines

### Rate Monotonic

- task with highest rep. rate = highest prio.

- Max. garantie Auslastung  $\sim 69\%$

- bei +70% nicht garantiert

- $\sim 30\%$  bleibt ungenutzt = teuer + ineffizient

- Round Robin: Bevorzugt

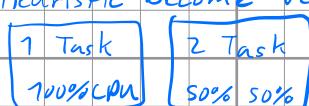
kurze Prozesse, single Queue, no starvation

- Multi-level, problem = multiple priorities, kann sein dass kleine prio. nicht ausgeführt wird.

- Heuristic

- Fair Share Scheduler, Problem:

Heuristic become very complex



- Cooperative Scheduler

Task decides when it has used enough processing time. Yields control to scheduler/named task

- Unix Scheduler

presented with unsorted ready-array CPU time called epoch

- Linux O(1) Scheduler

2 arrays active and passive

task sorted according to priority in the expired priority array  $\rightarrow O(1)$  operation

### Earliest Deadline First

- runtime scheduler

$$U = \sum_{i=1}^n \frac{C_e[i]}{Tr[i]} \leq 1$$

- Task mit nächsten Deadline wird als nächstes ausgeführt.

- prio. ist dynamisch

- garantie Terminierung:  $U \leq 1$

### Linux: Complexity Fair Scheduler (CFS)

- rot-schwarz Baum (ausbalanciert)

### Being Nice

Konzept das "nice" Werte benutzt, um zu beeinflussen, wie viel Prozessor-Zeit ein Prozess bekommt. Nur bei Prozessen die mir gehören, kann man den "nice" Wert ändern, nicht bei Fremden. Je höher der Wert, desto höher der Prozessor.  $\lceil Cgroups \Rightarrow \text{Ressourcenkontrolle} \rceil$

Gruppe von Prozessen die gemeinsame bestimmte Ressourcenlimits (CPU, RAM) unterliegen.

Subsystem = Kernel-Komponente für eine Ressource (CPU, Speicher) die das Verhalten der Cgroup-Prozesse steuert.  $\Rightarrow$  Ressourcen Kontrolle

Bsp. Controller: Begrenzung CPU Zeit

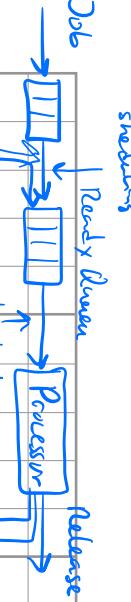
- Begrenzung des Speichers (Memory Controller)

- Erfassung CPU Nutzung

- Einfrieren und Fortsetzen von Prozessen

### Linux/Android BFS Scheduler

One single queue (Each task given a virtual deadline)



