Notizen Betriebssysteme

Klaus Kusche

1. Kapitel: Allgemeines

Betriebssystem = Operating System ("OS")

Im *weiteren* Sinne (eigentlich *"Systemsoftware"*, nicht *"Betriebssystem"*):

Eine Sammlung von Software, die den Betrieb
sowie die Bedienung & Verwaltung eines Computers ermöglicht.

Hier: Im *engeren* Sinne:

Der Betriebssystem-Kern ("Kernel")

= der Teil der Software, der auf dem Prozessor
(zumindest auf großen, modernen Prozessoren)
mit vollen Hardware-Rechten läuft.

"Mit vollen Hardware-Rechten":

- Zugriff auf ***I/O-Devices***
  (dürfen "normale" Programme nur indirekt über das Betriebssystem)
- Zugriff auf die ***Speicherverwaltungs-Hardware*** im Prozessor
  (MMU = "Memory Management Unit")
  ==› Nur das Betriebssystem "sieht" das *gesamte RAM* des Rechners
  ==› Das Betriebssystem legt fest, *wer auf welchte Teile des Speichers zugreifen darf*
- Zugriff auf einige ***reservierte Maschinenbefehle***
  (z.B. Ein- und Ausschalten der Interrupts ("monopolisiert" den Rechner),
  Umkonfigurieren der CPU, ...)
- Bezeichnung auf x86 (Intel, AMD):
  "Das Betriebssystem läuft in Ring 0" ("Supervisor-Mode")
  Ring 1 und 2 sind normalerweise unbenutzt,
  Anwendungen laufen in Ring 3 ("User Mode")

  Interne Bezeichungen für noch höhere Rechte:
  Ring -1 = VM-Supervisor bei Hardware-basierter Virtualisierung,
  Ring -2 = System Management Mode, z.B. bei CPU Temperaturüberschreitung

  Bei Virtualisierung *ohne* Hardware-Unterstützung (heute kaum noch):
  Host-OS bzw. VM-Supervisor in Ring 0, Gast-OS in Ring 1
- Die Hardware-Rechte des gerade laufenden Codestückes haben *nichts*
  mit den Benutzer-Rechten der Programme zu tun (mit *"root"* oder *"Administrator"*):
  Die Benutzer-Rechte sind ein reines Software- bzw. Betriebssystem-Konzept,
  das nur Anwendungen betrifft: Der Kernel selbst *steht über den Benutzer-Rechten*.

Aufgabe des Betriebssystems:

Stark vereinfacht:

Das Betriebssystem ist die "Schnittstelle" zwischen Anwendungen und Hardware.

Genauer:

- Das Betriebssystem ***verwaltet die Hardware-Ressourcen*** des Rechners und teilt sie einzelnen Anwendungen bzw. Benutzern zu.

  Wichtigste Ressourcen:

  - ***CPU***: Verwaltet durch *"Scheduler"*
  - ***RAM***: Verwaltet von der MMU (= *"Memory Management Unit"* = Hardware-Einheit in jedem Core), Memory Management des Betriebssystems erstellt dafür die *"Page Tables"* (= Steuerdaten für die MMU).
  - ***Massenspeicher*** (Disk, Stick, SSD, ...): Verwaltet durch *Filesysteme*.
  - ***Netzwerke*** (hier nur am Rande betrachtet)

- Das Betriebssystem ***abstrahiert die Hardware*** und stellt ***einheitliche, hardware-unabhängige Zugriffsmechanismen*** bereit.

  Beispiele:

  - Einheitlicher Netzwerk-Zugriff via TCP/IP, egal ob Ethernet, WLAN, DSL, ...
  - Einheitlicher File- und Ordner-Zugriff, egal ob SATA-Disk, USB-Stick, PCIe-SSD, Netzwerk-Laufwerk, ...

- Das Betriebssystem ***schützt Benutzer und Anwendungen voneinander*** und ***verwaltet Benutzer und Rechte*** (bzw. ist verantwortlich für deren Umsetzung):

  - Kein Zugriff auf fremde *Daten*
  - Faire *Ressourcen-Verteilung*, keine Monopolisierung von CPU oder RAM
  - Keine systemweiten Folgen fehlerhafter oder bösartiger Programme

- Das Betriebssystem ***startet alle anderen Programme am Rechner*** und bietet ***Kommunikations- und Synchronisationsmechanismen*** zwischen Programmen (sowie zwischen mehreren parallelen Abläufen eines einzelnen Programms).

Was "sieht" man als Computer-Benutzer vom Betriebssystem?

Im Normalbetrieb: ***Nichts!!!***

(nur beim Booten und bei System-Abstürzen, d.h. "Blue Screen" bzw. "Kernel Panic")

==› Der Betriebssystem-Kern interagiert *nicht* direkt mit dem Benutzer!

Bestandteile eines Betriebssystems:

- ***Device-Treiber*** (einzige Hardware-abhängige Komponente!)
- ***Scheduler***
- ***Memory Management***, Swapping
- ***Inter-Prozess-Synchronisation und -Kommunikation***
- ***Massenspeicher***-bezogene Funktionalitäten, u.a.:
  - Device Manager (RAID, Logical Volumes, ...)

- File-Systeme
- Filesystem-Buffering, Caching, Readahead, ...
- Filesystem Quotas
- I/O-Scheduling
- Update Notification Support

- ***Netzwerk***-bezogene Funktionalitäten, u.a.:
  - Protokoll-Support (TCP/IP, ICMP, ...)
  - Prüfsummen- und Kryptografie-Funktionen (IPsec, WLAN, PPP, ...)
  - Routing, Flußsteuerung, QoS-Priorisierung, ...
  - Packet-Level Firewall
  - Ev. einfache Netzwerk-Dienste (ARP, DHCP, DNS, NTP, ...)
  - Mit Filesystemen:
    Netzwerk-Filesysteme, Cluster-Filesysteme, Netzwerk-Block-Devices
- Ev.: ***Grafik*** (OpenGL, Direct3D, ...), Video-Encode/Decode, Sound, ...

GUI und Grafik-Treiber:

Wie weit **GUI und Grafik-Treiber** Bestandteil des Betriebssystem sind,
ist von System zu System *unterschiedlich*:

Linux:
- Das ***GUI*** läuft als *normale Anwendung* rein im User-Mode:
  Hat *nichts* mit dem Betriebssystem zu tun, ist beliebig austauschbar (es gibt viele)!
- Treiber für die ***Grafikkarten-Grundfunktionen*** (Auflösung einstellen, ...): Im *Kernel*
- 3D-Grafik- und Video-***Beschleunigungs-Code*** läuft *großteils im User-Mode* als Libraries

Windows:
- Teile des sichtbaren ***GUI*** gehören ***fix zum Betriebssystem***, keine Trennung möglich
- ***Grafikkarten-Treiber*** und ***3D-Beschleunigungs-Code*** gehören großteils zum *Kernel*

Probleme:

- GPU und VRAM bieten meist noch nicht alle HW-Möglichkeiten wie eine CPU, die zur sauberen Trennung und Verwaltung von mehreren Benutzern bzw. Anwendungen und zur Rechte- und Ressourcenverwaltung nötig sind
- Möglichst direkter Zugriff der Anwendungen auf die GPU-Hardware ist aus Performence-Gründen wünschenswert

==> Sichere, *gemeinsame Nutzung* durch *mehrere Programme*
oder sogar *mehrere Benutzer* gleichzeitig ist *technisch schwierig* zu lösen
==> "ewige Baustelle"

Betrachten wir nicht weiter...

Struktur der Software auf einem Rechner (nur ausführbarer Code):

- ***Anwendungen***, Commandline-Befehle, Dienste, Server, ...
  (egal, ob mit dem Betriebssystem mitgeliefert oder als separates Paket installiert)

  Ist technisch alles dasselbe: *"Ausführbares Programm"* (.exe-File)
  Unterscheiden sich nur, ob sie mit GUI, mit Befehlszeilen-I/O
  oder ganz im Hintergrund (ohne Benutzer-Interface) laufen.

  Auch der ***Commandline-Interpreter*** ist technisch eine *normale Anwendung*

(Windows: `cmd.exe` = *"DOS-Fenster"* bzw. *PowerShell*; Linux: *Shell*)
wenn auch eine sehr zentrale.

- (Shared) ***Libraries***
  Windows: `.dll` (*"dynamically loaded library"*)
  Linux: `.so` (*"shared object"*)
  Code, der *von mehreren Anwendungen gemeinsam benutzt* wird
  Laufen auch *im User-Mode* (==› Nicht Bestandteil des Kernels)
  Aber: *Enthalten oft System Calls*
  (d.h. machen das Betriebssystem für Anwendungen nutzbar, siehe unten)
- Betriebssystem-***Kernel***

Wann wird der Betriebssystem-Kernel-Code aktiv?

- ***Beim Booten*** gleich nach BIOS und Bootloader
  (BIOS gehört zur HW, nicht zum OS)
  (Bootloader wird vom OS installiert, ist aber ein eigenständiges Stück Code,
  das nur für diesen einen Schritt beim Booten *vor* dem OS läuft)
- Bei einem ***System Call***
- Bei einem ***Interrupt***

System Call:

= Aufruf einer Betriebssystem-Funktion durch eine Anwendung
(z.B. File-I/O, Hauptspeicher-Anforderung, Starten eines anderen Programms, …)

Gedanklich wie ein *Funktionsaufruf*:
Bekommt *Parameter-Werte* übergeben, liefert *Returnwert*,
Aufrufer macht danach normal weiter

Technisch kein CALL-Maschinenbefehl,
sondern *eigener Befehl* "SYSCALL" oder "SWI" (Software-Interrupt)
(mit entsprechendem SYSRET- oder IRET-Befehl,
der alles wieder zurückschaltet und in die Anwendung zurückkehrt)

Macht viel mehr als ein Funktionsaufruf, ähnlich Interrupt:

- Schaltet den Ausführungs-Modus auf ***Supervisor-Mode*** (Ring 0) um.
- Schaltet den ***MMU-Kontext auf Kernel-Kontext*** um (==› Kernel-RAM sichtbar).
- Sichert den ***kompletten Register-Zustand*** der Anwendung
  (weil es könnte während eines System-Calls eine andere Anwendung laufen,
  z.B. beim Warten auf I/O, d.h. Benutzer-Input, Platte, Netz, …).
- Führt den zum System Call gehörenden Kernel-Code aus.

==› *Viel langsamer als ein Funktionsaufruf, einige 100/1000 Takte statt ‹10 Takte!*

Zumindest unter Linux:
Nicht ein Syscall-Einstiegspunkt in den Kernel pro Kernel-Funktion,
sondern *ein einziger Syscall-Einstieg für alle Aufrufe* aus Anwendungen:
Nummer der auszuführenden Kernel-Funktion wird *als zusätzlicher Parameter* übergeben.

Windows: Syscalls sind *nicht offiziell dokumentiert*, in *Windows-Libraries versteckt*,
sollten von Anwendungen nicht direkt aufgerufen werden!

Stabile, *offizielle Schnittstelle* für Anwendungen zum Betriebssystem liegt *eine Ebene höher*:
Dokumentierte Aufrufe der mit Windows mitgelieferten Libraries.

Linux: Syscalls sind *wohldokumentierte, über Jahrzehnte stabile Schnittstelle* zum Kernel
(Dokumentation: `man` Sektion 2, einige hundert Funktionen)
Für jeden Code (auch Anwendungen) *direkt aufrufbar*:
Via Assembler-Code auch ohne Library-Aufruf, bzw. universelle Syscall-Library-Funktion!
(in der Praxis: Aufrufe mehrheitlich über C-Library usw., nicht direkt aus Anwendungen)

Interrupt:

= "Ungeplante" Umschaltung von der gerade laufenden Anwendung in das Betriebssystem.

Gründe bzw. **Interrupt-Quellen**:

- ***Hardware-Bausteine***: *"Liebe CPU, ich brauch was von dir / ich hab was für dich"*
  - Fast alle *I/O-Devices* ("bin mit dem I/O fertig", "neue Daten eingetroffen", ...)
  - *Timer* (regelmäßiger Systemtakt, Wartezeit abgelaufen, ...)
  - *Fatale HW-Fehler* (RAM Parity Error, Stromausfall, System zu heiß, ...)
- Die ***CPU*** selbst:
  - *Division durch 0*
  - Verletzung der *RAM-Zugriffsrechte* (MMU)
  - *Undefinierter oder unerlaubter Befehlscode*, undefinierter Speicherzugriff, ...
- ***Spezielle Maschinenbefehle*** im ausgeführten Anwendungs-Code:
  - *System Call* (wenn kein eigener SYSCALL-Befehl)
  - *Debugger-Breakpoint*

Jeder Interupt-Quelle ist eine ***fixe Nummer*** zugeordnet.
An einer fixen RAM-Adresse (meist 0) steht eine nur für den Kernel sichtbare Tabelle
(die *"Interrupt-Vektor-Tabelle"*), die jeder Nummer eine *Kernel-Code-Adresse* zuordnet,
nämlich die Adresse des jeweiligen ***"Interrupt Handlers"***, d.h. des Kernel-Codestückes,
das *für die "Bedienung" des Interrupts mit dieser Nummer zuständig* ist.

Was passiert? Ähnlich wie bei einem System Call: Die CPU (in Hardware, ohne SW)

- ***Unterbricht*** die gerade laufende Anwendung
  am Ende des gerade ausgeführten Maschinenbefehls (egal welcher).
- Schaltet den Ausführungs-Modus auf ***Supervisor-Mode*** (Ring 0) um,
  schaltet den MMU-Kontext auf ***Kernel-Kontext*** um (==> Kernel-RAM sichtbar).
- Sichert den ***Register-Zustand*** der Anwendung (zum nachher Weitermachen).
- Springt dem zu diesem Interrupt gehörenden ***Interrupt-Handler*** im Kernel,
  indem sie aus der "Interrupt-Vektor-Tabelle" die zu dieser Interrupt-Nummer
  gehörende Adresse lädt.

Wenn Interrupt-Handler **fertig** gerechnet hat
(endet meist mit speziellem Maschinenbefehl "Interupt return"):
Alles zurück, *Anwendung macht weiter, als ob nichts gewesen ist*
(außer bei "Schutzverletzung", Division durch 0, usw.).

Für die Anwendung *völlig transparent* bzw. nicht feststellbar,
bis auf die *"gestohlene" Zeit*.

Andere Interaktionen des Kernels mit Anwendungen:

- ***Signale:*** Kommunikation vom Kernel zu einer Anwendung
  Unterbrechen eine gerade laufende Anwendung (egal wo)
  und setzen ihre Ausführung mit einer *vorher festgelegten Funktion*
  in der Anwendung fort (*"Signal Handler"*).
  Konzept ähnlich Interrupts, aber auf Anwendungs-Ebene.

  Beispiele:
  - Bestimmte ***Anwendungs-Fehler*** (siehe CPU-Interrupts,
    werden vom OS teilweise als Signal an die Anwendung weitergereicht)
  - Bestimmte ***I/O-Probleme***
    ("Übergeordnetes Terminal beendet", "anderes Pipe-Ende beendet", ...)
  - ***Ctrl/C***, von Benutzer oder anderer Anwendung geschicktes ***"Beenden"***
  - ***Limit*** überschritten (z.B. max. CPU-Laufzeit, ...)
  - Anwendungs-***Timer*** abgelaufen
  - ***Sohn-Prozess*** hat geendet

Andere Schnittstellen des Kernels nach außen:
Bereitstellung von Informationen über Hardware, Systemzustand, Kernel, ...:

- ***Log-Mechanismus*** (zentrales Protokoll aller System-Meldungen)
- Unter Linux: ***Pseudo-Filesysteme /proc, /sys, ...***
  Sehen aus wie File-Systeme
  (enthalten Directories und lesbare, zum Teil auch schreibbare Files)
  Aber stehen auf keiner Platte o.ä.,
  sondern werden vom Kernel "simuliert" und mit Informationen gespeist.

"Monolithisches" Betriebssystem versus "Microkernel":

- ***"Monolithisches" Betriebssystem:***

  Der gesamte Kernel-Code läuft als ***ein einziges, großes Stück Software***
  (oder ein mittelgroßes Stück Software, das bei Bedarf Module mit Device-Treibern usw. nachlädt), d.h. ***alle Teile laufen mit vollen Supervisor-Mode-Rechten***, sehen das gesamte RAM und alle I/O-Devices, und können einander beliebig aufrufen bzw. beliebig gegenseitig auf Datenstrukturen oder gemeinsame Daten zugreifen.

- ***"Microkernel"-System:***

  Der eigentliche Kernel wird möglichst klein gehalten. Er umfasst nur
  - Den Scheduler
  - Die Speicherverwaltung
  - Mechanismen zur Synchronisation und zum Datenaustausch zwischen Prozessen

  Alles andere (Device-Treiber, Filesysteme, Netzwerk-Protokolle, ...)
  wird einzeln in jeweils separate Prozesse ausgelagert, die
  - nur auf ihre eigenen HW-Devices zugreifen können,
  - nur ihren eigenen, privaten Speicherbereich sehen,
  - mit reduzierten Rechten laufen und
  - mit dem Rest des Systems nur über vordefinierte Mechanismen kommunizieren.

"Sonderfälle" von Betriebssystemen:

- ***"Echtzeit-Betriebssystem":***

Primär: *Garantierte maximale Reaktionszeit auf Interrupts*
Ev. auch: *Garantierte maximale Laufzeiten für bestimmte Kernel-Funktionen*
(z.B. Speicher-Anforderung, Prozess-Wechsel, Prozess-Synchronisation, ...)

==› System reagiert innerhalb eines Zeitlimits auf externe Ereignisse
==› System verarbeitet bestimmte Daten innerhalb eines Zeitlimits
Zeit-Limits je nach Anwendung zwischen einigen µsec und etwas unter 1 sec.

"Harte" Echtzeit: Maximales Zeitlimit *darf nie überschritten werden*,
hätte *gefährliche Folgen* für Mensch und/oder Maschine
(z.B. Maschinensteuerungen, KFZ-Systeme wie ABS, ...).

"Weiche" Echtzeit: Überschreitung des Zeitlimits ist *störend, aber nicht gefährlich*
==› sollte *möglichst selten überschritten* werden
(z.B. Audio / Video-Systeme, Telekom)

- ***"Verteiltes" Betriebssystem:***

  Mehrere vernetzte Rechner verhalten sich zum Anwender hin
  wie ein einziges System.

- ***VM-Hypervisor:*** Dient nur der Ausführung und Überwachung von VM's

Liste von bedeutenden Betriebssystemen (Auswahl):

- **Microsoft:**
  - DOS (plus diverse DOS-Nachbauten)
  - Windows bis Windows 3.11 (16 Bit)
  - Windows 95 / 98 / ME (aufgeblasen auf 32 Bit)
  - Windows ab Windows NT (komplett neu) (plus ReactOS, ein Windows-Nachbau)
  - Windows CE (plus erste Windows-Phone-Versionen) (auch als Embedded-OS)
- **Die Unix-Familie:**
  - Historisch:
    - AT & T System V Unix (Bell Labs, Unix-Erfinder)
    - BSD (University of California at Berkeley)
    - Mach, NextStep, Hurd (neu entwickelte Microkernel mit Unix-Funktionalität)
    - ... und Dutzende kommerzielle Derivate von diesen drei Systemen
    - Minix (Prof. Tanenbaum, akademisches Lehr-Unix, Vorbild für Linus Torvalds)
  - Modern:
    - Linux (Open Source, "from Scratch"-Nachbau), auch Android
    - Einige BSD-Abkömmlinge (auch Open Source)
    - MacOS und iOS (Kernel: NextStep; Userland: BSD)
    - Kommerzielle Unix-Derivate (AIX, HPUX, Solaris + OpenSolaris, ...)
- **Echtzeit-Systeme:**
  - VxWorks
  - QNX (auch Blackberry)
  - OS/9
  - PikeOS (deutsch, Fa. Sysgo)
  - ... (viele)
  - Diverse Echtzeit-Linux-Varianten
  - Diverse Dual-Kernel-Linux-Varianten (z.B. RTAI)

- **IBM Großrechner:**
  z/OS und z/VM
- **DEC** (dann Compaq, dann HP, …):
  VMS (u.a. "unfreiwilliger" Lieferant der Kernel-Technologie für Windows NT)
- **Spezialsysteme:**
  Cisco IOS (für Netzwerk-Geräte)
- **"Ewige Baustelle":**
  Plan 9 (AT & T Forschungsprojekt nach Unix)

Standards:

Nur im Unix-Bereich:

Historisch: ***X/Open*** (Industrie-Konsortium der ~20 wichtigsten Unix-Anbieter)

Heute: ***POSIX*** ("Portable Operating System Interface"),
von der *IEEE* und der *"Open Group"* (Industrie-Konsortium, "Unix"-Owner) standardisiert:
- *C-Library* (Header) incl. System Calls
- *Shell* und *Commandline-Befehle*
- Erweiterungen für *Inter-Prozess-Kommunikation* und *Parallelität*, Echtzeit, …

Auch *unter Windows emuliert* (*"Microsoft Services for Unix"* bzw. *"Subsystem for Unix"*)
Auch *zahlreiche Nicht-Unix-Systeme* (VxWorks, QNX, …) sind kompatibel
(*"it walks like Unix, it talks like Unix …"*)

2. Kapitel: Prozesse, Threads, Scheduling

Prozesse, Tasks, Threads

Oft durcheinander gebracht, auch in Fachliteratur uneinheitlich verwendet.

Task:

Nicht klar bzw. einheitlich definiert.

Prozess:

Prozess = in Ausführung befindliches Programm
(Code + Daten)

Egal ob Anwendung, Dienst, ...:

Alles, was außerhalb des Kernels läuft, läuft in einem Prozess:

- Ein Programm wird erst mit dem Starten ein Prozess:
  Ein **.exe**-File an sich ist noch kein Prozeß.
- Dasselbe Programm kann zu einem Zeitpunkt
  von mehreren, voneinander unabhängigen Prozessen ausgeführt werden.

Aus Betriebssystem-Sicht:

Prozess = Einheit der Rechte-Verwaltung, Speicherverwaltung usw.

Ein Prozess
- hat während seiner Laufzeit eine *eindeutige Kennung* (Nummer): "Pid" = Process Id
- hat *zugeordnetes RAM* (Code, statische/globale Daten, Stack, Heap)
- hat einen zugeordneten **.exe**-*File* und eingeblendete *Shared Libraries*
- hat *offene Files* (und Schreib-Lese-Positionen usw. in jedem File),
  zugeordnete *Netzwerk-Verbindungen* und I/O-Devices, ...
- läuft mit bestimmten *Rechten* (z.B. bezüglich File-Zugriffen)
  d.h. im Namen eines bestimmten *Benutzers*/Eigentümers (und ev. einer Gruppe usw.)
- hat ev. zugeordnete *Inter-Prozess-Kommunikations-Ressourcen*
  (Shared Memory, Semaphore, ...)
- hat ev. ein *"Controlling Terminal"* (z.B. für Ctrl/C)
  bzw. gehört zu einer *"Session"* eines Benutzers
- hat ein *"Environment"*,
  zu dem u.a. die *Environment-Variablen* und das *aktuelle Verzeichnis* gehören
- hat (zumindest unter Unix/Linux) einen *Vater-Prozess* ("Parent Pid"),
  der ihn gestartet hat und *von seinem Ende informiert* wird
- hat zugeordnete *Ressourcen-Limits* (max. reales und virtuelles RAM,
  max. Anzahl offener Files, max. CPU-Verbrauch, max. Anzahl Threads, ...)

Thread:

Thread = Ein Ausführungs-Ablauf

Aus Betriebssystem-Sicht:

Thread = Einheit der CPU- bzw. Rechenzeit-Zuteilung

Ein Prozess enthält daher immer *mindestens einen Thread*,

eventuell auch *mehrere* / viele (*paralleles Programm*).

Jeder Thread gehört zu einem Prozess bzw. läuft *innerhalb eines Prozesses*
(d.h. hat Zugriff auf dessen RAM und dessen offene Files, ...;
"erbt" die Rechte und das Environment seines Prozesses, ...).
Ausnahme: In vielen Systemen gibt es auch *Betriebssystem-Threads* (gehören zum *Kernel*).

Von den Threads eines Prozesses können zu jedem Zeitpunkt keiner, einer oder
(bei Multicore-Systemen) mehrere gleichzeitig ausgeführt werden (d.h. wirklich rechnen).

Threads können während der Ausführung eines Programms beliebig gestartet werden
und enden (endet der letzte Thread, endet auch der Prozess).

Ein Thread umfasst im Wesentlichen nur

- einen eigenen *Program Counter*,
  d.h die *aktuelle Ausführungs-Stelle im Programmcode*
  (jeder Thread befindet sich zu jedem Zeitpunkt an genau einer
  aktuellen Codestelle im Programmcode seines Prozesses)
- und einen *Stack Pointer*, der angibt, wo im RAM des Prozesses dieser Thread
  seinen Stack (für lokale Variablen und Funktionsaufrufe) gespeichert hat.

Ob CPU-Zuteilungs-Prioritäten vom Betriebssystem pro Prozess verwaltet werden
(d.h. alle Threads des Prozesses haben gleiche Priorität) oder pro Thread,
ist von System zu System unterschiedlich.

Das *Starten eines neuen Prozesses* und ein *Prozess-Wechsel*
(umschalten der CPU auf einen anderen Prozess) ist relativ *langsam und aufwändig*
(vor allem wegen dem Neu-Anlegen oder *Umschalten der RAM-Zuordnung* und der *Rechte*,
erfolgt immer unter Beteiligung des Betriebssystems), das *Starten eines neuen Threads*
und ein *Thread-Wechsel* zwischen Threads desselben Prozesses ist *viel schneller*
(je nach Implementierung ist ein Thread-Wechsel ev. sogar ohne Betriebssystem möglich,
da *keine Umschaltung des RAM-Kontexts* und keine Änderung von Rechten usw. nötig ist).

Daher alter Begriff für Threads: ***"Lightweight process"***

Daher Fachausdruck für Umschalten zwischen zwei Prozessen: ***"Context switch"***

Prozess-Zustände (eigentlich: Thread-Zustände)

Auf jedem Core kann zu jedem Zeitpunkt
*nur **ein einziger Thread** ausgeführt werden!*

(ein *Hyperthreading-Core* wird aus Betriebssystem-Sicht
als *zwei getrennte Cores* betrachtet)

Daher:
Vereinfacht befindet sich ein Thread zu jedem Zeitpunkt in einem von drei Zuständen:

- ***Laufend:*** Der Thread hat einen Core zugewiesen bekommen und *rechnet*.
- ***Blockiert:*** Der Thread rechnet gerade nicht und könnte auch nicht rechnen
  (d.h. will momentan gar keine CPU), weil er entweder darauf wartet,
  dass eine *I/O-Operation* fertig wird, oder dass ein *Timer* abläuft, oder
  dass z.B. ein anderer Thread *gemeinsame Ressourcen freigibt* oder *Daten schickt*
  (Inter-Prozess-Kommunikation).

- ***Wartend = lauffähig:*** Der Thread *würde gerne weiterrechnen*, aber es ist gerade *kein Core frei*.

Übergänge:

- Ein neu gestarteter Thread wird wartend.
- Ein wartender Thread wird laufend, sobald ein Core frei wird und ihm das Betriebssystem diesen Core zuteilt.
- Ein laufender Thread kann einen Systemaufruf machen, der ihn blockiert (I/O, Timer, Warten auf einen anderen Thread, …).
- Fällt der Blockierungs-Grund weg (I/O fertig, Timer abgelaufen, …), wechselt der Thread von blockiert nach wartend.
- Das Betriebssystem kann einen laufenden Thread unterbrechen und ihm den Core wegnehmen.
  In diesem Fall wechselt der Thread direkt von laufend auf wartend (er möchte ja gleich wieder weiterlaufen).
- Ein laufender Thread kann von sich aus komplett enden (bei vielen Systemen bleibt er dann in einem vierten Zustand "beendet", bis alles aufgeräumt ist und jemand den Ende-Status abgefragt hat) (von außen kann ein Thread in jedem Zustand beendet werden).

Der Scheduler

… ist ein Bestandteil des Kernel

… verwaltet die Cores der CPU, d.h. teilt den Threads Rechenzeit zu

Er kümmert sich um zwei zentrale Fragen:

1.) Wenn ein Core frei wird:

   Welcher Thread kommt als Nächster dran,
   d.h. wer bekommt den freien Core?

2.) Für die gerade ausgeführten Threads:

   Wie lange dürfen sie rechnen,
   d.h. wann wird wem sein Core wieder *weggenommen*?

Welche **Anforderungen** werden an einen Scheduler gestellt?

- ***Effizienz, Systemdurchsatz:***
  - *Bestmögliche Nutzung* und Auslastung von CPU und I/O
  - *Geringer interner Overhead* des Schedulers, auch bei Verwaltung tausender Threads
  - *Nicht mehr Prozesswechsel als nötig* (verursacht Overhead, Cache-Ineffizienz, …)
- Bestmögliche ***Reaktionszeit:***
  - *Echtzeit-kritische Threads* müssen "sofort" drankommen
  - Der *interaktive Reaktions-Eindruck* (GUI, Desktop, Multimedia-Programme)

soll nicht unter CPU-intensiven Hintergrund-Programmen leiden
(Compile-Jobs, Mining, Video-Umcodierung, …)

- Die "Erwartungs-Psychologie":
  - Was intuitiv wenig CPU braucht, soll "sofort" reagieren
  - Was rechenintensiv ist, darf verzögert reagieren

- ***Fairness:***

  "Gleichwertige" Programme (bzw. bei Multiuser-Maschinen:
  Die Programme der einzelnen Benutzer) sollen *gleichschnell reagieren*
  und *gleichviel CPU bekommen* (oder bei hoher Last gleichwenig):

  Keiner darf den Rechner monopolisieren, keiner darf "verhungern".

Wann wird der Scheduler aktiv?

- Jedesmal, wenn ein *Thread laufbereit* wird:
  - I/O-Operation fertig
  - Neues Programm gestartet
  - Schlafender (blockierter) Thread z.B. durch Ablauf eines Timers oder durch Interprozess-Kommunikation aufgeweckt

  ==› Wo wird dieser Thread unter den Wartenden eingereiht,
  bekommt er ev. *sofort* einen Core zugeteilt, d.h. wird *ein anderer Thread verdrängt*?

- Jedesmal, wenn ein Thread die CPU abgibt (blockiert oder fertig wird)

  ==› Wer bekommt den frei gewordenen Core?

- *Periodisch* bei jedem Interrupt des *System-Timers* (typisch 50 Hz - 1000 Hz)

  ==› Threads mit abgelaufener Zeitscheibe durch andere Threads ersetzen
  (Fairness: *"Lass jetzt mal jemand anderen ran"*)

Umgekehrt gilt daher (zumindest auf Single-Core-Systemen):

Solange *alle Interrupts gesperrt* sind,
kann auch der Scheduler *nicht von sich aus aktiv* werden,
d.h. kann auch dem gerade laufenden Thread
nicht "unfreiwillig" die CPU weggenommen werden.

Zusätzlich zu den "echten" Prozessen gibt es normalerweise einen ***"Idle-Prozess"***
(mit so vielen Threads wie Cores), der drankommt, wenn es keinen laufbereiten Thread gibt, und der nichts tut (bzw. den Core in einen Sleep-Zustand schickt).

Grundsätzliche Arten von Scheduling:

- ***Reiner Batch- bzw. FIFO-Betrieb*** ("first come, first served"):

  Ein Job nach dem anderen wird "in einem durch" abgearbeitet:
  Er gibt die CPU nur ab, solange er auf I/O wartet, oder wenn er fertig ist
  (kann und darf also auch minutenlang die CPU monopolisieren,
  wenn er kein I/O macht).

- ***"Kooperatives" Multitasking:***

  Dem gerade laufenden Thread kann die CPU *nicht zwangsweise weggenommen*
  werden, sondern nur
  - wenn er I/O bzw. einen System Call macht

- oder wenn er dem Betriebssystem mitteilt, dass er jetzt die CPU *freiwillig abgibt* (System Call *"Yield"*)

==> *Fehlerhafte oder unkooperative Programme (die kein "yield" machen) können das System monopolisieren und andere blockieren!*

Gibt ein Thread die CPU ab, bekommt sie normalerweise der am *längsten* auf die CPU *wartende* Thread (also auch FIFO in der Reihenfolge der Ready-Queue).

Beispiele: Windows 3.1x

- ***"Präemptives" Multitasking, Round-Robin:***

  Präemptiv = das Betriebssystem kann einen Thread jederzeit und an jeder Stelle im Programm unterbrechen und ihm die CPU wegnehmen

  Jeder Thread darf maximal eine ***"Zeitscheibe"*** (d.h. ein *Intervall zwischen zwei periodischen Interrupts des System-Timers*, also einige Millisekunden) lang durchgehend rechnen, dann unterbricht ihn das Betriebssystem und nimmt ihm die CPU weg, er muss sich wieder *"ganz hinten anstellen"*.

  Effekt: ***Faire Verteilung***
  Alle lauffähigen Threads bekommen reihum immer wieder eine "Zeitscheibe" lang die CPU, d.h. jeder bekommt gleichviel und muss gleichlang warten.
  Je mehr lauffähige Threads es gibt, umso länger wird die Wartezeit, und umso weniger CPU-Anteil bekommt jeder.

  Klassischer Ansatz für *Desktop- und Multiuser-Systeme*.

- ***"Präemptives" Multitasking, rein prioritätsgesteuert:***

  Sobald ein Thread mit *höherer Priorität* als der gerade laufende laufbereit wird, wird dem gerade laufenden Thread *sofort die CPU weggenommen*, und der höherpriore Thread darf rechnen, solange er kann bzw. will.

  Gibt der gerade laufende Thread die CPU ab, bekommt sie der laufbereite Thread mit der höchsten Priorität.

  Umgekehrt formuliert:

  Ein Thread bekommt nur dann und nur solange CPU, wie es keinen laufbereiten Thread mit höherer Priorität gibt.

  ==> Threads mit geringer Priorität können *"verhungern"*!

  Ein höchstpriorer Thread bekommt jederzeit sofort beliebig viel CPU, unabhängig von der Systemlast.

  Klassischer Ansatz für *Echtzeit-Systeme*.

Reale Scheduling-Strategien:

"Round robin" und "rein prioritätsgesteuert" sind die beiden extremen Ansätze, die Praxis liegt irgendwo dazwischen:

- ***Gewichtetes Round-Robin:***
  Wichtige Threads bekommen größere Zeitscheiben,
  Threads mit geringer Priorität kleinere.

Oder: Unwichtige Threads bekommen nur bei jeder x-ten Runde eine Zeitscheibe.

- ***Beides:***
  Unterscheidung zwischen Echtzeit-Prozessen und Nicht-Echtzeit-Prozessen,
  zwei getrennte Scheduler:

  Was die rein prioritätsgesteuerten Echtzeit-Prozesse an CPU übrig lassen,
  teilen sich die Nicht-Echtzeit-Prozesse per Round-Robin fair und reihum auf.

  Z.B. Linux:
  Jeder Thread hat *entweder* eine *Echtzeit-Priorität* 1 … 99 (nur mit "root"-Rechten),
  oder eine *"normale" Priorität* (*"nice value"*) -20 … +19:
  0 … "normales" GUI-Programm
  >0 … "unwichtiges" Programm, z.B. Compile-Jobs, Dienste, …
  (bekommen bei Hochlast proportional weniger CPU-Anteil)
  <0 … "dringendes" Programm, z.B. Media-Player oder Spiel

Das I/O-Problem:

Das Scheduling von *reinen Rechen-Threads* (ohne I/O)
ist theoretisch und praktisch "einfach".

Aber *mit I/O* funktionieren die "einfachen" Ideen nicht mehr:

Würde man einen Thread, der *nach einem I/O* wieder laufbereit wird,
jedesmal *ganz hinten* in die Warte-Queue einreihen, wäre das in der Praxis untauglich:

- Er müsste nach dem I/O noch warten,
  bis alle anderen Threads ihre Zeitscheibe bekommen haben
  ==> Das *"Interaktive Feeling"* bei GUI-Programmen wäre *katastrophal*
- Er käme bestenfalls ein paar Mal pro Sekunde dran,
  könnte daher *nur ein paar I/O's pro Sekunde* machen
  ==> Kopieren, Runterladen usw. liefe nur mit ein paar KB/sec Durchsatz
- Solche Threads rechnen im Normalfall zwischen zwei I/O's nur ganz kurz
  ==> In Summe bekäme er *viel weniger CPU-Zeit* als die anderen
- Aber: Würde man einen Thread nach jedem I/O *vorne* in die Queue stellen,
  dann würde ein Threads mit viel I/O und auch viel CPU-Verbrauch dazwischen
  viel mehr CPU als alle anderen bekommen!

Daher sind stark vereinfacht folgende Fragen zu lösen:

- Wird ein Thread, der I/O gemacht hat und wieder laufbereit wird,
  hinter oder vor andere bereite Prozesse gereiht?
- Bekommt er nach jedem I/O jedesmal wieder eine volle *Zeitscheibe*,
  oder nur das, was von der Zeitscheibe vor dem I/O noch übrig ist?

In der Realität wurden statt "Round Robin" verschiedenste Ansätze implementiert,
nach denen die wartenden Threads geordnet und der nächste Thread ausgewählt wird:

- *Deadline-basiert:* Jedem Thread wird eine Deadline zugewiesen,
  bis zu der er das nächste Mal aktiviert werden muss.
  Der Thread mit der knappsten (zeitlich als nächstes ablaufenden)
  Deadline kommt als nächstes dran.
- Der Thread, der *am längsten warten* musste, kommt als nächstes dran.

- Der Thread, der von dem ihm zustehenden CPU-Anteil bisher *am wenigsten verbraucht* hat, kommt als nächstes dran.
- Ähnlich, Grundlage vieler "fairer" Scheduler:
  Der Thread mit dem *höchsten Verhältnis Wartezeit/Laufzeit* kommt als nächstes dran.
- ***"Multilevel feedback queue"*** oder Ähnliches:
  War für viele Jahre die *Standard-Scheduler-Strategie* von Windows (immer noch), Linux (nicht mehr) und vielen Unix-Varianten:
  - *Mehrere Round-Robin-Queues* verschiedener Priorität, zwischen denen die Rechenzeit verteilt wird.
  - *Laufende dynamische Anpassung der Thread-Priorität*
    je nach Verhalten des Threads:
    Thread macht viel interaktives I/O
    ==› Thread bekommt höhere Priorität, d.h. kommt in eine bessere Queue
    Thread verbraucht viel CPU ohne I/O
    ==› Priorität sinkt, d.h. Thread kommt in eine schlechtere Queue

Scheduler-Spezialthemen:

- Automatische (heuristische) ***Prozesstyp***-Erkennung, unterschiedliches Scheduling für verschiedene Prozesstypen:

  - *Interaktive Prozesse* (GUI-Programme)
  - *Echtzeit-Prozesse*
  - *Hintergrund-Dienste*
    (dauernd vorhanden, aber selten aktiv, ohne Benutzer-Interaktion)
  - *Batch-Jobs*
    (laufen einmal gestartet mit möglichst viel CPU bzw. I/O und ohne Benutzer-Interaktion durch, bis sie fertig sind)

  Bzw. grundlegender: Erkennung und unterschiedliche Behandlung von
  - "CPU-bound" Prozessen
  - "I/O-bound" Prozessen

- Scheduling unter Berücksichtigung von ***Thread- und Prozessgruppen***:

  - Wenn jeder Thread denselben fairen Anteil bekommt, bekommt ein Multithread-Programm insgesamt deutlich mehr CPU als ein sequentielles Programm.