<p>Groups 1 vs 2:</p> <p>v1: Erste Version, viele unabhängige Controller → inkonsistent.</p> <p>v2: Einheitlichere Struktur, ersetzt v1 langfristig, beide Versionen können koexistieren In v2 kann ein Controller nicht gleichzeitig in V1 &amp; V2 genutzt werden.</p>	<p>V1 Konzepte</p> <p>Mounting: Controller werden auf tmpfs gemountet → schnelles virtuelles RAM Dateisystem sudo mount -t cgroupcpu none /sys/fs/cgroup/cpu Controller können einzeln oder gemeinsam auf ein Dateisystem gemountet werden. Prozesse bestehen aus Threads: V1 erlaubt getrennte Zuordnung von Threads zu Cgroups → Problematisch bei Memory-Sharing, in V2 ist das eingeschränkt. Unmounting: nur möglich wenn keine Prozesse in der Cgroup &amp; keine Untergruppen mehr vorhanden.</p>
<p>PID: when writing PID into the cgroups.procs, all threads in the process are moved into the new cgroup at once.</p>	<p>erlaubt getrennte Zuordnung von Threads zu Cgroups → Problematisch bei Memory-Sharing, in V2 ist das eingeschränkt. Unmounting: nur möglich wenn keine Prozesse in der Cgroup &amp; keine Untergruppen mehr vorhanden.</p>
<p>CPU Subsystem: Erzeugt N Prozesse auf N CPU Kerne: <math>N/2 \rightarrow</math> uneingeschränkter Zugriff <math>(N/2)/2 \rightarrow</math> nur 70% eines Kerns <math>(N/2)/2 \rightarrow</math> nur 20% eines Kerns</p>	<p>V2: alle Controller sind in einer einzigen Hierarchie gemountet. <u>mount   grep cgroup2</u> Befehl zum anzeigen</p>
<p>Kontrolle über CPU-Nutzung mit top (Spalte %CPU)</p>	<p>Erlaubt kontrolliertes Ressourcenmanagement pro Thread</p>
<p>Cgroups erstellen: mkdir /sys/fs/cgroup/devices/group0 voller Zugriff → anzeigen mit: cat /sys/fs/cgroup/devices/group0/devices.list Zugriff auf /dev/null blockieren</p>	<p>Subdivide memory in static segments Problem: internal fragmentation</p>
<p>Process der Cgroup zuweisen: echo 0 &gt; /sys/fs/cgroup/devices/group0/tasks</p>	<p>Memory Management</p>
<ul style="list-style-type: none"> <li>Fast cache in 1-3 Layers</li> <li>Main memory slower but larger</li> <li>Sekundär memory, hard-disks <ul style="list-style-type: none"> <li>- program and files</li> </ul> </li> <li>Tertiary memory <ul style="list-style-type: none"> <li>- backup storage, tapes</li> </ul> </li> <li>OS and processes resident in main memory</li> </ul>	<p>Fragmentation: Static memory division • Dynamic memory division: Aufteilung des Speichers je nach Process Bedarf Problem: external fragmentation → Verdichtung Solution: Buddy alg. &amp; Paging</p> <p>Buddy alg.: Speicherverwaltung durch Aufteilung in „Buddy“-Blöcke Ablauf: Process A will 100K, Speicher wird so lange halbiert bis passender Block gefunden wird (hier 128K). B fordert 240, 256 ist noch frei. A gibt Speicher frei → Buddy Prinzip, prüft ob zusammengeführt werden kann. Der 128 Block &amp; 128 freier Block kommen zusammen = 256K B gibt Speicher frei, zwei 512K entstehen → 1M frei wieder</p>
<p>Free space Management</p>	<p>Einfache Imp., schnelles Finden von Blöcken. Nachteil: Fragmentierung, Plazierung, Free Space Management</p>
<ul style="list-style-type: none"> <li>Freie Speicherbereiche bei nächster Anforderung finden.</li> <li>Bitmap → platzsparend, schnelle Suche 1 = frei, 0 = belegt</li> <li>Linked List alg.</li> <li>Bei Speicher ausgeben ⇒ swapping</li> </ul>	<p>Swapping: Sekundärspeicher (z.B. Festplatte) wenn Prozesse pausieren/beenden → Speicher wird freigegeben bei Neustart Prozesses → Swapping zurück in den Ram</p>
<p>Ablauf Swapping:</p> <ol style="list-style-type: none"> <li>1. Prozess ausgelagert (Swap Out)</li> <li>2. Speicher frei für andere Prozesse</li> <li>3. Prozess wird wieder geladen (Swap In), bei Fortsetzung</li> </ol>	<p>Virtual Memory: nicht ganzer Prozess muss im RAM liegen. Prozesse nutzen nur kleine Teile vom Code (Spatial Localisation)</p> <p>Nicht benötigte Teile auf Festplatte ausgelagert → mehr Prozesse gleichzeitig laufen</p> <p>Lösung → Paging: Prozess → in Pages aufteilen, Hauptspeicher in Frames teilen Pages werden in Frames geladen. Verwaltung über Page Table. → MMU verwaltet Page Table</p> <p>Aufgabe MMU: Logische Add in physische Add umrechnen.</p>
<ul style="list-style-type: none"> <li>on-Demand Paging: Nur benötigte Pages werden geladen (Lazy Loading)</li> <li>Resident Set: Alle geladenen Pages eines Prozesses</li> <li>Working Set: Aktuell genutzte Pages</li> </ul>	<p>Load Control: sorgt dafür, dass nicht zu viele Prozesse gleichzeitig im Hauptspeicher aktiv sind.</p>