    ==› *Faire Aufteilung pro Programm*, nicht pro Thread, d.h. alle Threads eines Programms müssen sich einen Anteil teilen

  - Wenn ein Benutzer viele Programme gleichzeitig laufen lässt, bekommt er in Summe einen größeren CPU-Anteil als ein Benutzer, der nur mit einem Programm arbeitet

    ==› *Faire Aufteilung pro Benutzer-Session*, dann Unterteilung auf die Prozesse einer Session.

- ***Core-Affinität***:

  Threads bevorzugt auf den Core legen, auf dem sie zuletzt gelaufen sind.

  Grund: In diesen Caches dieser Cores liegen noch die Daten dieser Threads

==› Threads laufen auf diesen Cores etwas schneller.
Nachteil: Weniger gleichmäßige Auslastung aller Cores.

- ***NUMA-Berücksichtigung***:

  Auf einem NUMA-System ("Non uniform memory access")
  ist nicht jeder RAM-Bereich für jeden Core gleich schnell ansprechbar.
  Daher: Threads bevorzugt auf jene Cores legen,
  in deren RAM die Daten der Threads liegen ==› Thread läuft dort schneller.

- Berücksichtigung von ***Hyperthreading-Cores***:
  Logisch als *zwei getrennte Cores* verwaltet.

  Aber: Threads müssen sich die *Rechenzeit teilen*,
  bekommen also de facto weniger CPU-Zeit als bei "alleiniger" Ausführung
  auf zwei verschiedenen Cores (bei Fairness-Berechung berücksichtigen).

  Dafür: Beide logischen Cores haben Zugriff auf *denselben Cache*,
  daher kein Effizienzverlust beim Hin- und Herschieben eines Threads.

- Kein starrer ***Takt des System-Timer-Interrupts***:

  Einige Systeme verwenden keinen fixen Systemtakt mehr, sondern der Scheduler
  programmiert bei jedem Aufruf das nächste System-Timer-Intervall so,
  wie er es braucht (d.h. wann er wieder etwas tun muss).
  Wenn kein Programm läuft oder nicht alle Cores belegt sind ==› Längere Ticks

  Vorteil:
  - *Höhere Effizienz*, weniger Betriebssystem-Overhead
  - CPU kann sich länger schlafen legen, wird seltener aufgeweckt
    ==› Vor allem bei mobilen Devices deutlich ***geringerer Stromverbrauch***!

- Der Scheduler als ***Überlast-Wächter***:

  - *Batch-Jobs* (Zeit-unkritische Hintergrund-Jobs)
    bei hoher interaktiver Systemlast zur Entlastung *gar keine CPU mehr zuteilen.*

  - *Speicherintensive Prozesse*, die sehr viel Paging verursachen
    und das System dadurch einbremsen, einbremsen bzw. selten aktivieren,
    um den Paging-Druck zu senken.

- Interne ***Datenstrukturen*** des Schedulers:

  Typischerweise ein oder mehrere *Queues* (eine pro Priorität),
  eventuell *Bäume* (z.B. Linux CFQ-Scheduler)
  oder *Priority Queues* (Sonderform von Bäumen, wurden dafür erfunden).

- Scheduler auf ***großen Systemen***:

  Problem mit klassischen Schedulern bei Rechnern mit *vielen Cores*:
  Alle Prozessoren teilen sich dieselben *zentralen Scheduler-Datenstrukturen*
  ==› Zugriff darauf muss synchronisiert bzw. serialisiert werden
  (*"es darf immer nur einer", d.h. nicht mehrere Prozesswechsel gleichzeitig möglich*)
  ==› Wartezeiten und Performence-Probleme im Scheduler selbst

  Daher:
  Getrennte Scheduler-*Datenstrukturen pro Core*,
  "Umschaufeln" zwischen Prozessoren nur bei deutlich ungleicher Last.

3. Kapitel: Virtuelle Speicherverwaltung

Ziele:

1.) Möglichst ***effiziente RAM-Nutzung***, z.B.
- Von *mehreren* Prozessen benutzer Code liegt nur *einmal* im Speicher
- Im RAM liegen nur Code und Daten, die *wirklich benutzt* werden
- Im Notfall kann in Summe *mehr* Speicher angefordert / benutzt werden, als das System RAM hat.

2.) ***Schutzfunktion***:
- Schutz der *einzelnen Prozesse* und ihrer Daten voreinander
- Schutz des *Betriebssystems* vor Anwender-Programmen
- Erkennen und Abfangen von *"Fehlzugriffen"* (z.B. Nullpointer)
- Ausführung von *Daten als Code* verhindern (wichtige Malware-Angriffs-Technik!)

3.) Finden aller Speicherbereiche, die zu einem Prozess gehören
==› *Sauberes Aufräumen* auch bei unsauberem Prozess-Ende (Absturz)

4.) Vereinfachung der Programmierung:

Jedes Programm "sieht" das RAM,
als ob ihm der Rechner alleine gehören würde!

==› Code und Daten stehen bei allen Programmen immer an *derselben Adresse*
==› Jedes Programm sieht *zusammenhängende, fortlaufende Speicherbereiche*

Egal, was sonst noch alles gleichzeitig im Speicher liegt!

Idee:

Virtueller Speicher
bzw.
virtuelle Adressierung

- Die Adressen, die ein Prozess für seinen Code und seine Daten verwendet, sind *"virtuelle Adressen"*:

Sie entsprechen nicht den tatsächlichen RAM-Adressen!

- Die Prozessor-Hardware, konkret die

MMU (Memory Management Unit)

in jedem Core bildet diese virtuellen Adressen bei jedem Speicherzugriff automatisch auf die "echten" RAM-Adressen ab:

"reale Adressen" bzw. "physische Adressen"

- Diese Abbildung wird *durch das Betriebssystem gesteuert bzw. konfiguriert*.
- *Jeder Prozess* hat
  - seinen *eigenen virtuellen Adressraum*
    (d.h. mehrere Prozesse können unabhängig voneinander *dieselben* virtuellen Adressen benutzen)

    - und *seine eigene Abbildung von virtuellen auf reale Adressen*
      (d.h. derselbe virtuelle Adressbereich in verschiedenen Prozessen
      kann auf verschiedene RAM-Bereiche abgebildet werden,
      aber auch auf dieselben RAM-Bereiche, z.B. bei gemeinsamem Code.
      Umgekehrt können verschiedene virtuelle Adressen in verschiedenen Prozessen
      auch auf denselben RAM-Bereich verweisen).
- Umgekehrt *"sieht"* jeder Prozess *nur seinen eigenen Speicher*,
  sowohl virtuell (weil er von der Existenz anderer Adressräume gar nichts weiß)
  als auch real (weil die RAM-Bereiche anderer Prozesse in seinem virtuellen
  Adressraum nicht eingeblendet und damit für ihn gar nicht erreichbar sind).
- Der *Kernel* hat traditionell seinen *eigenen virtuellen Adressraum*.
  Üblicherweise ist *das gesamte RAM* des Rechners im Adressraum des Kernels
  vollständig und linear fortlaufend eingeblendet (d.h. der Kernel sieht alles).

Im Detail:

- Die *Zuordnung* von virtuellen zu realen Adressen geschieht *nicht Byte für Byte*,
  sondern in

  Blöcken fixer Größe, sogenannten Pages

  (deren Größe ist durch die Prozessor-Hardware vorgegeben, meist 4 KB):

  - Der *gesamte virtuelle Adressraum* jedes Prozesses wird in 4-KB-Pages geteilt
  - Das *RAM* wird in 4-KB-Pages geteilt
  - Sowohl virtuelle als auch reale Pages beginnen auf durch 4 K teilbaren Adressen
  - Eine RAM-Page wird einer virtuellen Page zugeordnet

  Aufeinanderfolgende virtuelle Pages müssen nicht in aufeinanderfolgenden
  RAM-Pages gespeichert werden, sondern können aus ganz *verstreut liegenden
  RAM-Pages* zusammengestellt werden
  (d.h. man braucht für eine große, zusammenhängende virtuelle
  Speicher-Anforderung keine physisch zusammenhängenden RAM-Pages,
  was die Suche nach freiem RAM viel einfacher macht).

  Dass *alle Speicherblöcke gleich groß* sind, macht die Speicherverwaltung generell
  sehr viel *einfacher*: Es gibt *keinen Verschnitt* zwischen Blöcken verschiedener Größe,
  es gibt keine Suche nach einem freien Bereich der richtigen Größe, ...
- Daher (bei 4 KB Pages):

  - Die *hinteren 12 Bit* der virtuellen Adresse sind das *Offset innerhalb der Page*
    (sowohl innerhalb der virtuellen als auch innerhalb der realen Page).

    Sie werden *unverändert* von der virtuellen in die reale Adresse *übernommen*.
  - Alles *vor den letzten 12 Bit* in der virtuellen Adresse wird als *Page-Nummer*
    betrachtet und von der MMU durch eine Page-Adresse im RAM *ersetzt*.

Page Tables

Grundsätzliches Konzept der Page Tables:

- Festgelegt wird die *Zuordnung von virtuellen zu realen Adressen* durch die

Page Tables.

Das sind (gedanklich) große Arrays im RAM, die die *Page Table Entries* enthalten. Jeder Page Table Entry ist für *eine* Page zuständig.

- *Pro Prozess existiert eine separate Page Table*,
  zusätzlich hat der Kernel meist seine eigene Page Table.

  Welche Page Table gerade *aktiv* ist, legt ein *Register im Prozessor* fest (bei x86: Register CR3): Es zeigt auf den Anfang der aktuellen Page Table. Dieses Register darf *nur vom Betriebsystem* (im Supervisor-Mode) geändert werden.

- Die Page Tables sind *nach virtuellen Adressen organisiert*, d.h. die *vorderen Bits* (Page-Nummer) einer virtuellen Adresse sind der *Index in die Page Table* und bestimmen, welcher Page Table Entry für diese Adresse bzw. Page zuständig ist.

Wären die Page Tables wirklich *flache*, lineare Arrays, würden sich *zwei Probleme* ergeben:

- Der *Speicher-Overhead* für die Page Tables ist nicht vernachlässigbar: Er beträgt je nach Prozessor in etwa 0,1 % des virtuellen Adressraumes, also rund 1 KB pro MB. Da die Summe der virtuellen Adressräume aller Prozesse aber um ein Vielfaches (ev. um ein Hundertfaches oder mehr!) größer sein kann als das reale RAM, können die Page Tables durchaus einige Prozent des realen RAM's belegen.

  Konkret würde ein *32-Bit-Adressraum* (4 GB) auf x86-Prozessoren eine *4 MB große Page Table* erfordern (und zwar jeweils eine *pro Prozess*!), auf 64-Bit-Systemen (bei denen derzeit nur 48 Bit der Adresse signifikant sind, die oberen 16 Bit werden ignoriert) wäre es schon 512 GB, weil es ja für jede mögliche 32-Bit-Adresse bzw. 48-Bit-Adresse einen Eintrag geben muss, auch für die nicht gültigen, vom Programm nicht belegten Adressen (in der Realität wären bei "flachen" Page Tables wohl weit über 95 % aller Einträge solche, die *keinem realen RAM* entsprechen!).

  Ein Ziel ist daher, für große, zusammenhängende, *unbenutzte Bereiche* virtueller Adressen den Status *"keine gültige Adresse"* *deutlich kompakter zu speichern* als durch entsprechend viele einzelne PTE's.

- Die Page Tables selbst haben keine virtuellen, sondern reale Adressen. Für die 4 MB große Page Table eines jeden Prozesses müsste daher ein *zusammenhängender, 4 MB großer Block* im realen RAM reserviert werden.

  Das widerspricht aber fundamental der Idee, das gesamte RAM ausschließlich in lauter 4 KB großen, voneinander unabhängigen Pages zu verwalten, und würde die ganze Speicherverwaltung des Kernels *um ein Vielfaches komplizierter* machen.

  Es ist also notwendig, die *Page Tables nicht als durchgehendes Array* zu speichern,

sondern in Blöcke in der Größe einer Page aufzuteilen.

Man organisiert die Page Tables daher ***hierarchisch*** (mehrstufig):

- Man teilt das große Page-Table-Array in lauter einzelne *Blöcke in der Größe einer Page*.
- Diese Blöcke bilden das *unterste (letzte) Level* der Page Table.
  Sie enthalten wie bisher die Einträge, die *einer virtuellen Adresse eine RAM-Page zuordnen*.
- Wir nehmen einmal an, in einem solchen Block haben n Page Table Entries Platz.

  Dann legt man für *je n Blöcke* des letzten Levels *einen Block des vorletzten Levels* an.

Die *Einträge* der Blöcke des vorletzten Levels zeigen *nicht auf je eine RAM-Page*, sondern auf jeweils *einen Page-Table-Block des letzten Levels*.

- Dann wird für je n Blöcke des vorletzten Levels ein Block des vorvorletzten Levels angelegt, der auf diese Blöcke zeigt, usw.
- Das wiederholt man, bis es *einen einzigen obersten Block* für die ganze Page Table gibt.

Auf x86 gilt:

- Im *32 Bit Modus* umfasst *ein 4 KB großer Block* der Page Table *1024 Entries* zu je 4 Bytes. Man braucht daher 10 Bits (2^{10} = 1024) als Index pro Block bzw. Level.

  Die Page Tables sind *zweistufig*. Die virtuelle Adresse wird in 3 Teile geteilt:
  - 10 Bit Index in die First Level Page Table (diese enthält nur einen Block),
  - 10 Bit Index in die Second Level Page Table (max. 1024 Blöcke),
  - 12 Bit Offset innerhalb der Page.

  Ein Eintrag in der First Level Page Table (bzw. ein Block der Level-2-Page-Table) deckt daher *4 MB* virtuellen Adressraum ab.

- Im *64 Bit Mode* umfasst *ein 4 KB großer Block* der Page Table *512 Entries* zu je 8 Bytes. Man braucht daher 9 Bits (2^9 = 512) als Index pro Block bzw. Level.

  Im *64 Bit Mode* werden *derzeit nur 48 Bit große Adressräume* unterstützt (256 TB), die Page Tables dafür sind *vierstufig*
  (fünfstufige Page Tables für größere Adressräume sind gerade in Vorbereitung).
  Die virtuellen 48-Bit-Adressen werden in 9 / 9 / 9 / 9 / 12 Bits aufgeteilt.

  Ein Level-1-Entry ist daher für *512 GB* zuständig,
  ein Level-2-Entry für *1 GB*,
  ein Level-3-Entry für *2 MB*,
  und ein Level-4-Entry für eine einzelne 4-KB-Page.

Der Vorteil dieses Konzeptes liegt darin, dass man große virtuelle Adressbereiche, denen *kein RAM* zugeordnet ist, *gleich in den oberen Levels der Page Tables als ungültig* markieren kann, und dann für diese Bereiche *gar keine Page-Table-Blöcke in den unteren Levels* braucht (bzw. die einzelnen Page-Table-Blöcke der unteren Levels erst *"on Demand"* anlegt, wenn wirklich auf den Speicherbereich zugegriffen wird und der Eintrag in der oberen Page Table auf "gültig" gesetzt wird).

Beispiel für x86 32 Bit (im Vergleich zu einer 4 MB großen "flachen" Page Table):
Wenn ein Programm nur *4 MB ganz oben* und *32 MB ganz unten* in seinem Adressraum benutzt (was eine typische und für viele Programme ausreichende Speicherbelegung ist), dann brauchen seine Page Tables nur 40 KB statt 4 MB:
Einen Page-Table-Block zu 4 KB für die Level 1 Page Table (in dem alle Einträge außer den untersten 8 und dem obersten auf "ungültig" gesetzt sind),
einen Level-2-Block für die oberen 4 MB, und 8 Level-2-Blöcke für die unteren 32 MB.

Und solange es große, freie, zusammenhängende RAM-Bereiche gibt, können große Speicheranforderungen auch gleich durch Zuordnung von 4 MB großen RAM-Blöcken in der Level-1-Page Table bedient werden, was ebenfalls die entsprechenden Page-Table-Blöcke auf Level 2 einspart.

Page Table Entries (PTE's)

Ein einzelner Page Table Entry enthält folgende Informationen:

- Ein ***Adressfeld***, genau so groß, dass der *vordere Teil der realen Page-Adressen* (ohne Offset, d.h. ohne die letzten 12 Bit) darin Platz hat. Es enthält:
  - Die ***Adresse der Page im RAM***, wenn es sich um einen *Last-Level-PTE* einer im RAM befindlichen Page handelt.
  - Die ***Adresse des untergeordneten Page-Table-Blocks***, wenn es sich um einen "gültigen" PTE in einer *Page Table auf höherem Level* handelt.
  - Irgendeine ***Betriebssystem-Information*** (z.B. eine Disk-Block-Nummer), wenn es sich um einen PTE handelt, dem *kein RAM zugeordnet* ist.
- Ein ***"gültig"-Bit***, das anzeigt, ob diesem PTE bzw. dieser virtuellen Adresse gerade *Speicher im RAM zugeordnet ist oder nicht*.
- Bits, die die ***erlaubten Speicher-Zugriffe*** festlegen:
  - *Lesen, Schreiben, Ausführen*
  - *"Kernel only"* (siehe unten)
- Ein ***"Used"-Bit***, das *bei jedem Zugriff* auf die Page gesetzt wird
- Ein ***"Dirty"-Bit***, das *bei jedem Schreiben* in die Page gesetzt wird, d.h. das anzeigt, ob der Inhalt der Page verändert wurde, seit sie angelegt bzw. ins RAM geladen wurde.
- Eventuell ***einige Bits***, die vom ***Betriebssystem*** frei verwendet werden können.

Der **Ablauf** bei jedem Speicherzugriff, egal ob Code oder Daten, ist daher folgender:

- Die MMU zerlegt die virtuelle Adresse in ihre Teile,
- nimmt die vorderen Teile als Index in die Page Tables des aktuellen Prozesses und *lädt die entsprechenden Page Table Entries aus dem RAM*,
- *prüft* diese PTE's, ob der virtuellen Adresse eine *reale Adresse* zugeordnet ist, und ob die *Rechte* den aktuellen Speicherzugriff erlauben,
- wenn *ja*, macht einen *RAM-Zugriff* auf die reale Adresse, die sich aus den PTE's und dem Page Offset in der virtuellen Adresse ergibt, und setzt das *"Used"-Bit* und ev. das *"Dirty"-Bit*, wenn sie noch nicht gesetzt sind,
- und wenn *nein*, löst einen *"Page Fault"* aus (siehe unten), d.h. übergibt die *Kontrolle an das Betriebssystem*.

Das *Betriebssystem* ist also *bei "normalen" (erfolgreichen) Zugriffen nicht beteiligt*, diese laufen *rein in Hardware*.

Das ***Betriebssystem***

- *verwaltet* das RAM, insbesondere den *freien Platz*,
- *erstellt* (vor dem Start eines neuen Prozesses) und *aktualisiert* (bei Auftreten eines "Page Faults") die *Page Tables*,
- und bricht Prozesse bei *ungültigen Speicherzugriffen* (auch "Page Fault") ab.

Der "Translation Lookaside Buffer"

Da die Page Tables ja auf Grund ihrer Größe auch im normalen RAM liegen,
würde das bisher vorgestellte Konzept jeden Speicherzugriff extrem verlangsamen:
Bei *jedem Speicherzugriff* müssten zuerst einmal die betroffenen *Page Table Entries*
aus mehreren Levels Page Tables *aus dem RAM geladen* und geprüft werden.

Daher enthält die MMU *in jedem Core* einen

Cache für mehrere Dutzend oder Hundert Page Table Entries:
Dieser Cache heißt TLB ("Translation Lookaside Buffer")

und speichert die zuletzt benutzten Page Table Entries
(bei großen Cores gibt es sogar zwei Levels TLB).

Aber bei jedem System Call oder *Prozesswechsel* wird die *Page Table umgeschaltet*
(auf die Page Table des Kernels oder eines anderen Prozesses).
Damit werden alle Einträge im TLB *ungültig* (weil sie zur "falschen" Page Table gehören),
d.h. der TLB muss *jedesmal komplett geleert* werden.

Die ersten Code- und Daten-Zugriffe nach einer solchen Umschaltung dauern daher
besonders lang, weil die dafür nötigen Page Table Entries erst frisch aus dem RAM geladen
werden müssen (dieser TLB Flush war jahrelang der Hauptgrund, warum ein System Call
oder ein Prozesswechsel einige 1000 Takte gekostet hat).

Man versucht dieses Problem mit zwei Maßnahmen zu reduzieren:

- Die Page Table Entries bekommen ein ***zusätzliches Rechte-Bit "Kernel Only"***,
  das von der MMU geprüft wird: Es besagt, dass diese Page zwar gültig ist,
  aber dass ein Zugriff nur für Code erlaubt ist, der im Supervisor Mode läuft
  (also nur für den Kernel).

  Mit diesem Bit ist es möglich, in der Page Table *jedes einzelnen Prozesses* den
  kompletten Kernel-Bereich einzublenden (weil die User-Daten und die Kernel-Daten
  verschiedene virtuelle Adressbereiche nutzen, kollidieren die Einträge nicht),
  ohne dass der Prozess dadurch auf Kernel-Daten zugreifen kann
  (weil das "Kernel Only"-Bit den Zugriff verhindert).

  Man *erspart* sich damit den *Wechsel zur Page Table des Kernels bei jedem System Call*,
  weil die Page Table jedes Prozesses dem Kernel vollen Zugriff auf alle seine Daten
  ermöglicht.

- Weiters hat man in den aktuellsten x86-Prozessoren in jedem Page Table Entry
  zusätzlichen Platz für eine Art ***Prozesskennung*** reserviert
  (das PCID-Feature: *Process Context Identifier*).

  Die MMU *vergleicht* bei jedem Zugriff, ob der PCID des Page Table Entry
  aus dem TLB zum aktuell laufenden Prozess passt. Wenn nicht,
  verwirft sie den TLB-Eintrag und lädt ihn neu aus dem RAM. Damit muss man
  bei einem Prozesswechsel nicht mehr vorbeugend den gesamten TLB leeren.

  Wechselt ein Core immer wieder zwischen *nur wenigen Prozessen* hin und her,
  teilen sich diese den TLB: Bekommt ein Prozess den Core, so hat er gute Chancen,
  dass seine TLB-Einträge *vom letzten Mal zumindest noch teilweise vorhanden* sind,
  und muss nicht nach jedem Prozesswechsel mit komplett leerem TLB beginnen.

Page Faults

Ein ***"Page Fault" (Seitenfehler)***
ist ein ***Zugriff*** eines Programms auf eine ***virtuelle Adresse***,
für die in der Page Table (noch) ***keine RAM-Adresse zugeordnet*** ist,
d.h. auf eine virtuelle Page, deren Page Table Entry den ***Status "ungültig"*** enthält.

Auch ein Verstoß gegen die im Page Table Entry festgelegten ***Zugriffsrechte***
beim Zugriff auf eine an sich gültige Seite löst einen Page Fault aus.

Bei einem Page Fault schickt die MMU einen ***Interrupt***, d.h. das laufende Programm wird *unterbrochen und die Kontrolle an das Betriebssystem übergeben*.

Das Betriebssystem
- *lädt* entweder *die Page nach* (und setzt den PTE auf "gültig")
- oder *bricht das Programm ab*.

Der ganze Mechanismus ist so gebaut, dass der betroffene Prozess
den ***abgebrochenen Befehl bzw. Speicherzugriff automatisch wiederholt***,
nachdem das Betriebssystem die Page geladen und die Page Tables modifiziert hat,
und daher vom Page Fault bzw. Seiten-Lade-Vorgang überhaupt *nichts bemerkt*.

Freie Pages

Alle RAM-Pages, die momentan *weder irgendeiner virtuellen Page* irgendeines Prozesses zugeordnet sind noch *Code oder Daten des Kernels* selbst (bzw. Page Tables) enthalten, werden für die laufende Programmausführung nicht benötigt, sind also *"frei verfügbar"*.

Da aber ***"freier Speicher verschwendeter Speicher ist"***, weil er ja zur Systemleistung überhaupt nichts beiträgt, wird das Betriebssystem versuchen, einen Großteil der "frei verfügbaren" Pages intern zur *Steigerung der Systemleistung* zu nutzen:

- ***Disk Cache:*** Von der Disk *in letzter Zeit gelesene (oder geschriebene) Daten* werden im RAM gehalten, damit sie nicht nochmals gelesen werden müssen, wenn man ein zweites Mal darauf zugreift.
  Alle weiteren Zugriffe werden dadurch um Größenordnungen schneller.

  Das betrifft nicht nur Files mit Anwendungsdaten, sondern vor allem häufig benutzte *Programme und Libraries* sowie *Directories und Filesystem-Metadaten* (z.B. das Verzeichnis freier Platten-Sektoren).

- ***File Readahead:*** Erkennt das Betriebssystem, dass ein *File sequentiell gelesen* wird, lädt es die *nächsten Daten aus diesem File auf Vorrat* ins RAM, damit sie beim nächsten Lesezugriff sofort verfügbar sind.

  Dies gilt sowohl für Files, die per File-I/O gelesen werden (Anwendungsdaten), als auch für Files, die mittels Page Faults gelesen werden (.exe-Files, Libraries, mmap-Files).

- **Write Buffer:** Schreibt ein Prozess Daten in einen File, speichert das System die Daten *zuerst einmal ins RAM* und schreibt sie *erst später auf Platte*.
  (bei Linux im Notebook-Mode kann das bis zu 10 Minuten dauern, damit die Platte länger abgeschaltet werden kann).
  Einerseits kann der Prozess dadurch sofort weiterarbeiten und muss nicht auf den Disk-Schreibvorgang warten, und andererseits spart das unnötige

Disk-Schreibvorgänge, wenn der File sofort wieder geändert oder gelöscht wird.

Besonders effizient (aber auch gefährlich!) ist diese Strategie bei Directories und Filesystem-Metadaten, die oft mehrmals kurz hintereinander geändert werden und wo sich daher viele Änderungen, die sonst jedesmal Disk-I/O zur Folge hätten, *zu einem einzigen Schreibvorgang zusammenfassen* lassen.

Nur ein geringer Teil der "frei verfügbaren" Pages kommt wirklich ein die vom Kernel verwaltete *Liste freier RAM-Pages*. Wird zur *Bedienung eines Page Faults RAM benötigt*, entnimmt der Kernel *aus dieser Liste* eine freie Page.

Paging / Swapping

Anmerkung:
Die *genaue Definition* und Unterscheidung der Begriffe "Paging" und "Swapping" ist *sehr uneinheitlich*!

Wenn das ***RAM knapp*** wird, d.h. wenn das Betriebssystem *Platz für neue Pages* braucht, aber im RAM keine Pages frei sind, hat das Betriebssystem zwei Ansatzpunkte:

- Einerseits *reduziert es seine File Caches und Readahead- bzw. Write-Buffer*.
- Andererseits werden

  benutzte bzw. gültige Seiten laufender Prozesse aus dem RAM entfernt (und in den Page Tables wieder auf "ungültig" bzw. "nicht im RAM" gesetzt).

  Dafür gibt es in Wesentlichen *zwei Fälle*:

  - Pages, die ***1:1 einen File-Inhalt*** enthalten (Code und Konstanten aus .exe-Files und Shared Libraries, mmap-Pages), können *einfach verworfen* werden, da sie bei Bedarf wieder angelegt und *aus dem File befüllt* werden können.

    Diese Pages können also *ohne Disk-I/O freigegeben* werden (außer mmap-Pages, die "dirty" sind: Diese müssen zuerst in den File zurückgeschrieben werden).

  - Für Pages, deren Inhalt *nicht aus einem File* wiederherstellbar ist (das gilt für praktisch ***alle Daten-Bereiche***), muss Platz in einem *speziellen Bereich der Platte* (***"Pagefile", "Swap Partition"***) reserviert werden. Dorthin wird der *aktuelle Inhalt der Page gesichert*, bevor die Page freigegeben wird.

    Greift der Prozess wieder auf diese Daten zu, muss im RAM eine Page reserviert und mit den gesicherten Daten *aus dem Pagefile gefüllt* werden (die Page landet dabei im Normalfall an einer ganz anderen Stelle im RAM als ursprünglich).

    Das Freischaufeln der Page erfordert also einen *Disk-Schreibvorgang* (außer die Page wurde schon einmal aus- und wieder eingelagert und ihre Kopie auf Platte ist noch aktuell, d.h. die Page ist nicht "dirty").

Das "Freischaufeln" von Platz beginnt *nicht erst*,
wenn ein Prozess mehr RAM braucht und die Freiliste *komplett leer* ist (denn das würde alle Platz anfordernden Prozesse *viel zu lange blockieren*), sondern *läuft "auf Verdacht" ständig im Hintergrund* (***"Page Stealer"***) und befüllt die Freiliste, sobald die Anzahl der *freien RAM-Pages unter einen bestimmten Grenzwert* fällt,

damit RAM-Anforderungen jederzeit "sofort" befriedigt werden können.

Der Page Stealer garantiert also einen *gewissen Vorrat an freien Pages im RAM*
und nimmt Prozessen dazu notfalls "vorbeugend" Speicher weg und lagert ihn aus.
Wenn der Page Stealer eine Page vorbeugend in den Pagefile kopiert und freigegeben hat,
und der Prozess greift wieder darauf zu, bevor die Page für andere Zwecke verwendet wurde, dann kann sie ohne Disk-I/O sofort wieder zugeordnet werden.

Der Zugriff auf ausgelagerte Daten ist *um Größenordnungen langsamer* als ein RAM-Zugriff (10 ms vs. 50 ns) und *belastet CPU und Platte*. Bei gravierendem Speichermangel
(wenn die laufenden Prozesse so wenig RAM bekommen, dass sie jeweils nach wenigen ausgeführten Befehlen schon wieder RAM nachladen und auf die Platte warten müssen)
ist das *System nur mehr bedingt nutzbar*: Das System *wartet vor allem auf die Platte*,
die *"nutzbringende" Rechenarbeit sinkt* auf wenige Promille der CPU-Leistung,
die *Reaktionszeit der Prozesse* auf Benutzer-Interaktion liegt im hohen Sekunden-Bereich.

Dieser Zustand ***("Trashing" bzw. "Seitenflattern")*** sollte daher nach Möglichkeit *vermieden* werden: Es ist sinnvoll, in diesem Fall die Anzahl oder den RAM-Bedarf der laufenden Prozesse zu reduzieren oder das RAM zu erweitern,
um die Effizienz (den "nutzbringenden" Durchsatz) des Systems wieder zu steigern.

Als Faustregel (zumindest unter Linux/Unix) gilt, dass das System anders ausgelegt werden sollte, wenn die *ausgelagerte Datenmenge die RAM-Größe übersteigt*
(also weniger als 50 % der laufend benutzten Daten im RAM gehalten werden können).

Echtzeit-Systeme usw. sollten *völlig ohne Pagefile bzw. Swapspace* konfiguriert werden,
weil Paging und Swapping das zeitliche Verhalten unvorhersehbar beeinflussen.

Alternativ bieten Betriebssysteme die Möglichkeit,
dass *hochpriore Prozesse* ihren *Speicher* ganz oder teilweise *"anpinnen"*,
d.h. sie bekommen *RAM fix vorab zugeordnet* (nicht erst bei Bedarf),
und dieses RAM wird *nie ausgelagert*.

Working Set

*Die **Menge der Seiten** (Code und Daten),*
*die ein Prozess **aktuell wirklich benutzt***
(nicht nur reserviert bzw. zugeordnet hat,
*sondern tatsächlich **immer wieder darauf zugreift**),*
*heißt **"Working Set"** des Prozesses.*

Kann das Betriebssystem zumindest dieses Working Set *ständig im RAM* halten,
läuft der Prozess ohne deutliche Verzögerung, steht ihm weniger RAM zur Verfügung,
wird er massiv verlangsamt.

Tools wie der Task Manager oder "`ps`" unter Linux können für jeden Prozess sowohl die *Größe seines virtuellen Speicherbereiches* als auch des *real belegten RAM's* anzeigen.

Replacement-Strategien

Für die Effizienz eines Systems mit Paging ist es ganz entscheidend,
dass bei Platzmangel diejenigen Seiten aus dem Speicher verdrängt werden,
die *in absehbarer Zeit nicht mehr benötigt* werden
(bzw. im Idealfall in Zukunft gar nicht mehr benötigt werden).

Da ein Blick in die Zukunft aber nicht möglich ist, muss man sich bei der Auswahl der zu verdrängenden Seiten *am Zugriffsverhalten der Vergangenheit orientieren*:
Die Grundidee ist, dass die Wahrscheinlichkeit eines Zugriffes in naher Zukunft umso geringer ist, *je länger die Seite nicht mehr benutzt wurde*, d.h. man will diejenigen Seiten auslagern, bei denen seit dem letzten Zugriff die meiste Zeit vergangen ist.

Leider wäre eine genaue Buchführung über die Zeit des letzten Zugriffs viel zu aufwändig, man muss daher eine *näherungsweise Lösung* finden.

Die "Replacement-Strategie" legt fest,
wie die zu verdrängenden Seiten ermittelt werden.

Dabei hilft, dass die MMU bei jedem Zugriff auf eine Page hardwaremäßig automatisch das ***Used-Bit*** in der Page Table setzt:

- Eine Idee ist, die Pages beim Anlegen in eine *verkettete Liste* zu hängen.
  Bei Speichermangel geht man diese Liste *vom "alten" Ende her* durch.

  Hat die älteste Page das Used-Bit *gesetzt*, nimmt man sie hinten aus der Liste, hängt sie als "jüngste" Page wieder ein, und löscht ihr Used-Bit.

  Ist das Used-Bit der ältesten Page *nicht gesetzt*, so wurde sie sehr lange nicht mehr benötigt und wird ausgelagert (und auch aus der Liste entfernt).

- Statt der sich ständig ändernden Liste kann man auch eine *zyklische Liste* aller Page Table Entries verwenden, in der ein *Pointer ständig im Kreis läuft*:

  Neue Pages werden *hinter* dem Pointer eingehängt

  Ist die aktuelle Page (auf die der Pointer zeigt) *"Used"*,
  so bleibt sie in der Liste, aber das "Used"-Bit wird *gelöscht*,
  und der Pointer wechselt auf die Page davor.

  Ist die aktuelle Page *nicht "Used"*, so ist seit dem letzten Zugriff mindestens ein kompletter Umlauf der zyklischen Liste vergangen:
  Die Page wird aus der Liste genommen und *ausgelagert*.

- Eine ganz andere Strategie ist folgende:

  Für jede Page wird ein *"Alters-Zähler"* gespeichert (in x86-Page Table Entries sind beispielsweise drei Bits ohne hardwaremäßig definierte Aufgabe vorhanden, die vom Betriebssystem dafür benutzt werden können).

  Der *"Page Walker"* durchläuft in regelmäßigen Abständen alle Page Tables.
  Ist das *Used-Bit gesetzt*, wird der *Alters-Zähler auf 0* gesetzt
  (weil die Page frisch benutzt ist) und das Used-Bit gelöscht.
  Ist das *Used-Bit nicht gesetzt*, wird der *Alters-Zähler um eins erhöht*,
  weil auf die Page eine komplette Periode lang nicht zugegriffen wurde.

  Bei Speichermangel durchläuft der "Page Stealer" die Page Tables und sucht nach Pages mit gelöschtem Used-Bit und hohem Alters-Zähler.

Daneben spielt auch die *Größe des Working Sets* bei der Replacement-Strategie eine Rolle: Die Strategie kann *große Prozesse benachteiligen* und kleine bevorzugen.
Dadurch bekommen große Prozesse weniger bzw. seltener CPU,
und der Paging-Druck verringert sich dadurch hoffentlich.

Wenn sichergestellt ist, dass ein großer, sehr paging-intensiver Prozess

ein *reiner Batchjob* ist, der nur die Restkapazitäten des Rechners sinnvoll nutzen soll, dann kann dieser Prozess bei extremem Speicherdruck auch bis zur Beruhigung der Lage *komplett unterbrochen* und sein gesamter Speicher in einem Rutsch *vollständig ausgelagert* werden: Das bringt einerseits viel freien Platz auf einmal und andererseits meist eine deutliche Reduktion des Pagings, weil zumindest dieser Prozess nicht mehr laufend Speicher nachfordert.

Memory Overcommit

"Memory Overcommit" liegt vor, wenn allen Prozessen vom Betriebssystem *insgesamt mehr virtueller Speicher* zugesichert wurde, als das System *real in Summe von RAM und Pagefile* wirklich hat.

Beispielsweise endet ein **`malloc`** erfolgreich, obwohl gar nicht mehr genug freier "echter" Speicher (RAM+Swap) vorhanden ist, um die gesamte angeforderte Datenmenge wirklich zu speichern, oder ein neues Programm lässt sich starten, obwohl alle seine Daten-Bereiche gar nicht komplett ins RAM geladen werden können.

"Memory Overcommit" ist also *eine Wette* darauf, dass *nie* alle Programme gleichzeitig den Ihren virtuell zugeordneten Speicher wirklich *zu hundert Prozent benutzen* (was zumindest unter Unix konzeptbedingt meist ein sicherer Tipp ist).

- *Windows* lässt standardmäßig *kein Overcommit* zu: Bei jeder Speicher-Anforderung wird *sofort "echter" Platz im vollen Umfang* der Anforderung im RAM oder im Pagefile *reserviert*. Ist die nicht möglich, schlägt die Anforderung fehl (**`malloc`** liefert Fehler, Programm lässt sich nicht starten, mmap schlägt fehl, …).

  Das ist die *sichere Variante* (weil es nie zu der fatalen Situation kommt, dass ein Zugriff auf aus der Sicht des Programmes "vorhandenen" virtuellen Speicher mangels realem Speicher scheitert), aber sie nutzt die Hardware sehr *ineffizient* (weil Programme schon mit Fehlermeldungen enden, obwohl RAM und Pagefile de facto noch deutlich mehr Last bedienen könnten).

- *Linux* bietet verschiedene Einstellungen, per Default wird *Overcommit zugelassen*. Daher kann es passieren, dass der erste tatsächliche Zugriff auf schon erfolgreich angeforderten und zugewiesenen Speicher *scheitert*, weil es das System im Moment des Page Faults nicht mehr schafft, der betreffenden (logisch benutzbaren) virtuellen Page reales RAM zuzuordnen.

  In diesem Moment ist es aber *nicht mehr möglich*, dem Programm eine geordnete *Fehlermeldung* zukommen zu lassen (das Programm macht ja keinen System Call, sondern führt irgendeinen ganz normalen Befehl mit Speicherzugriff aus).
  In Linux greift in diesem Fall der ***"Out-of-Memory Killer"***: Er sucht nach irgendwelchen *Heuristiken* (Working-Set-Größe, bisherige Swap-Aktivität, …) ein oder mehrere Prozesse aus, die *ohne Vorwarnung gekillt* werden (ähnlich Ctrl/C), um wieder Platz zu schaffen.

Fortgeschrittene Anwendungen virtueller Speicherverwaltung:

1.) mmap (Memory mapped Files):

mmap ist eine *alternative Methode des Zugriffes auf Daten in Files*:

- Der *gesamte File-Inhalt* (bzw. in seltenen Fällen ein Teilbereich des File-Inhalts)

wird irgendwo in einen bisher noch *unbenutzen Bereich des virtuellen Adressraumes* eines Prozesses *eingeblendet*: Der mmap-System-Call liefert einen *Pointer auf den Anfang* des für den File reservierten Bereiches im Adressraum.