## Page Replacement

Problem: Prozess benötigt mehr Pages, aber nicht genug physischer Speicher verfügbar.

Lösung: Freie Frames zurückgewinnen (Page Replacement)

Alg: Optimal (Ersetzt Pages die in Zukunft am spätesten benötigt werden.)

LRU (Ersetzt die am längst nicht verwendeten)

FIFO (Ersetzt die ältesten Pages)

I/O Zugriffsarten

Port Mapped I/O (PMIO) (cat / proc / io ports)

- Eigener Add. Raum. Nicht in normal. Speicher
- Max. 64 Ports

Memory Mapped (cat / proc / iomem)

- Im normalen Speicher eingebettet

Memory Mapped Ressourcen

(sudo less / proc / iomem)

I/O Access

Programmed I/O

- CPU übernimmt all Aufg.
- Use Busy Waiting
- CPU wait, until I/O angeschlossen

Interrupt-driven I/O

- I/O informiert CPU bei Fertigstellung
- Interrupts unterbrechen Ablauf
- Asynchron (CPU)

I/O exclusive vs Shared access

Ex: Nur 1 Process, selten in Praxis

Sh: Mehrere Prozesse teilen Zugriff →

I/O Scheduling

(## iostat -p vda), (man iostat)  
(iowait, x-wait, aq-un-sz)

I/O Blocking vs non Blocking

B: Prozess wartet auf Ergebnis

N: Anfrage wird gestellt, Ergebniss kommt später

→ Prozess blockiert nicht.

Nicht verwechseln mit

Sync vs Async: Entscheidung der Prozesse

Blocking non Blocking: Eigenschaft des I/O System

Linux device Access

↓ I/O system call

↓ driver call

continual repeat

return system call

return from driver

↓ I/O commands

↓ interrupts

Input-Output - Principles (sudo fdisk -l)

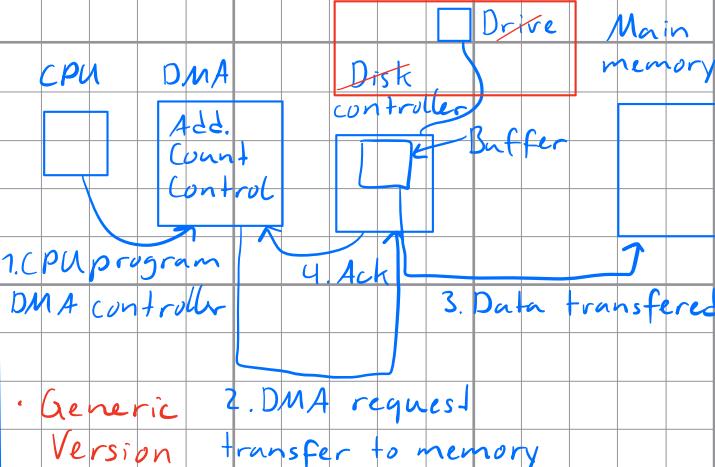
• Block-level (block of bits) ⇒ hard drive

• Character-level (streams of characters)

↳ operates on character streams: "asdfe... "

I/O Ports on Linux: sudo less / proc / io ports

I/O Gen. Architecture Network Interface



Interrupt Handling (less / proc / interrupts)

- Interrupt unterbricht CPU
- CPU führt dann Interrupt-Service-Routine (ISR)
- Nach ISR, ursprünglicher Ablauf fortsetzen

Type of Interrupts.