- Das Programm kann ausgehend von diesem Pointer auf den Fileinhalt zugreifen, als ob es sich um ein *riesiges Array* handeln würde (d.h. das Programm "sieht" den gesamten File-Inhalt als normales "Array of Bytes" im Speicher, beginnend bei diesem Pointer).
- Der Fileinhalt wird aber weder "vorbeugend" komplett in diesen Speicherbereich kopiert noch durch explizite Systemaufrufe geladen oder zurückgespeichert, sondern nur durch MMU-Mechanismen:
  - Am Anfang liegt noch *nichts* im RAM, der gesamte Bereich ist *"ungültig"* markiert.
  - Erst bei einem *Zugriff* auf den mmap-Bereich (*Page Fault*) wird eine freie RAM-Page gesucht und *genau dieser Teil* des Files dorthin geladen. Der File-Inhalt kann also *in beliebiger Reihenfolge* geladen werden.
  - Wird länger nicht auf die geladenen Daten zugegriffen, oder wird das RAM knapp, werden die Pages wieder freigegeben (sie können ja bei Bedarf wieder geladen werden).
  - *Schreibt* das Programm in einen mmap-Bereich, so markiert die MMU diese Page als *"dirty"*. *Nur die Dirty-Pages* werden periodisch, nach Ende des Programms, oder vor dem Freigeben wegen RAM-Knappheit in den File *zurückgeschrieben*.

  Mit mmap können Files *relativ effizient gelesen und geschrieben* werden (vor allem bei nicht-sequenziellem Zugriff), aber *nicht vergrößert* werden.

  Unter *Linux* werden *alle Shared Libraries* über den mmap-Mechanismus geladen. Außerdem dient mmap als *Shared-Memory-Mechanismus* (indem *mehrere Prozesse gleichzeitig denselben File* mappen: Die Pages liegen dann *nur einmal* im Speicher, nicht einmal pro Prozess, d.h. alle Prozesse "sehen" und ändern dieselben Daten).

  Weiters können durch mmap-Einblenden von Pseudo-Files wie **/dev/zero** große 0-initialisierte Speicherbereiche allokiert werden.

2.) "Copy on Write":

Gerade unter Linux kommt es konzeptbedingt sehr oft vor, dass ein Prozess eine *Kopie des Speicherbereiches* eines anderen Prozesses bekommt. Typischerweise wird *nur ein kleiner Teil dieser Kopie verändert*, weite Teile bleiben ident zum Original.

Gleich alles zu kopieren wäre daher sowohl Platz- als auch Zeitverschwendung. Der Kernel kopiert daher nichts, sondern blendet *denselben Speicherbereich* in beiden Prozessen ein, aber setzt ihn bei beiden auf *"Read only"*.

Versucht ein Prozess, seine "Kopie" zu beschreiben, löst das einen *Page Fault* aus. Erst dann *kopiert* das Betriebssystem die eine betroffene Page, gibt jedem Prozess eine separate Kopie, und setzt diese bei beiden Prozessen wieder auf "read/write".

3.) Zero Copy I/O:

Prinzipbedingt werden I/O-Daten *mindestens einmal umkopiert*, nämlich zwischen Anwendung (virtuellem Speicher eines Prozesses) und Kernel-Speicherbereich, weil ja die Device-Treiber des Kernels ihre I/O-Daten

normalerweise nur aus einem Kernel-Speicherbereich übernehmen können bzw. nur dort ablegen können.

Bei Durchsatz-intensiven I/O-Devices (schnelle Platten, schnelle Netzwerkkarten) verbraucht dieses Kopieren beträchtliche CPU-Resourcen.

Eine der möglichen Verbesserungen besteht darin, dass man die RAM-Bereiche, aus denen die Treiber Daten schreiben oder lesen, *abwechselnd* in die Adressräume des Kernels und die Adressräume der Prozesse einblendet, um sich das Kopieren zu sparen.

Zusammenfassung möglicher Page Faults:

a) ***Zugriff auf eine nicht gemappte Page***
(d.h. auf eine virtuelle Adresse, der keine reale RAM-Adresse zugeordnet ist, sondern die in der Page Table als "ungültig" markiert ist):

- Page liegt im ***Code- oder Konstanten-Bereich*** des virtuellen Adressraumes, d.h. in einem Bereich, dem *1:1 ein .exe- oder .dll-File* zugeordnet ist:
Nachladen der Page aus dem .exe-File oder einer Shared Library, weitermachen.

  Programme und Libraries werden auf modernen Systemen also *nicht vorbeugend komplett geladen*, sondern nur stückweise *"on demand"* (bei Bedarf) ins RAM geholt: ***"Demand Paging"*** (und bei Speicherdruck auch wieder aus dem RAM verdrängt).

- Page liegt *unmittelbar unter der letzten* ***Stack-Page***:
Stack um eine Page *vergrößern*, solange das Größen-Limit nicht überschritten ist.

- Page ist als ***"ausgelagert"*** markiert, d.h. enthält schon gültige Daten, aber wurde in den Pagefile bzw. Swapspace auf der Platte verlagert:
Page *zurück ins RAM* holen und Zugriff erlauben.

- Page liegt in einem noch nicht geladenen oder wieder verdrängten ***mmap-Bereich***: Entsprechenden Teil des Files ins RAM laden, Zugriff erlauben.

- Page liegt in einem angeforderten, aber ***noch nie benutzten Datenbereich*** (*"Zero Page"*, z.B. malloc oder uninitialisierte globale Variablen):
Neue RAM-Page ev. mit lauter 0-Bytes initialisieren, Zugriff erlauben.

- Page liegt in einem ***ungültigen Bereich*** des virtuelle Adressraumes, d.h. in einem Adressbereich, der dem Prozess *nie zugeordnet* wurde und weder Code noch Datenbereiche enthält: *Programm-Abbruch*
(***"Schutzverletzung"*** bzw. ***"General Protection Fault"*** unter Windows, ***"Segment violation (SIGSEGV)"*** unter Linux)

b) ***Verletzung des Speicherschutzes***
(z.B. Schreiben von nur-Lese-Pages, Ausführen von Daten-Pages, Zugriff auf "nur Kernel"-Pages...):

- Schreibzugriff auf eine nur-Lese-Page, die im Betriebsystem als ***"Copy on write"*** markiert ist:
Page kopieren und in der Page Table als schreibbar eintragen

- Sonst: ***Programm-Abbruch*** wie oben

Konsequenzen für die Programmierung

Je *weniger Pages* das Working Set eines Prozesses umfasst, umso *schneller* startet er, umso weniger belastet er das System (betr. RAM und Paging), und umso geringer ist die Gefahr, dass er während der Ausführung auf Paging warten muss.

Das Ziel ist daher, *Daten möglichst zusammenhängend* zu speichern und *Speicherzugriffe möglichst lokal* zu halten, damit sie sich auf möglichst wenige Pages verteilen (das steigert auch die Cache-Effizient beträchtlich!).

Für den Programmcode bieten Compiler Hilfsmittel, um alle *"heißen" Codestellen* in .exe-Files und Libraries möglichst kompakt und nahe beieinander anzuordnen und die *"kalten" Codeteile* in andere Bereiche des .exe zu verlagern.

Weiters gibt es Tools, die die Cache-Effizient messen oder simulieren.

x86 Segmentierung

Die x86-Prozessorfamilie unterstützt Paging *erst seit dem 80386*.
Beim ursprünglichen 8086 sowie bei 80186 und 80286 gab es nur eine *Segment-basierte Umsetzung von logischen (virtuellen) auf reale Adressen*, beim 80286 wurde diese erstmals um Zugriffs-Schutzmechanismen ergänzt.

Diese Segmentierung wird auch in heutigen x86-Prozessoren *noch von der Hardware unterstützt* (sie wirkt *vor* dem Paging-Mechanismus auf die virtuellen Adressen), gilt aber als veraltert und wird von den meisten Betriebssystemen *nicht mehr genutzt*. Wir besprechen sie hier daher nicht.

Der Segment-Mechanismus wird allerdings aktuell von einigen Betriebssystemen für zwei Zwecke eingesetzt, die mit der Grundaufgabe von Speicherverwaltung und Speicherschutz nur am Rande zu tun haben:

- In *Usermode-Prozessen* mit parallelen Threads wird ***"Thread-local storage"***, also globale/statische Variablen, von denen jeder Thread seine eigene Kopie hat, unter Verwendung des Segment-Mechanismus realisiert.
- Im *Kernel* wird der Segment-Mechanismus für Variablen und Datenstrukturen genutzt, die unter gleichem Namen und gleicher Adressierung ***ein Mal pro Core*** existieren.

4. Kapitel: I/O-Devices

Grundsätzliche Arten von Devices

(ohne Anspruch auf Vollständigkeit, ohne Sonderfälle wie z.B. GPU):

- ***Char-Devices:***

  Das sind Devices, die aus Sicht von System und Anwendungen primär *Daten byteweise und sequentiell lesen und/oder schreiben*.

  Beispiele: Tastatur, Maus, ser. und par. Schnittstelle, Textmode-Konsole, I/O-Devices in Steuerungen und Embedded Systems (I/O-Pins, A/D-Wandler usw.), Soundkarte, ...

- ***Block-Devices:***

  Das sind Devices, auf die aus Systemsicht *nicht byteweise* zugegriffen werden kann, sondern bei denen der Datentransfer *immer in Blöcken fixer Größe* erfolgt (Bei Platten meist 512 Bytes oder 4 KB, technisch ein Disk-Sektor, bei optischen Laufwerken 2 KB).

  Diese Devices enthalten eine *fixe Anzahl solcher Blöcke*, auf die nicht nur sequentiell, sondern *in beliebiger Reihenfolge* zugegriffen werden kann.

  Beispiele sind *Platten* (intern / extern), SSD's, CD's und DVD's, Bänder, SD-Cards, ...

  Aus *Anwendungssicht* kann zwar im Prinzip direkt blockweise auf solche Devices zugegriffen werden (z.B. beim Formatieren oder bei einem "Image Backup"), aber im Normalfall wird auf Block Devices *nur indirekt über ein Filesystem* zugegriffen.

- ***ioctl-Devices:***

  Das sind aus Anwendersicht (zumindest unter Linux) *normale Char-Devices*, aber ihr Hauptzweck bzw. der primäre Zugriff besteht *nicht im Datentransfer*, sondern im *Aufruf irgendwelcher Device-spezifischer System-Calls*.

- ***Netzwerk-Devices:***

  Auf Netzwerk-Devices wird *aus Anwender-Sicht weder byteweise noch in fixen Blöcken* noch über ein Filesystem zugegriffen, sondern primär mit eigenen Netzwerk-System-Calls, die die *Funktionalität der Netzwerk-Protokolle* abbilden.

Device-Zugriff

Auf unterster Ebene (in den *Device-Treibern*) kann der Datentransfer von und zu I/O-Devices auf mehrere Arten laufen:

- ***Polling:***

  Die CPU *wartet mittels Software-Schleife und wiederholter Prüfung eines Status-Bits* des Bausteins, bis der I/O-Baustein bereit zum Datentransfer ist, und kopiert die Daten dann *Byte für Byte softwaremäßig* vom oder zum Baustein.

  Nur für ganz kleine Embedded Systems geeignet, die parallel zum I/O-Transfer nichts anderes zu tun haben, da das Warten und das Daten Übertragen *die CPU vollständig belegt bzw. blockiert*.

- ***Interrupt + Software-I/O:***

  Der Baustein zeigt durch einen *Interrupt* an, dass er *bereit zum Datentransfer* ist.
  Der Interrupt-Handler überträgt die *Daten* dann *softwaremäßig*
  (per Kopier-Schleife).

  Vorteil: Die CPU ist in der Wartezeit *nicht mehr blockiert*
  (aber während des Datentransfers schon).

- ***DMA (Direct Memory Access) + Interrupt:***

  Der DMA-Controller ist ein *eigener Baustein* bzw. eine eigene Einheit in der CPU.
  Er kann auf eine Hardware-Anforderung (eigene Steuerleitung *"DMA Request"*) hin
  selbsttätig (ohne Zutun der CPU) *Daten zwischen Speicher und I/O-Devices kopieren*.
  Die CPU initialisiert den DMA-Controller dazu mit Quell-Adresse, Ziel-Adresse
  und Anzahl der zu übertragenden Bytes.

  Nachdem die CPU den I/O-Baustein und den DMA-Controller entsprechend
  initialisiert hat, signalisiert der I/O-Baustein dem DMA-Controller,
  wann er zum Datentransfer bereit ist, und der DMA-Controller *überträgt*
  ohne Zutun der CPU die Daten. Erst wenn der Datentransfer *fertig* ist, informiert
  entweder der I/O-Baustein oder der DMA-Controller per *Interrupt* die CPU.

- ***Busmaster + Interrupt:***

  Hier entfällt der separate DMA-Controller: Der *I/O-Baustein* kann selbst
  die Kontrolle über den *Speicher-Bus übernehmen* (wie die CPU) und
  selbsttätig Daten transferieren. Die CPU schreibt nur bei der Initialisierung
  des I/O-Requests zusätzlich zu den I/O-Steuerdaten die Quell- oder
  Ziel-Speicheradresse des Datentransfers im RAM in den I/O-Baustein.

  Wenn der Transfer abgeschlossen ist, löst der I/O-Baustein einen Interrupt aus,
  um das Ende des I/O-Requests anzuzeigen.

- ***Mailbox + Busmaster + Interrupt:***

  Das oben beschriebene Prinzip hat zwei Nachteile:
  - Die Steuerdaten für eine I/O-Operation, die die CPU zu Beginn
    in den I/O-Baustein schreiben muss, können ziemlich umfangreich sein,
    d.h. das Schreiben *dauert* vergleichsweise lange.
  - Die Steuerdaten für die *nächste* I/O-Operation können erst dann an den Baustein
    übertragen werden, wenn die vorhergehende Operation abgeschlossen
    und ihr Erfolgsstatus ausgelesen ist.

  Man kann die Busmaster-Fähigkeit aber nicht nur zum Nutzdaten-Transfer nutzen:
  Manche I/O-Bausteine sind in der Lage, auch die *Befehle und Steuerdaten*
  für ihre I/O-Operationen *selbsttätig aus dem Speicher* zu laden.
  Die CPU legt die I/O-Steuerdaten daher *im RAM* ab (ev. gleich für *mehrere*
  I/O-Operationen) und überträgt nur mehr *deren Adresse an den I/O-Baustein*.
  Dieser holt sich die Anweisungen, führt die I/O-Operationen aus,
  hinterlegt das Ergebnis jeder Operation in den RAM-Steuerdaten,
  und meldet den Abschluss der Operationen wieder per Interrupt.

5. Kapitel: Filesysteme

Filesysteme setzen auf *Block Devices* auf, aber *nicht* direkt auf *ganze Platten*
(außer bei Floppies, CD's und DVD's, und bei "kleinen" USB-Sticks, SD-Karten usw.),
sondern im einfachsten Fall auf Platten-*Partitionen* und in Servern auf *"Logical Volumes"*
eines Volume Managers.

Filesysteme betrachten Speichermedien also ganz abstrakt als

"Fixe Anzahl gleich großer Blöcke mit direktem Zugriff".

Sie sind *unabhängig* von der konkreten *Speicher-Hardware* bzw. den Device-Treibern
und (zumindest logisch betrachtet) auch *weitgehend unabhängig* von *Caching*, *Readahead*
und *Write Buffering*, das das Betriebsystem unabhängig davon für Disk-Blöcke betreibt.

Nur in einem Fall interagieren Filesysteme direkt mit der darunterliegenden Hardware
(und sind auch spezifisch an diese Hardware angepasst):
Bei *Filesystemen auf direkt angebundenen Flash-Chips* ohne eigenen Flash-Controller,
wie sie in Handies und Embedded Systems vorkommen:
Hier kümmert sich das Filesystem selbst um das nötige Umkopieren von Daten
und Löschen von Flash-Blöcken, um Prüfsummen und Fehlerkorrektur,
um Wear Leveling (gleichmäßige Verteilung der Schreibzugriffe auf alle Zellen)
und Bad-Block-Management bzw. Reserve-Blöcke usw..

Daneben unterstützen fast alle Betriebssysteme auch eine Verwendung eines
Teils des RAM's (meist bei Bedarf dynamisch reserviert) als *"Datenträger" für Filesysteme*
für temporäre Files oder für Files, bei denen es besonders auf Geschwindigkeit ankommt.

Anmerkung zu Unix/Linux

Auf Linux sieht es auf den ersten Blick so aus, als ob das System ein einziges Filesystem
über alle (internen und externen) Laufwerke verwaltet. Dieser Eindruck täuscht:

- Auch Linux verwaltet *ein Filesystem pro Partition* bzw. Logical Volume.
- Eines davon ist das *root-Filesystem*, d.h. das Filesystem,
  auf dem das Directory liegt, das der Benutzer als "**`/`**" (oberstes Directory) sieht.
  Das ist auch das Filesystem, das beim Systemstart normalerweise
  als erstes Filesystem "gemountet" wird.
- Unix / Linux erlaubt es, in ein beliebiges (sinnvollerweise leeres) Directory
  eines Filesystems (dem "Mount Point") *den kompletten Verzeichnisbaum
  eines anderen Filesystems* (auch auf einem anderen Laufwerk
  oder von einem anderen Filesystem-Typ) *einzublenden* ("zu mounten"):
  Man "sieht" die Directories z.B. eines CD-Laufwerks dann im root-Filesystem
  beispielsweise unterhalb des Unterverzeichnisses "**`/media/cdrom`**" so,
  als ob sie an dieser Stelle im root-Filesystem liegen würden.
- Aus Betriebssystem-Sicht handelt es sich trotzdem um *getrennt* verwaltete
  Filesysteme, nur die Darstellung der Filesysteme zum Benutzer hin
  ist zu einem einzigen Verzeichnisbaum verschmolzen.

Volume Manager

Volume Manager bilden bei Servern logisch

eine Ebene ***zwischen den Platten und den Filesystemen:***

Sie bilden ***"Physical Volumes" (= Platten)***
auf ***"Logical Volumes" (= Logische Laufwerke,***
wie virtuelle Partitionen) ab.

In jedem "Logical Volume" wird dann *ein* Filesystem eingerichtet.

Bezeichnung in Windows: *Dynamische Datenträger*.

Dabei bieten Volume Manager u.a. folgende Möglichkeiten:

- *Zusammenfassen mehrerer Platten-Partitionen* auf ein oder *mehreren Platten*
  zu einem einzigen Logical Volume
  (d.h. ein Logical Volume kann *viel größer* sein als die größte Platte im System!).
- *Zusammenfassen* mehrerer Platten oder Storage-Systeme
  zu einem *Software-RAID-Verbund* (zur Erhöhung der Datensicherheit)
  oder zu *"Striped Volumes"* (= RAID 0, zur Erhöhung der Geschwindigkeit).
- *Dynamische Verwaltung* des Plattenplatzes,
  Vergrößerung und Verkleinerung von Logical Volumes
  (z.B. bei Hinzufügen neuer Platten),
  nachträgliches Anlegen oder Löschen von Volumes auf bestehenden Platten.
- Eventuell *Verschlüsselung* von kompletten Volumes auf Block-Ebene.
- Eventuell *Snapshots* und/oder *Thin Provisioning* (siehe Spezialthemen).
- Eventuell *Multipath* (auf dasselbe Plattenlaufwerk bzw. dasselbe Storage-System
  wird *über mehrere Controller* zugegriffen).
- Einrichten sogenannter *Loop Devices*: Loop Devices sind *Logical Volumes*,
  deren Daten nicht auf einem "rohen" Datenträger,
  sondern *in einem (sehr großen) File eines anderen Filesystems* abgelegt werden.