- synchronous (entsteht durch instruction dir by zero)
- asynchronous (entsteht durch external event)
  - ↳ I/O device, Keyboard
  - via INT CPU pin

Interrupts are

- Maskabel: can be ignored
- Non Maskabel: cannot be ignored, signaled via NMI CPU pin

Sequential vs. Random Access

S: (z.B. Netzwerkstream): FIFO

R: (z.B. HDD, RAM): Daten sind gespeichert und können direkt adressiert werden

Buffered I/O: Daten gehen über Pufferspeicher.

Direct I/O: Keine Pufferung → schneller aber weniger flex

Error Handling

smartctl -a /dev/hda1

↳ physikalische Geräte info

Alle Geräte-Nodes im Verzeichnis: ls -al /dev/

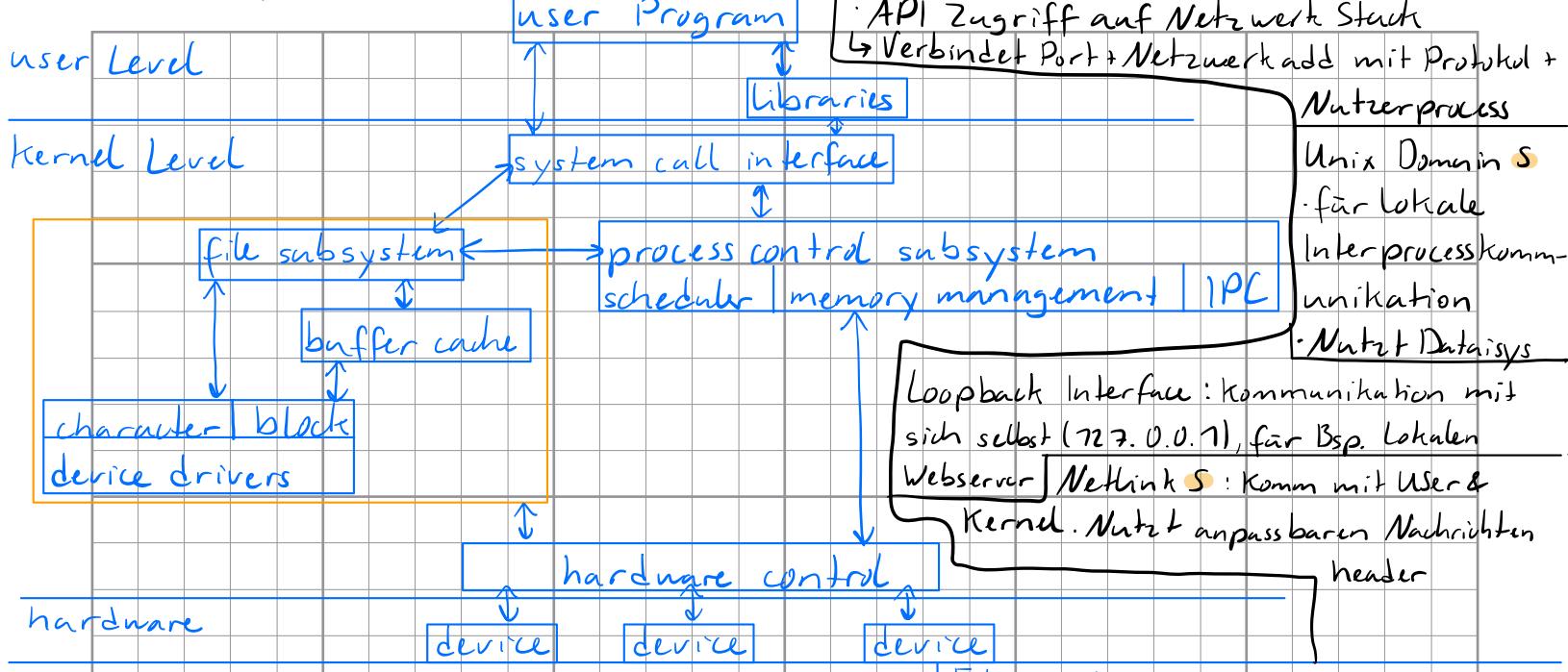
Partitionstabelle: sudo fdisk -l

Udevinfos: udevadm info --attribute-walk /dev/vda1

Gerätinfos: udevadm info --query=all --name

= dev/vda1

# Linux 10 Subsystem and Device Model



## Building & Using a Custom Kernel

- 1) Defining the configuration of Kernel
- 2) Creating a Makefile
- 3) Executing the Makefile / 4) Waiting

**lsmod:** List modules in Running K  
**insmod:** insert module in Running K  
**rmmod:** remove ""

## File - Access

- . Sequential Acc
- Zugriff in feste Reihenfolge
- Lesen & Schreiben passiert vorwärts
- Zurückspulen um alle Positionen zu erreichen.

**Random Acc**  
**Beliebig Zugriff**  
 Read/Write erfolgt direkt an beliebige stelle.

**File System**

- Store very large amounts of info.
- Support multiple storage media
- Fast find and protect data
- Read and write efficiently linear sequences of fixed-size blocks

**Directory**  
 Root / . Working .  
 Home ~ . Parent ..

**System Crash**

Help identify inconsistency in data blocks

- Used block table
- Free block table

check consistency: fsck

**Virtual File System (VFS):** Integration verschiedener

## File imp. - Contiguous Allocation

- + Easy imp.
- + Top lose - Performance
- Fragmentierung in Laufe der Zeit
- Anhängen von Dateien = schwierig

Wichtige Parameter = Blocksize

**Blatssize**  
**klein:** Viele Blöcke pro Datei  
 Geringe interne Fragmentierung  
**Langsamer Zugriff**  
**Gross:** Wenig Blöcke pro Datei  
 Höhere Datenrate  
 Größere interne Fragmentierung

(heterogener) Dateisysteme in einheitlicher Struktur

Tmpfs → Stores file in RAM

Read/Write Op, based on memcpy ⇒ fast

If system crash ⇒ Data is lost (not persistent).

## Deleting a file, einer atomaren Transaction:

1. Remove the file from its directory
2. Release the i-node to the pool of free i-nodes
3. Return all the disk blocks to the pool of free disk blocks.

## Volume management (LVM)

- . Partitioning, formatting, mounting
- . Logical Volume management
- . Flexible Speicherzuweisung
- . Kombinierung von mehreren Partitionen
- . Volumen verschiebar & vergrößbar

Init PV: pvcreate /dev/vdb /dev/vdc

PV: Reale Festplatten, die Speicher bereitstellt.

VG: Sammlung aus physischen & logischen Volumen

LV: Virtuelle Partition mit Dateisystem, unabhängig von physischer Lage

LVM Funktionen: Freie Speicherverwaltung

- . Online - Resizing
- . Snapshot
- . Balancing
- . Thin - Provisioning
- . Software - RAID

<p>Btrfs: All in one Dateisystem</p> <ul style="list-style-type: none"> <li>Gut für grosse &amp; kleine Funktionen</li> <li>COW: Änderungen erzeugen Kopien anstatt überschreiben</li> <li>Subvolume: wie Verzeichnis, aber sep. einhängbar</li> <li>Extent: logischer Bereich in Datei → physischer Speicher</li> <li>Chunks: Geräte werden gruppiert</li> <li>kleine Dateien ⇒ in 1-Node gespeichert</li> <li>große Dateien ⇒ nutzen Extents</li> <li>Rebalancing bei Fragmentierung</li> </ul>	<p>Overlay FS/AuFS - Überblick</p> <ul style="list-style-type: none"> <li>Überlagert Dateisystem transparent über anderes</li> <li>Ursprünglicher Zweck: read-only Medien Henke → Container &amp; Cloud Computing (Docker)</li> </ul>
---	---