  Genutzt werden Loop Devices einerseits zur *Verschlüsselung*
  (indem der darunterliegende File verschlüsselt wird) und andererseits,
  um z.B. *ISO-Images*, die in normalen Platten-Files gespeichert sind,
  wie eine echte CD zu mounten (oder z.B. um *Filesysteme einer VM*,
  die in einem File am Host liegen, im Host als Filesystem zu mounten).

RAID

RAID-Verbünde von Plattenlaufwerken (*"Redundant Arrays of inexpensive Disks"*)
dienen primär der *Erhöhung der Daten- bzw. Ausfall-Sicherheit*,
teilweise auch der *Geschwindigkeits-Erhöhung*:

- **RAID 0 ("Striping"):**

  RAID 0 teilt die Blöcke eines Logical Volumes in lauter gleich große "Stripes"
  (deshalb heißen RAID-0-Verbünde oft auch "striped Volumes"), z.B. je 64 KB,
  und *verteilt* diese Streifen *reihum auf die zur Verfügung stehenden Platten*
  (erste 64 KB ==› erste Platte, zweite 64 KB ==› zweite Platte, usw.).

RAID 0 bietet ***"Null" Sicherheit*** (weil die Daten nur einfach und *nicht redundant* gespeichert werden und bei Ausfall einer Platte weg sind), aber *erhöht die Geschwindigkeit*: Gleichzeitige Zugriffe verteilen sich auf mehrere Platten, und bei einzelnen langen, sequentiellen Zugriffen kann man von mehreren Platten parallel lesen und die Datenströme dann zu einem einzigen zusammenfügen (analog beim Schreiben).

- **RAID 1 ("Mirroring"):**

  RAID 1 speichert *alle Daten doppelt* auf *zwei gleich großen Laufwerken*, d.h. ein Laufwerk ist ein *exaktes Spiegelbild* der Daten des anderen Laufwerkes.

  RAID 1 erhöht primär die *Datensicherheit*, weil selbst bei Ausfall einer Platte noch alle Daten vorhanden sind (auf der anderen Platte), *verdoppelt* aber den *Plattenplatz-Bedarf*.

  Beim Schreiben ist RAID 1 Geschwindigkeits-neutral (beide Platten werden so beschrieben wie sonst eine einzelne Platte), beim Lesen verdoppelt es den Durchsatz (weil alle Daten wahlweise von einer der beiden Platten gelesen werden können und daher in Summe doppelt so viele Zugriffe in derselben Zeit abgearbeitet werden können).

- **RAID 10:**

  RAID 10 ist die *Kombination von RAID 0 und RAID 1*:
  Die Daten werden auf mehrere Platten *gestriped*,
  und diese Platten werden dann auf noch einmal so viele Platten *gespiegelt*.

- **RAID 5:**

  Ein RAID-5-Verbund besteht aus *n+1 Platten*.
  n Platten enthalten Nutzdaten, die verbleibende Platte enthält die *Prüfsumme*:
  Das i-te Bit der Prüfsumme ist die *Parität* der i-ten Bits aller Datenplatten.
  Der zusätzliche Platzbedarf ist daher 1/n, wesentlich besser als bei RAID 1.

  Um die Zugriffe bestmöglich zu verteilen und eine Überlastung der Parity-Platte zu vermeiden, werden Nutzdaten und Parity *wieder gestriped*:
  Für die ersten 64 KB enthält beispielsweise Platte 1 die Prüfsumme,
  für die nächsten 64 KB Platte 2, dann Platte 3 usw..

  Es darf *maximal eine Platte ausfallen*, denn der Inhalt einer Platte kann aus dem Inhalt aller anderen Platten rekonstruiert werden: Für Stripe-Gruppen, bei denen die Parity auf der kaputten Platte lag, ist ohnehin nur die Parity weg, und alle Nutzdaten sind noch da, womit man die Parity neu berechnen kann. Für Stripe-Gruppen, bei denen Nutzdaten auf der kaputten Platte lagen, lässt sich das i-te verlorene Bit dadurch berechnen, dass man die Parität aus den i-ten Bits aller verbleibenden Platten berechnet.

  Die *Lese-Geschwindigkeit* wird durch RAID 5 *erhöht*, aber die *Schreib-Geschwindigkeit verschlechtert sich deutlich*, auch gegenüber einer einzelnen Platte: Bei jedem Schreibzugriff müssen *zuerst die alten Daten und die alte Parität gelesen* werden. Dann wird daraus die *neue Parität berechnet*, und schließlich müssen *die neuen Daten und die neue Parität geschrieben* werden. Außerdem belastet die Paritätsberechnung die CPU (wenn sie nicht in spezieller Hardware erfolgt).

  Für die Performance besonders nachteilig ist der *Recovery-Modus* nach dem

Austausch einer defekten Platte: Da ja zum Wiederherstellen der Daten *alle Blöcke von allen Platten* gelesen werden müssen und auch über alle Blöcke die Parität neu berechnet werden muss, werden sowohl die Platten als auch die CPU so stark belastet, dass ein parallel laufender Produktivbetrieb beeinträchtigt werden kann.

- **RAID 6:**

  RAID 6 ist ähnlich RAID 5, aber mit *zwei Paritätsplatten* und daher auch mit doppelt so hohem Platz-Overhead. Dafür verkraftet RAID 6 den gleichzeitigen Ausfall zweier Platten pro RAID-6-Verbund.

Aufgaben eines Filesystems

- **Abstraktion der darunter liegenden Devices:**

  Filesysteme bieten zum Benutzer bzw. zu Programmen hin eine *einheitliche Sicht und Zugriffsweise auf Files und Directories*, egal, ob es sich um "echte" Platten, um SSD's, um direkt angebundene Flash-Chips (z.B. in Handys und Embedded Systems), um USB-Sticks, Speicherkarten usw., um CD- und DVD-Laufwerke oder um Netzwerk-Laufwerke bzw. Cloud-Storage handelt.

- **Platzverwaltung:**

  Filesysteme verwalten den Platz auf Block Devices:

  Sie führen ein *Verzeichnis freier Blöcke* sowie für jeden File ein Verzeichnis *der von den Daten des Files belegten Blöcke*, und sie teilen jedem File bei Bedarf neuen Platz zu.

  Optimierungskriterium ist u.a., den Inhalt eines Files auf *möglichst wenige räumlich getrennte Platten-Bereiche* aufzuteilen, d.h. so weit wie möglich in *sequentiell aufeinanderfolgenden Blöcken* zu speichern. Das beschleunigt sowohl Plattenzugriffe (weniger Kopfbewegungen) als auch Flash-Zugriffe (große zusammenhängende Bereiche lassen sich effizienter beschreiben bzw. löschen).

- **Verwaltung der File-Metadaten:**

  Zu diesen Metadaten gehören neben *Filenamen und Directories* unter anderem
  - die *Filegröße*,
  - die *Zugriffsrechte* auf Files und Directories,
  - ihre *Besitzer*,
  - und das *Datum* des Anlegens bzw. der letzten Änderung.

  Teilweise liegen diese Metadaten in beim Formatieren des Block Devices fix reservierten Bereichen (z.B. die Freispeicher-Liste), teilweise liegen sie in Bereichen, deren Platz wie der Platz normaler Files dynamisch verwaltet und zugeordnet wird (z.B. Directories).

- **Prüfung von Zugriffsrechten:**

  Die Filesystem-Logik schützt Files (bzw. generell die auf Block Devices gespeicherten Daten) vor unberechtigtem Zugriff.

Einschränkungen

Umgekehrt sind die Filesysteme auch für Einschränkungen in den Speicher-Fähigkeiten

eines Filesystems verantwortlich:

- Die **Länge von Filenamen** hängt vom Aufbau der internen Directory-Daten ab: Bei FAT in DOS waren es ursprünglich nur 8+3 Zeichen, später waren es dann 32 oder 64 Zeichen, heute ist die Grenze oft 255 Zeichen.
- Wenn die Filegröße in Bytes in den Directory-Einträgen z.B. in einem 32 Bit Int gespeichert wird, so ist die **maximale Größe eines Files** auf 2 bzw. 4 GB *beschränkt*.
- Wenn für Blocknummern 32 Bit Int's verwendet werden, sind maximal 2^{32} Blöcke möglich. Bei 1 KB Blöcken beschränkt das die **maximale Filesystem-Größe** auf 4 TB.

  FAT16 verwendete nur 16 Bit Nummern für Cluster, mit max. 64 KB Cluster-Größe waren damit max. 4 GB große Filesysteme möglich. FAT32 ist auf 28 Bit beschränkt, und Cluster können maximal 32 KB sein, womit sich höchstens 8 TB Filesystem-Größe ergeben.
- Manchmal gibt es auch Beschränkungen in der **Directory-Größe** (z.B. FAT: Das Wurzel-Verzeichnisses hat fix 512 Einträge) oder in der **File-Anzahl** (die Anzahl der "Inodes" wird bei vielen Unix/Linux-Filesystemen beim Formatieren fix festgelegt).

Zugriffsrechte

Hier gab es im Laufe der Zeit große Weiterentwicklungen:

- Unter DOS (**FAT**) gibt es nur *vier Bits*: **Readonly, Hidden, System, Archive** (das Archive-Bit diente dazu, Backup-Programmen anzuzeigen, welche Files seit dem letzten Backup geändert worden sind)
- Das klassische **Unix**- und **Linux**-Rechtesystem besteht aus *3 * 3 Bits für die Rechte*, außerdem wurde zu jedem File ihr *Besitzer* und eine *Benutzergruppe* gespeichert: **Lesen, Schreiben und Ausführen** (bzw. bei Directories: **Hineinwechseln**) jeweils für den *Besitzer* des Files, für die *Mitglieder der Gruppe*, der der File gehört, und für *alle anderen* Benutzer.

  Daneben gibt es noch das **Setuid**- und das **Setgid-Bit** (das einen ausführbaren File *mit den Rechten des File-Besitzers bzw. der File-Gruppe* laufen lässt und nicht mit den Rechten und der Gruppe von dem Benutzer, der ihn gestartet hat) und das **Sticky-Bit**, das in für alle schreibbaren Directories das *Löschen fremder Files verhindert*.
- Heute können praktisch alle Filesysteme (sowohl unter Windows als auch unter Linux) **ACL's ("Access Control Lists")**:

  Jeder File hat eine *beliebig lange Liste von Rechte-Einträgen*, und jeder Eintrag in dieser Rechte-Liste *gibt oder nimmt* einem bestimmten Benutzer oder einer bestimmten Benutzer-Gruppe *("Wer?")* bestimmte Rechte *("Was?")* (wie Lesen, Schreiben, Löschen, ...).

Beispiel Platzverwaltung: FAT

Die Verwaltung der zu einem File gehörenden Datenblöcke sowie der freien Blöcke erfolgt je nach Filesystem sehr unterschiedlich.

In FAT-Filesystemen werden alle Block-Zuordnungen in der **FAT ("File Allocation Table")** gespeichert. Diese ist ein *großes Array von Blocknummern* mit so vielen Einträgen,

wie das Filesystem logische Blöcke hat (wobei die "echten" Blöcke zu 512 Bytes oder 4 KB zu größeren logischen Blöcken ("Clustern") von bis zu 64 KB zusammengefasst werden).

Der i-te Eintrag der FAT gehört zum i-ten Block im Filesystem und enthält die *Blocknummer des nächsten Blockes desselben Files* (wenn der i-te Block zu einem File gehört) bzw. *des nächsten Blockes der Freiliste* (wenn der i-te Block zur Freiliste gehört) bzw. *eine Ende-Markierung* (wenn der i-te Block der *letzte Block* eines Files bzw. der Freiliste ist). *Kaputte Blöcke* usw. werden ebenfalls in der FAT markiert.

Die zu einem File gehörenden Blöcke sind also als *verkettete Liste* in der FAT gespeichert, ebenso die freien Blöcke.
Die Nummer des *ersten Blockes* eines Files steht *in ihrem Directory-Eintrag*.

Beispiel Platzverwaltung: "Klassische" Unix- und Linux-Filesysteme

Unter Unix/Linux ist die Directory-Information *zweigeteilt*:

- In den ***Directories*** wird nur der *Filename und eine Nummer* gespeichert:
  Die ***Inode-Nummer***.
- Die *eigentliche File-Information* (Größe, Datum, Rechte, zugeordnete Blöcke, ...), aber ohne Name und ohne Zuordnung zu einem Verzeichnis, liegt im ***Inode***, einem *Eintrag der Inode-Table*, wobei die *Inode-Nummer* aus dem Directory der *Index* des zum File gehörigen Inodes in der Inode-Table ist.

Von der Speicherung und Platzverwaltung her werden *Directories wie normale Files* behandelt, *Inodes* hingegen liegen in einem *speziellen, beim Formatieren reservierten Teil* der des Filesystems (die Anzahl der Directory-Einträge ist daher beliebig, aber die maximale Anzahl der Inodes wird beim Formatieren festgelegt).

Durch dieses Inode-Konzept ergibt sich auch,
dass es unter Unix/Linux *zwei Arten von Verweisen* gibt:

- Der normale Verweis heißt unter Linux ***"symbolischer Link"***
  (***"Symlink"***, manchmal auch Softlink genannt):

  Im Inode eines Symlinks wird der *Filename des Ziel-Files*
  (also des Files, auf den der Symlink zeigt) gespeichert
  (z.B. dort, wo normalerweise die Nummern der Datenblöcke gespeichert werden).

  Damit ist bei Symlinks *klar zwischen Original und Link darauf unterschieden*.

  Die Probleme von Symlinks sind Folgende:

  - Der *Originalfile*, auf den ein Symlink zeigt, kann *trotz Symlink gelöscht, verschoben oder umbenannt* werden: Weder "weiß" das Original, dass ein Symlink auf ihn zeigt, noch bekommt der Symlink mit, wenn es kein Original mit diesem Namen mehr gibt:
    Der Symlink zeigt dann "ins Leere", jeder Zugriff liefert einen Fehler.
  - Symlinks können *zyklisch aufeinander weisen*,
    ohne dass man jemals zu einem echten File kommt.
  - Symlinks können relative oder absolute Pfade enthalten.

    Relative Symlinks funktionieren nicht mehr, wenn man sie in ein anderes Directory kopiert, aber das Ziel nicht mitwandert,

und absolute Pfade sind schlecht, wenn man Symlink und Ziel gemeinsam verschiebt (relative Links funktionieren in diesem Fall weiter).

- Alternativ kann man ***"Hardlinks"*** anlegen. Das bedeutet, dass einfach *mehrere Directory-Einträge* ***dieselbe Inode-Nummer*** *enthalten* und daher direkt auf denselben File verweisen, d.h. *ein und derselbe File wird in mehreren Directories unter mehreren Namen sichtbar* (jeder Inode hat daher unter Unix/Linux einen "Link Count": Die Anzahl der auf ihn verweisenden Directory-Einträge).

  Für *Directories* ist das der Normalfall, jedes Directory hat *mindestens Link Count 2:* Einmal für den Eintrag im übergeordneten Directory, der zu dem Directory führt, und einmal für den Eintrag '**.**' im Directory selbst (und je ein weiteres Mal für den '**..**'-Eintrag in jedem seiner Unterverzeichnisse).

  Bei Hardlinks gibt es *kein Original und keinen Verweis*: Alle Directory-Einträge für denselben File (Inode) sind *gleichberechtigt und technisch nicht unterscheidbar*. Der File (d.h. Daten und Inode-Eintrag) wird erst dann *gelöscht*, wenn sein *Link Count auf 0* geht, d.h. wenn es keinen Directory-Eintrag für ihn mehr gibt.

  Auch hier gibt es Probleme:

  - Hardlinks sind *nur innerhalb eines Filesystems* möglich.
  - Hardlinks müssen von *Backup-Programmen* speziell unterstützt werden, sonst werden beim Restore zwei getrennte Files daraus.
  - Viele Programme (z.B. Editoren) ändern einen File nicht, indem sie den Inhalt des bestehenden Files aktualisieren, sondern indem sie beispielsweise einen *neuen File anlegen*, dann den alten File umbenennen und *dem neuen File den alten Namen geben*, und schließlich den umbenannten *alten File löschen*.

    Dieses Vorgehen *bricht aber Hardlinks* auf den File: Sie zeigen nachher *immer noch auf die alte Version* und nicht auf die neue, denn die neue Version hat zwar *denselben Namen* (der für Hardlinks aber nicht relevant ist), aber eine neue, geänderte Inode-Nummer, und ist daher aus Hardlink-Sicht ein anderer File.

Die Speicherung der ***Nummern der Datenblöcke***, die den *Inhalt eines Files* enthalten, erfolgt in klassischen Unix-Filesystemen ganz anders als in FAT-Filesystemen:

- *Im Inode selbst* ist Platz für eine *kleine Anzahl von Block-Nummern* (je nach Filesystem 12 bis 16, wir nehmen einmal an, es sind 13).
- Die *ersten 10* dieser Nummern verweisen *direkt auf die ersten 10 Blöcke des Files*. Bei einer Blocksize von 512 Bytes lassen sich also die ersten 5 KB des Files sofort und direkt ansprechen, und solange der File kürzer als 5 KB ist, wird keine Platzbelegungs-Information außerhalb des Inodes benötigt.
- Reichen diese 10 Blocknummern nicht, zeigt die *elfte Blocknummer* im Inode auf einen Block, in dem *128 weitere Blocknummern* von Datenblöcken gespeichert sind (auf den *"einfach indirekten Block"*). Das reicht für weitere 64 KB Daten.
- Reicht das auch nicht, zeigt die *zwölfte Blocknummer* im Inode auf einen Block (den *"zweifach indirekten Block"*), der die Blocknummern von *128 weiteren einfach indirekten Blöcken* enthält. Das reicht für die nächsten 128*64 KB, also 8 MB Daten.

- Und wenn das immer noch zu wenig ist, zeigt die *letzte Blocknummer* im Inode auf einen *"dreifach indirekten Block"*, der die Nummern von 128 zweifach indirekten Blöcken enthält, womit sich weitere 128*8 MB Plattenplatz zuordnen lassen, also 1 GB. Größere Files waren in den ursprünglichen Unix-Filesystemen (vor über 40 Jahren!) nicht möglich.
- Bei heutigen Filesystemen sind die Blöcke oft größer (z.B. 4 KB). Das wirkt sich gleich zweifach auf die maximale Filegröße aus: Einerseits weil ein Block mehr Nutzdaten enthält, und andererseits, weil ein indirekter Block mehr Blocknummern enthält und daher z.B. auf 1024 statt 128 weitere Blöcke verweist.
- Insgesamt ist der *Zugriff auf kleine Files* bzw. auf *das vordere Ende großer Files* also *schnell*. Je größer ein File ist bzw. je weiter man nach hinten liest, umso mehr Plattenzugriffe sind nötig, um an die Nummern der Datenblöcke zu kommen, und umso größer ist der Platz-Overhead für die Blocknummern.
- Die *freien Blöcke* werden unabhängig davon gespeichert, z.B. in einem Bit-Array mit 1 Bit pro Block.
- In modernen Unix-Filesystemen (XFS, BtrFS, ZFS, ...) erfolgt die Speicherung der Datenblock-Nummern zum Teil anders (z.B. in *B+-Bäumen* ähnlich wie bei Datenbanken), und ein einzelner Eintrag enthält meist nicht mehr einen einzelnen Block, sondern einen *Extent* (siehe unten: Anfangsblock plus Anzahl der Blöcke). Das Konzept, die ersten paar Einträge gleich direkt im Inode zu speichern, blieb aber in vielen Filesystemen erhalten.

Neben Files, Directories und Soft Links können Directory-Einträge unter Unix/Linux auch noch folgende Dinge repräsentieren:

- **Unix Sockets** (wie *"lokale Netzverbindungen"*) und **FIFO's** (benannte *Pipes*, zur Kommunikation zwischen zwei Programmen).
- **Char-Devices** und **Block-Devices** (siehe oben bei I/O-Devices: Jedes I/O-Gerät und jedes Logical Volume wird unter Linux durch einen Namen und Directory-Eintrag unter dem Directory **`/dev`** repräsentiert).

Des Weiteren gibt es unter Linux vom Kernel "vorgetäuschte" *Pseudo-Filesysteme* (vor allem **`/dev`**, **`/sys`** und **`/proc`**). Der Benutzer sieht unter diesen Directories ganz normale Verzeichnisbäume, die zahlreiche Files enthalten.

Diese Files und Directories existieren aber nirgends real (nicht einmal im RAM): Sie werden beim Zugriff darauf *dynamisch vom Kernel erzeugt* und mit Inhalten versorgt. Der Kernel stellt auf diese Weise dem "Userland" laufend *aktuelle Informationen* zur Hardware, zum Kernel selbst, zu laufenden Prozessen, zur Systemauslastung, zum Netzwerk und zu vielen anderen Bereichen des Systems zur Verfügung (der "Task Manager" unter Linux bezieht die angezeigten Informationen beispielsweise ausschließlich aus diesen Pseudo-Filesystemen).

Manche dieser Files sind auch schreibbar und *setzen diverse Kernel-Parameter* (ändern beispielsweise die Stromspar-Modi des Prozessors oder der Grafikkarte, oder setzen irgendwelche Kernel- oder Netzwerk-Parameter).

Formatierung und File System Check

Für jeden vom Betriebssystem unterstützten Filesystem-Typ bringt das System normalerweise *zwei Hilfsprogramme* mit:

- **Formatierung:**

  Dieses Programm *initialisiert* ein Block Device mit einem *"leeren" Filesystem*: Unter anderem reserviert und initialisiert es die *fixen Metadaten-Bereiche* des Filesystems (z.B. die Freiliste) und legt ein *Wurzelverzeichnis* an.

  Typischerweise werden dabei nur die soeben genannten Bereiche beschrieben bzw. zurückgesetzt. Der *Rest* des Block Devices (also alles außerhalb der reservierten Bereiche) bleibt *unverändert*, d.h. vorher vorhandene File-Daten und auch Directories "überleben" die Formatierung normalerweise: Sie sind nachher zwar unsichtbar, aber mit entsprechenden Tools oft weitgehend rekonstruierbar (es ist ja auf Grund des leeren Wurzelverzeichnisses nur ihr Directory-Eintrag nicht mehr erreichbar, ihr Daten-Bereich wurde zwar in die Freiliste eingetragen, aber nicht wirklich gelöscht oder überschrieben).

  Bei vielen Filesystemen ist es (nachdem das "Logical Volume" über den Volume Manager vergrößert bzw. um zusätzliche Platten erweitert wurde) auch möglich, bestehende *Filesysteme ohne Informationsverlust zu vergrößern*, manche Filesysteme erlauben auch eine *Verkleinerung*.

- **File System Check:**

  Dieses Programm kommt zum Einsatz, wenn das System *"hart" abgeschaltet* oder das Speichermedium *im laufenden Betrieb entfernt* wurde, d.h. wenn das Betriebsystem nicht mehr die gesamte im RAM gepufferte Filesystem-Information auf die Platte zurückschreiben konnte und die Filesystem-Metadaten daher *inkonsistent* sind.

  Erkannt wird die Notwendigkeit eines File System Checks daran, dass beim Booten bzw. beim Mounten des Filesystems dessen ***"Dirty-Bit"*** gesetzt ist (siehe unten).

  Nach dem hardware-mäßigen Ausfall einzelner Platten-Sektoren kann das File-System-Check-Programm ebenfalls versuchen, zu retten, was noch zu retten ist.

  Es versucht, *wieder einen konsistenen Filesystem-Metadaten-Zustand herzustellen* (notfalls unter Verlust von Files oder File-Inhalten bzw. mit dem Risiko der Zuordnung falscher File-Inhalte). Im günstigsten Fall kann es *"verlorene" File-Inhalte durch Anlagen neuer Directory-Einträge wieder zugreifbar* machen.

  Es prüft unter anderem:

  - *Konsistenz der Platzverwaltung:*
    - Gibt es Blöcke, die sowohl in der Freiliste stehen als auch einem File zugeordnet sind?
    - Gibt es Blöcke, die mehreren Files zugeordnet sind?
    - Gibt es Blöcke, die weder in der Freiliste stehen noch belegt sind?
  - *Konsistenz der Platzzuordnung:* Entspricht die *Gesamtgröße* des zugeordneten *Platzes* eines Files der im Directory eingetragenen *Filegröße*?
  - *Konsistenz der Directories:*
    - Stimmen die Einträge `.` und `..` in jedem Directory?

- Gibt es Directories oder Directory-Einträge,
  die von der Wurzel aus *nicht erreichbar* sind?
- Gibt es *Zyklen* im Directory-Baum?

- *Unter Linux: Inode-Konsistenz:*
  - Stimmt der Hard-Link-Count?
  - Gibt es gültige Inodes, auf den kein Directory-Eintrag mehr verweist?
  - Gibt es als gelöscht markierte Inodes,
    auf die noch ein Directory-Eintrag verweist?

  Falls ein Filesystem *zusätzliche Indices zum raschen Suchen* in großen Directories anlegt, werden diese bei der Prüfung meist frisch aufgebaut.
  Auch die *Quota-Daten* (siehe unten) müssen meist neu berechnet werden.

  Optional kann ein File System Check meist auch ein *"Probelesen"* (oder sogar ein Lesen, Zurückschreiben und nochmaliges Lesen) aller Blöcke im Filesystem durchführen, d.h. einen *Test auf physisch defekte Blöcke*.

Darüber hinaus gab es früher auch *"Reorganisationsprogramme" ("Defragmentierer")*, die die Daten innerhalb eines Filesystems so umkopierten, dass jeder File einen einzigen zusammenhängenden Plattenbereich belegte und auch der Freiplatz danach zusammenhängend war, um neue Files effizient anlegen zu können.

Diese Programme bieten bei modernen Filesystemen aber keinen wesentlichen Vorteil mehr und sind riskant (Datenverlust bei Unterbrechung).
Manche moderene Filesysteme haben stattdessen eine ständig im Hintergrund arbeitende, fix eingebaute Defragmentierung.

"Special Features"

Darüber hinaus bieten moderne Filesysteme heute eine Vielzahl zusätzlicher Features:

- **Wählbare Blockgröße:**

  Bei vielen Filesystemen ist die Einheit der Platzverwaltung nicht zwingend gleich der physischen Blockgröße des darunterliegenden Block Device, sondern kann frei festgelegt werden (typischerweise in Zweierpotenzen der physischen Blockgröße).

  Für Filesysteme mit *vielen kleinen Files* nimmt man *eher kleine Blöcke*
  (das *reduziert den verlorenen Platz* im letzten Block jedes Files),
  bei Filesystemen mit *hauptsächlich großen Files* wählt man *große Blöcke*
  (das reduziert den Platzbedarf für die Block-Verwaltung, den Zeitbedarf für die Allokation, und die Fragmentierung).

- **Dirty-Bit:**

  Das Dirty-Bit zeigt an, dass das *Filesystem in Verwendung* ist, d.h. das sein Inhalt verändert wurde bzw. gerade verändert wird und momentan *möglicherweise kein konsistenter Stand auf dem Medium* gespeichert ist.

  Das Dirty-Bit wird vom Betriebssystem *automatisch vor dem ersten Schreibzugriff* auf ein Filesystem *gesetzt* und beim *ordnungsgemäßen Herunterfahren* des Systems oder beim *ordnungsgemäßen Auswerfen* des Mediums (d.h. nachdem garantiert alle Daten und Metadaten vollständig gespeichert wurden) *zurückgesetzt*.

  Ist das Dirty-Bit *beim Mounten eines Filesystems gesetzt*, muss

vor der Verwendung des Filesystems ein *File System Check* gemacht werden. Ist es gelöscht, kann das Filesystem sofort verwendet werden.

- **Multi-Mount-Protection:**

  In großen Rechenzentren sind *Storage-Systeme* meist über Netzwerke *mit mehreren Servern verbunden*. Dadurch können dieselben Platten bei Ausfall eines Servers sofort einem anderen Server zugeordnet werden, und dieser kann den Betrieb fortsetzen. Eine Ebene höher gilt dasselbe für Platten-Images, die VM's zugeordnet werden.

  Normale Filesysteme (solange sie nicht speziell als Cluster-Filesysteme entworfen sind) werden aber meist *irreparabel inkonsistent und verlieren Daten*, wenn *zwei Rechner* versuchen, unabhängig voneinander *gleichzeitig schreibend* auf dasselbe Filesystem zuzugreifen. Moderne Filesysteme enthalten daher Mechanismen, die versuchen, einen *gleichzeitigen Mount* desselben Logical Volumes *auf mehreren Rechnern* zu erkennen und *zu verhindern*.

- **Locking:**

  Das Filesystem verhindert, das Files gelöscht werden, die gerade verwendet oder als Programm ausgeführt werden (unter Linux können auch verwendete Files gelöscht werden. Es wird allerdings nur der Directory-Eintrag gelöscht, d.h. der File wird unsichtbar, aber schon aktive Zugriffe funktionieren weiterhin. Der File-Inhalt selbst wird erst gelöscht, nachdem das letzte zugreifende Programm den Zugriff beendet hat).

  Weiters führt das Filesystem Buch, welche Anwendung gerade welche Bereiche eines Files benutzt, und versucht, konkurrierende Zugriffe zu verhindern oder zu sequentialisieren. Es erlaubt Programmen auch, mittels *spezieller Systemaufrufe* den *ganzen File oder Bereiche davon explizit für den gleichzeitigen Zugriff durch andere Programme* zu sperren.

- **Datensicherheit und redundante Speicherung:**

  Fast alle Filesysteme speichern *wesentliche Informationen mehrfach* (und an weit auseinanderliegenden Stellen) auf der Platte, um besser gegen physische Lesefehler geschützt zu sein (z.B. unter Linux den *"Superblock"* mit grundlegenden Informationen über das Filesystem als Ganzes, oder bei FAT-Filesystemen die *gesamte FAT*).

  Moderne Filesysteme berechnen sogar laufend *für jeden Block Metadaten* (und ev. auch für jeden Block Nutzdaten) eine *Prüfsumme*, die mitgespeichert wird und hilft, beim Lesen "umgefallene" Bits zu erkennen und ev. zu reparieren (zusätzlich zur Hardware-Fehlerkorrektur der Platte, die mit einer Wahrscheinlichkeit im Bereich 2^{-15} bis 2^{-18} Fehler übersiehen könnte).

- **Konsistenz-Garantien:**

  Grundsätzlich können sowohl das Betriebssystem als auch die Plattenlaufwerke selbst *zu schreibende Daten* puffern und zu einem beliebigen Zeitpunkt und vor allem *in beliebiger, umsortierter Reihenfolge* tatsächlich auf die Platte speichern.

  Jedes Filesystem kann aber die *Schreib-Reihenfolge* soweit *beeinflussen*, dass gewisse *Konsistenz-Garantien* gelten, selbst wenn Hardware-Fehler auftreten oder das System hart beendet wird (z.B. dass Files beim Umbenennen

nicht verloren gehen oder beim Ersetzen eines Files stets entweder der alte oder der neue Fileinhalt vorhanden ist).

- **Quotas:**

  Viele Filesysteme erlauben es, *Platz-Limits für alle Files eines Benutzers*, einer Benutzergruppe oder eines Unterverzeichnis-Baumes festzulegen, damit niemand einen unfairen Anteil der Platte belegen oder die Platte komplett anfüllen kann. Dieses Platzlimit eines Benutzers usw. heißt Quota.

  Selbst bei Filesystemen ohne Quotas werden normale Benutzer normalerweise am Schreiben gehindert, sobald das Filesystem zu 95 % voll ist:
  Nur Programme mit *Administrator-Privilegien* dürfen die *letzten 5 % Platz nutzen*.

- **Extent-basierte Platzverwaltung:**

  Sowohl FAT-Filesysteme als auch klassische Unix-Filesysteme speichern den Platz, der einem File zugeordnet wurde, als *Folge einzelner Blöcke*.
  Dies ist aus mehreren Gründen *nicht sonderlich effizient*:

  - Bei sehr großen Files müssen *Millionen von Block-Nummern* gespeichert werden.
  - Da ja das Ziel ist, jedem File *möglichst sequentiell aufeinanderfolgende Blöcke* zuzuordnen, enthalten diese Block-Listen oft *große Bereiche aufeinanderfolgender Nummern*.

  Deshalb ist man bei den meisten modernen Filesystemen dazu übergegangen, die Platzbelegungs-Information *nicht mehr als einzelne Blocknummern* zu speichern, sondern in Form von *Extents*, wobei ein Extent einen *zusammenhängenden Block-Bereich* darstellt: Gespeichert wird dann pro File eine Liste von Extents, wobei für jeden Extent die *Anfangs-Blocknummer* und die *Anzahl der Blöcke* gespeichert wird.

- **Change Notification Service:**

  Ein Programm (z.B. ein File-Explorer) kann dem Betriebssystem mitteilen, dass es von allen *Änderungen*, die *andere Programme* in einem bestimmten File oder Directory vornehmen, *sofort informiert* werden möchte (z.B. um seine eigene Anzeige entsprechend zu aktualisieren).

  Bei Filesystem-Änderungen schickt die Filesystem-Logik im Betriebssystem dann an alle registrierten Programme entsprechende Signale.

- **Charset Translation:**

  Bei Filesystem-Metadaten (insbesondere *File- und Directory-Namen*) ergibt sich dasselbe Problem wie bei den Inhalten von Textfiles:
  - Was passiert, wenn das System beim Lesen eines Filesystems mit einem *anderen Zeichensatz* läuft als beim Schreiben?
  - Was passiert, wenn neu anzulegende Filenamen usw. Zeichen enthalten, die im Zeichensatz des Filesystems *gar nicht dargestellt* werden können?

  Manche Filesysteme können in diesem Fall automatisch eine *Umwandlung* bzw. eine spezielle *Codierung von Sonderzeichen* vornehmen.

- **Secure Overwrite:**

  Normalerweise werden beim *Löschen von Files* nur deren bisher belegte *Blöcke in die Freiliste eingetragen*: Die Blöcke selbst werden *nicht überschrieben*, die Daten

sind daher bei direktem Zugriff auf das Logical Volume *nach wie vor auslesbar*.

Manche Filesysteme bieten ein *"echtes" Löschen* an, bei dem z*u löschende Daten und Metadaten* wirklich sofort auf dem Medium *physisch überschrieben* werden (bei SSD's geht dieser Mechanismus allerdings ins Leere, weil eine SSD einen Block bei jedem Schreiben auf andere Speicherzellen abbildet, es werden daher andere Bereiche überschrieben als die mit den ursprünglichen Daten).

- **File-Versionen:**

  Manche Filesysteme erlauben es, zu einem File nicht nur den aktuellen Stand, sondern *auch ältere Versionen* zu speichern (und mit einer entsprechenden Versionsnummern-Angabe hinter dem Filenamen auch jederzeit darauf zuzugreifen).

- **Extended Attributes:**

  Moderne Filesysteme erlauben es, *zu jedem Directory-Eintrag* beliebig viele Einträge der Form ***"Name=Wert"*** zu speichern. Primär dienen diese Einträge zum Speichern der *ACL's für die Zugriffsrechte*, aber es lassen sich auch *beliebige Informationen* für ganz andere Zwecke dort ablegen (das gehärtete GrSec-Linux speichert dort beispielsweise u.a., welche Speicher-Zugriffsschutz-Mechanismen für ein ausführbares Programm beim Start des Programms anzuwenden sind).

- **Mehrfache Streams oder Forks**
  (das sind verschiedene Bezeichnungen für dasselbe Feature):

  Normalerweise wird zu jedem Filenamen *eine Folge von Bytes als Nutzdaten* gespeichert. Manche Filesysteme erlauben es, neben diesem "primären" Stream, der beim normalen Zugriff auf den File gelesen bzw. geschrieben wird, noch *weitere Daten-Streams unter demselben Filenamen* abzulegen (z.B. Metadaten zu einer Bilddatei), die alternativ gelesen oder geschrieben werden können und automatisch so wie die primären Daten mit dem File kopiert, gesichert, gelöscht usw. werden.

- **Discard und Thin Provisioning:**

  Bei normalen Plattenlaufwerken wurde die *Platte nicht informiert*, wenn *Daten gelöscht* wurden: Die Blöcke wurden zwar in den Filesystem-Metadaten als "frei" eingetragen, aber die Platte bekam davon nichts mit.
  Aus Sicht der Platte war es daher nicht möglich, festzustellen, welche Blöcke gültige Daten enthielten und welche nicht.

  Heute ist dieses Wissen aber in vielen Fällen nötig:

  - **SSD's** können das *Flash um Vieles effizienter verwalten* (vor allem schneller und mit viel weniger Lebensdauer-schädlichen Löschzyklen beschreiben), wenn sie genau wissen, welche Blöcke noch *gültig* sind (*erhalten* werden müssen) und welche *gelöscht* bzw. *für andere Daten verwendet* werden können.

  - **Storage-Systeme und Cloud-Speicher** reservieren für ein Logical Volume nicht mehr gleich beim Anlegen die geforderte Speichermenge, sondern teilen dem Volume *erst dann Block für Block echten Speicher* zu, *wenn wirklich Daten geschrieben* werden. Sie speichern also nur die *"wirklich benutzten" Blöcke*, und beim Backup werden auch nur diese gesichert. Werden *Blöcke gelöscht*, so werden diese *dem Logical Volume wieder entzogen*

(und möglicherweise *anderen* Logical Volumes zugeordnet).

Man nennt dieses Konzept ***"Thin Provisioning"***.

- Vor allem in den *Images* von Logical Volumes, die zu *virtuellen Maschinen* gehören, werden meist nur die Blöcke wirklich physisch gespeichert, die in der VM schon einmal tatsächlich beschrieben wurden, und die Images werden auch wieder "geschrumpft", wenn Daten darauf gelöscht werden.

Damit der Storage Provider (die SSD bzw. die Cloud-Speicherverwaltung) weiß, welche Blöcke frei und welche belegt sind, schicken moderne Filesysteme auf Wunsch für jeden Block, der *im Filesystem freigegeben* wurde und daher *am Speichermedium verworfen* werden kann, einen ***"Discard"-Befehl*** mit der Blocknummer an das darunter liegende Logical Volume.

- **Sparse Files:**

  Sparse Files sind *Files mit Löchern*, d.h. Files, bei denen *nicht der gesamte Inhalt wirklich Platz auf der Platte* belegt: Bereiche der File-Daten, die zwar angelegt, aber noch nie geschrieben wurden, werden ebenso ausgelassen wie Blöcke, die mit lauter Null-Bytes beschrieben wurden oder die von der Anwendung bewusst als "Loch" markiert wurden.

  Löcher werden mit einer speziellen Kennung in der Liste der vom File belegten Blöcke eingetragen und *beim Lesen automatisch als Block von Null-Bytes* gelesen.

- **Journalling:**

  Ein *Filesystem-Check* auf großen, "klassischen" Filesystemen kann Minuten oder Stunden dauern. Man nimmt daher eine Verlangsamung des laufenden Betriebs in Kauf, um die *Recovery-Zeit nach Abstürzen zu reduzieren*:

  Jede geplante Änderung der Metadaten wird *zuerst* in das ***Journal***, einem reservierten Bereich im Filesystem (oder ein separates Block Device), geschrieben. Dabei werden logisch zusammengehörige Änderungen zu *"Transaktionen"* zusammengefasst: Am *Ende jeder einzelnen Transaktion* ist das Filesystem wieder *in einem konsistenen Zustand*. Die einzelnen Transaktionen werden *strikt in logisch aufeinanderfolgender Reihenfolge* ins Journal geschrieben (was zuerst gemacht werden soll, steht zuerst im Journal).