## Networking Requirements on OS

- Kommunikation durch Stream / Datagram
- Standernetzwerke für Benutzeranwendungen
- Sicherheit gegen Angriffe
- Traffic Routing

## Network Interface Card (NIC)

- Mit Sys. Bus mit PC verbunden
  - Enthält 2 ICs:
    - PHY (Physical Layer Device - L1)
    - MAC (Media Access Controller - L2)
- Bsp. USB, Ethernet, ...

## MAC-DMA (Direct Memory Access)

- Nutzt DMA für Datenübertragung (Tx, Rx)
- Achtung: Über-/Unterlauf
- Unterstützt mehrere DMA-Jobs mit Buffer-Deskriptoren

## Layer 2 (Data-Link Layer)

- Überträgt Frames vom NIC in Hauptspeicher. Prüft Konsistenz der Frames. Unterstützt Bridging

## Zero-Copy Stack

OS-Modell: 7 Schichten

Klassisch: mehrfache Frame-Kopien

Rx: Header-Daten entfernen

Tx: " hinzufügen

Problem: Metadaten < Payload → ineffizient

Zero-Copy Idee:

- Frames liegen in Layer-2-Buffer
- Nur Pointer zu Metadaten werden übergeben

Bsp. L3 bekommt IP-Header-Pointer von L2

## L2 Buffering sk\_buff

- Metadatendaten werden geklont, Payload nicht kopiert
- Jeder NIC pflegt Ring-Buffer mit SKB-Pointer
- SG-DMA überträgt Daten direkt ins SKB

Data-Link-Aufgaben: Unterscheidet ARP & andere Frames

Prüft Ethernet Header, führt Bridging durch, leitet an next Layer weiter

## Layer: L1 - Physical Layer

### L2 - Device Drivers

#### L2 - Buffering

L2 - Data Link - Bridging

L3 - Network Layer - Routing

L4 - TCP/UDP handling

L7 - Network Sockets

PHY: Regelt Signalpegel, Takt

Stellt Verbindung zu Gegenstück her

Empfang: decodiert, entschlüsselt, deserialisiert Signal

Senden: serialisiert, verschlüsselt, codiert Daten

Liefert Bitstrom an MAC

MAC: Konfiguriert Priority über MDIO

Empfang: Prüft Integrität, speichert Daten in RAM

Meldet an NIC-Driver bei eingehendem Frame

Senden: Fügt FCS hinzu, Achtet auf Pufferunterläufe

Device Driver (NIC-Driver) Kernel-Space Treiber

Rx (Empfang): Wird bei Frame-Empfang informiert

Prüft, verwirft fehlerhafte Frames. Formatiert

Frames für Software-Stack.

Tx (Senden): Initialisiert MAC-DMA-Kanäle

Meldet zu sendende Frames an MAC. Liest Status der Übertragung

Bringt Host seitigen Buffering

L3 Network Layer: Addressierung über LAN hinaus

Prüft IP-Header-Eigenschaften, leitet Frame weiter

Firewall: Blockiert schädlichen Traffic

Routing: Leitet Pakete an Zieldomäne ausserhalb Subnetz

Netfilter-System (Linux)

Hook-System mit 5 Pkt.

NF\_DROP(0): Paket verwerfen

NF\_ACCEPT(1): Weiterverarbeiten

NF\_STOLEN(2): Funktion übernimmt Packet

NF\_QUEUE(3): Queue für User-Space

NF\_REPEAT(4): Hook erneut aufrufen

L4 Handling (Transport Layer)

UDP: Unzuverlässig, SOCK\_RAW: Zugriff auf Layer 3,4 ohne Parsing

TCP: Zuverlässig