  Erst nachdem die komplette Transaktion einer Änderung *physisch im Journal gespeichert* ist, wird ihre *Durchführung im Filesystem selbst* freigegeben (d.h. das Filesystem eilt seinem Journal *nie voraus*). Und erst wenn die Änderung im Filesystem tatsächlich auf dem Speichermedium durchgeführt ist, wird sie im Journal gelöscht (bzw. von neuen Einträgen zyklisch überschrieben).

  Nach einem harten Abschalten reicht es, *alle im Journal verzeichneten kompletten Transaktionen nochmals durchzuführen* (was in Zehntelsekunden geht), um einen *konsistenten Zustand* des Filesystems herzustellen. Ein Filesystem-Check ist nicht mehr nötig. Steht eine Transaktion nur unvollständig im Journal, wird sie beim Recovery ignoriert: Sie wurde im Filesystem selbst sicher noch nicht begonnen.

  Bei besonders hohen Datensicherheits- bzw. Konsistenz-Anforderungen werden nicht nur alle Metadaten-Updates, sondern auch *alle Nutzdaten-Updates* zuerst im Journal eingetragen. Sonst kann es passieren, dass Änderungen der File-Inhalte den Metadaten-Änderungen vorauseilen oder nachhinken.

- **Verschlüsselung:**

  Manche Filesysteme können den *Inhalt einzelner Files* (dann sind die *Metadaten* weiterhin *im Klartext* am Medium gespeichert) oder den *gesamten Filesystem-Inhalt* (einschließlich Directories und Metadaten) *"on the fly"* automatisch verschlüsseln und entschlüsseln.

- **Komprimierung:**

  Manche Filesysteme können File-Inhalte (und ev. auch Metadaten) *"on the fly"* transparent komprimieren und beim Lesen wieder dekomprimieren.

  Ganz unabhängig davon gibt es *Readonly-Filesysteme* (z.B. zur Speicherung im Flash von Embedded Systems oder auf CD's), die *beim Erstellen einmal als Ganzes* möglichst hoch komprimiert werden, um Hardware-Platz zu sparen, und dann im komprimierten Zustand verwendet und gelesen werden können.

- **Copy-on-write:**

  Bei Filesystemen, die Copy-on-write unterstützen, werden beim *Kopieren von Files* nicht die Daten selbst kopiert, sondern *nur die Directory-Einträge*:
  Beide Einträge verweisen auf *dieselben Datenblöcke*.
  Erst wenn eine der beiden Kopien *geändert* wird, werden diejenigen Blöcke, die von der Änderung betroffen sind, *tatsächlich kopiert*, und jeder der beiden Files bekommt seine eigene Version der betroffenen Blöcke.

- **Deduplizierung:**

  Deduplizierung ist die aggressive Form von Copy-on-Write: Bei *jedem Datenblock*, der in das Filesystem *geschrieben* werden soll, wird geprüft, ob es irgendwo (egal bei welchem File!) *schon einen Datenblock mit demselben Inhalt gibt* (zuerst durch Berechnen eines Hashwertes und einer Prüfung, ob schon Blöcke mit diesem Hashwert registriert sind, und erst bei gleichem Hashwert durch tatsächlichen Vergleich der neuen Daten mit allen Blöcken mit diesem Hash).

  Ist dies der Fall, so wird *kein neuer Block* reserviert und gespeichert, sondern es wird stattdessen in der Blockliste des geänderten Files ein *Verweis auf den schon bestehenden Block* eines anderen Files eingetragen.

  Dieser gemeinsame Block wird bei beiden Files als *"Copy-on-write"* markiert: Beschreibt einer der beiden diesen Block, wird die Verknüpfung aufgehoben und durch zwei getrennte Blöcke ersetzt (außer für die neuen Daten findet sich wieder ein "Deduplizierungspartner").

  Für normale PC's würde sich dieser Aufwand kaum lohnen, aber in *Rechenzentren* kann er sehr effektiv sein:

  - *VM-Image-Files* tendieren dazu, in weiten Teilen idente Daten zu enthalten (weil ja in allen Images *weitgehend dieselbe Software* installiert ist).
    Bei Installationen mit vielen verschiedenen VM's (z.B. bei Cloud-Providern!) ist die Ersparnis durch Deduplizierung am Host-System daher meist recht hoch.
  - Auch auf *Filesystemen*, die die *Daten vieler Benutzer derselben Firma* speichern, gibt es meist einen beträchtlichen Anteil identer File-Inhalte. Gerade in Software-Entwicklungs-Firmen kommt es häufig vor, dass jeder Entwickler in seinen Directories einige Megabytes identer Sourcen, Libraries usw. liegen hat.

- Besonders heiße Kandidaten für Deduplikation sind *klassische Mail-Server*: Einerseits liegt dieselbe Mail als File einmal beim Absender und einmal bei jedem Empfänger. Andererseits ist es in Firmen üblich, dass Mail-Attachments oft über zahlreiche Mails an eine große Anzahl von Mitarbeitern verschickt werden: In diesem Fall sind zwar die eigentlichen Mail-Files verschieden, aber sie verweisen auf inhaltlich idente Attachment-Files.

- **Snapshots:**

  Snapshots *frieren* (im Optimalfall in Sekundenbruchteilen) einen *konsistenten Zustand des Filesystems zu einem bestimmten Zeitpunkt* ein. Sie stellen diesen eingefrorenen Zustand des Filesystems dann *als zweites, separates Filesystem* zur Verfügung, das unabhängig vom primären Filesystem gemountet werden kann (im einfachsten Fall nur lesend, bei manchen Filesystemen auch schreibend): Daten-Änderungen, neue Files oder Löschungen im primären Filesystem wirken sich nicht auf den Snapshot aus und solche im Snapshot beeinflussen das primäre Filesystem nicht.

  Snapshots haben zahlreiche Anwendungen:

  - Sie erlauben in großen Rechenzentren mit rund-um-die-Uhr-Betrieb *konsistente Backups bei laufenden Anwendungen*: Es wird ein Snapshot erstellt, und das Backup-Programm hat dann viel Zeit, den im Snapshot fixierten Stand wegzusichern, während die Anwendungen am primären System weiterarbeiten.
  - Ein solcher Snapshot des Vortages, Vorvortages usw. kann auch dazu verwendet werden, um den Benutzern im Fall von versehentlich gelöschten oder fehlerhaft geänderten Files einen *problemlosen Zugriff auf den alten Stand* zur Verfügung zu stellen: Sie sehen dafür einfach ein "Gestern-Laufwerk".
  - Snapshots können *vor kritischen Operationen* (z.B. komplexen Software-Updates oder Datenbank-Reorganisationen) erstellt werden. Scheitert die Operation, kann das primäre System auf den Stand des Snapshots *zurückgesetzt* und damit das misslungene Update rasch und spurlos rückgängig gemacht werden. Geht sie gut, kann durch Vergleich beider Systeme analysiert werden, was sich alles geändert hat.

  Snapshots sind nur effizient möglich, wenn das Filesystem entweder Log-strukturiert aufgebaut ist (siehe unten) oder Copy-on-write unterstützt: In diesem Fall ist ein Snapshot mit relativ geringem Aufwand implementierbar und benötigt auch kaum zusätzlichen Platz: Nur diejenigen Blöcke, die seit dem Erstellen des Snapshots in einem der beiden Filesysteme geändert wurden, werden im Moment des Änderns doppelt gespeichert. Alle Blöcke, die *im Snapshot und im primären Filesystem ident* sind, liegen *nur einmal* auf dem Speichermedium.

- **Log-strukturierte Filesysteme:**

  Ein Log-strukturiertes Filesystem *schreibt immer nur "neue" Daten und Metadaten in zuvor freie Blöcke* des Speichermediums, es *aktualisiert nie schon geschriebene Daten "in Place"*:

  - Bei einer Änderung eines Files oder eines Directories wird zuerst die neue, geänderte Kopie in einen freien Bereich geschrieben.
  - Dann wird eine geänderte Kopie aller betroffenen Metadaten gespeichert,

damit diese auf die neuen anstatt der alten Daten zeigen.

- Zuletzt werden die alten Versionen der Daten und Metadaten als freier Platz für zukünftige Schreibvorgänge registriert.

Das Schreiben erfolgt dabei *rein sequentiell reihum* in die freien Lücken. Erreicht es dabei das Ende des Mediums, beginnt es wieder in den inzwischen freigegebenen Blöcken am Anfang des Speichermediums.

Log-strukturierte Filesysteme haben mehrere Vorteile:

- *Snapshots* können sehr einfach implementiert und sehr schnell erstellt werden: Der Snapshot *zeigt einfach auf den im Moment des Snapshot-Erstellens gültigen Stand der Files, Directories und Metadaten* und ignoriert alles, was danach in das Filesystem geschrieben wurde. Normalerweise werden Daten und Metadaten freigegeben, nachdem sie durch neuere Daten an anderer Stelle ersetzt wurden. Gehören die Daten aber *zu einem aktiven Snapshot*, werden sie *nicht freigegeben*, sondern bleiben in ihrem "eingefrorenen" Zustand erhalten. Die Freigabe erfolgt erst, wenn der Snapshot deaktiviert wird.

  Aus denselben Gründen sind *File-Versionen* leicht zu realisieren: Die alten Stände eines Files bleiben ja ohnehin auf der Platte gespeichert, bis das Schreiben sie nach einer kompletten Runde überschreibt. Man muss also nur die Freigabe dieses Platzes verhindern und in den Metadaten zusätzlich auf die alte Version verweisen.

  Auch ein *"Undelete"- bzw. "Undo"-Feature* ist relativ leicht zu implementieren, weil vor kurzem gelöschte bzw. geänderte Daten noch relativ lange unverändert auf der Platte verfügbar sind.

- Log-strukturierte Filesysteme brauchen *keinen File System Check* nach einer harten Unterbrechung: Jedesmal, wenn beim Anhängen an das Filesystem ein konsistenter Zustand erreicht wird, wird dies entsprechend markiert bzw. vermerkt. Bei einem Recovery wird *einfach auf den letzten Konsistenzpunkt zurückgesetzt* und von dort weitergeschrieben. Damit hat das Filesystem wieder den Zustand des letzten Konsistenzpunktes: Alle *seitdem* erfolgten, unvollständigen Änderungen sind dadurch *sauber und spurlos entfernt*.

- Die Schreiblast verteilt sich ziemlich gleichmäßig auf den gesamten Platz, was gut für Flash-Speicher ist. Auch das sequentielle Schreiben großer Blöcke kommt sowohl Flash als auch konventionellen Laufwerken entgegen.

Fast alle Filesysteme für "rohe" Flash-Devices arbeiten Log-strukturiert, aber auch einige moderne "große" Filesysteme wie BtrFS und ZFS sind so aufgebaut.

- **Overlay-Filesysteme bzw. Union-Filesysteme** (das ist beides dasselbe):

  Solche Filesysteme *kombinieren zwei getrennte Filesysteme* zu einem einzigen logischen Filesystem.

  - Das *primäre Filesystem* ist normalerweise *groß, bereits gefüllt, und readonly*.
  - Das Overlay-Filesystem ist *normalerweise viel kleiner, les- und schreibbar*, und am Anfang in vielen Fällen noch *komplett leer*.

  Der Benutzer "sieht" im Wesentlichen den Inhalt des primären Filesystems,

aber er kann es ändern und beschreiben (obwohl das Filesystem readonly ist): Die *Änderungen* werden *in das Overlay-System geschrieben* und *beim Lesen an Stelle der primären Daten gelesen* (d.h. wenn dieselben Files oder Directories in beiden Filesystemen liegen, bekommt der Benutzer den Stand aus dem Overlay, das Overlay überlagert also die Daten im primären Filesystem).

Selbst Löschen ist möglich: Die Files und Directories werden dann im Overlay mit einer *speziellen Gelöscht-Markierung angelegt*. Diese Gelöscht-Einträge sorgen dafür, dass die entsprechenden Files des primären Filesystems (die ja immer noch vorhanden sind) beim Lesen *nicht mehr angezeigt* werden.

Solche Filesysteme haben vor allem drei Verwendungen:

- Bei *boot-baren Systemen auf CD/DVD* erlauben sie ein normales Arbeiten (incl. Anlegen und Ändern von Files) direkt von der CD weg (ohne Installation), obwohl das *primäre Medium readonly* ist: Die Änderungen landen entweder in einem RAM-Overlay (dann sind sie beim Reboot weg), oder auf einem USB-Stick o.ä. (dann kann man sie gemeinsam mit der CD mitnehmen und später auf demselben oder einem anderen Rechner auf dem letzten Stand weiterarbeiten).
- In *Ausbildungseinrichtungen* o.ä. schützen sie die primäre Installation vor Veränderungen und erlauben es, nach jeder Stunde in Sekundenschnelle durch Löschen des Overlays wieder den *Urzustand herzustellen*.
- Bei *virtuellen Maschinen* (oder z.B. bei vielen PC's, die ihr System oder ihre Software von einem gemeinsamen Netzlaufwerk beziehen) erlauben sie es, ein *gemeinsames Filesystem mit der derselben SW-Installation* für alle Maschinen zu verwenden, aber trotzdem jede Maschine unabhängig zu betreiben und jeder Maschine ihre eigenen Daten schreiben zu lassen, ohne dass die anderen darauf Zugriff haben. Das spart Platz (weil die Overlays jeder Maschine viel kleiner als eine komplette Kopie des Systems) und Administrationsaufwand (weil nur eine einzige Installation aktualisiert werden muss).

Ebene des Backup

Ein Backup kann auf beiden Ebenen (*"über" oder "unter" dem Filesystem*) ansetzen:

- Ein ***File-orientiertes Backup*** speichert *File für File* incl. ihrer Metadaten auf das Backup. Das ist der Regelfall. Ein solches Backup erlaubt es, *wahlweise alle Files oder bestimmte Files* zurückzusichern.
- Ein ***Image-Backup*** kopiert das gesamte *Block Device* bzw. Logical Volume *sequentiell als Folge von Blöcken*, *ohne* Kenntnis des Filesystems bzw. der Zuordnung von Blöcken zu Files und Directories.
  Es benötigt *mehr Platz* (weil ja auch alle unbenutzten Blöcke mitkopiert werden, außer Speichermedium und Backup-Programm beherrschen Thin Provisioning) und kann nur *als Ganzes* in ein exakt gleich großes Logical Volume zurückgespielt werden, ist aber (vor allem auf "echten" Platten, nicht SSD's) oft schneller, weil die Platte nur sequentiell (mit minimaler Kopfbewegung) gelesen wird.

  Andererseits kann (oder darf) ein VM-Host (bzw. ein Cloud-Provider, der "nackte" virtuelle Maschinen bereitstellt) oft nicht feststellen, mit welchen Filesystemen die Logical Volumes seiner Gastsysteme formatiert sind. Er kann daher kein File-

orientiertes Backup machen, sondern muss das gesamte Logical Volume sichern.

In beiden Fällen gibt es beide Backup-Typen:

- Beim **vollen Backup** werden *alle Daten* auf das Backup kopiert. Aus einem vollen Backup kann daher der gesamte Datenbestand wiederhergestellt werden.
- Beim **inkrementellen Backup** werden nur diejenigen Dateien (beim File-orientierten Backup) bzw. Disk-Blöcke (beim Image-Backup) kopiert, die sich seit dem letzten Backup *geändert* haben. Zur *Wiederherstellung* benötigt man daher das letzte zeitlich davorliegende *volle Backup und das inkrementelle Backup* (wenn es alle Änderungen seit dem vollen Backup enthält) oder alle seitdem erzeugten inkrementellen Backups (wenn jedes nur die Änderungen zum unmittelbar davorliegenden Backup enthält).

Netzwerk-Storage

Bei Netzwerk-Storage ist zu unterscheiden, ob das Netzwerk

"über" der Filesystem-Ebene
(d.h. zwischen Anwendung und Filesystem)

oder ***"unter" der Filesystem-Ebene***
(d.h. zwischen Filesystem und Plattenlaufwerken)

sitzt.

Im *ersten Fall* spricht man von *Netzwerk-Filesystemen*,
im *zweiten Fall* von *Storage Area Networks* und ev. *Cluster-Filesystemen*.

In vielen Fällen ist auch *beides* der Fall:
Die *Clients* greifen über ein *Netzwerk-Filesystem* auf den Filesystem-Server zu,
und der *Filesystem-Server* greift über ein *Storage Area Network* auf die Platten zu.

Netzwerk-Filesysteme

Netzwerk-Filesysteme

stellen ein auf dem Server laufendes "normales" Filesystem über das Netz zur Verfügung,

d.h. sie setzen auf ein schon vorhandenes, lokales,
mehr oder weniger beliebiges Filesystem auf.

Beispiele sind

- die *Microsoft Windows Fileserver* (deren Filesharing-Protokoll *SMB* bzw. *CIFS = "Common Internet File System"* heißt),
- *NFS = "Network File System"* als klassisches *Unix-/Linux-Netzwerk-Filesystem*
- und *AFS sowie Coda* für *Großinstallationen*.

Die Grenzen verschwimmen dabei: Linux kann heute problemlos als CIFS-Client und CIFS-Server agieren, Windows als NFS-Server und -Client. Auch AFS gibt es für beides.

Alle diese Systeme basieren auf *"normalen" TCP/IP-Netzen*.
Die Protokoll-Befehle zum Zugriff sind *File-orientiert, wie bei lokalen Zugriffen*

auf ein Filesystem (*"Directory anzeigen", "File öffnen / schließen", "Blöcke aus File lesen / in File schreiben", "File löschen", "Rechte setzen", ...*). CIFS entspricht komplett diesem Modell, NFS verwendet ein serverseitig "zustandsloses" Protokoll (zumindest in den klassischen Versionen), und AFS und Coda arbeiten mit kompletten File-Inhalten und lokalen Caches (d.h. ein File wird beim Öffnen gleich komplett in einen lokalen Cache kopiert und nach dem Schließen als Ganzes zurückkopiert).

Allen Systemen gemeinsam ist, dass jeder Server auf seine Platten *exklusiven Zugriff* hat und die Filesysteme darauf *lokal und allein verwaltet* (außer man verwendet "unter" dem Netzwerk-Filesystem ein Cluster-Filesystem, siehe unten).
Der Server stellt damit betreffend Verfügbarkeit einen Single Point of Failure dar.

Cluster-Filesysteme

Bei der Verwendung von Logical Volumes auf Storage-Systemen und SAN's muss immer zuerst eine Grundfrage geklärt werden:

Wie viele Server greifen ***gleichzeitig*** auf ein Logical Volume zu?

- Wenn ein Logical Volume zu jedem Zeitpunkt *immer nur einem einzigen Server* zugeordnet ist, kann man das Logical Volume mit einem *beliebigen, "ganz normalen" Filesystem* formatieren, *als ob es eine lokale Platte wäre*.

  Der primäre Sinn bzw. Nutzen des SAN besteht dann nur darin, bei Ausfall eines Servers dessen Logical Volumes möglichst rasch (bzw. sogar automatisch) einem anderen Server zuzuordnen, der die Arbeit des ausgefallenen Servers übernimmt und mit dessen letztem Datenstand weiterarbeitet.

- Wenn hingegen *mehrere Server gleichzeitig* und gleichberechtigt dasselbe Logical Volume (und damit dasselbe Filesystem) lesen und schreiben wollen, ist dafür ein ***Cluster-Filesystem*** nötig:

  Es sorgt vor allem dafür, dass *konkurrierende Updates* von Filedaten und vor allem Metadaten durch mehrere Server (oder z.B. gleichzeitige Platz-Allokation durch mehrere Server) *nicht zu inkonsistenten Filesystemen* oder Daten führen, aber z.B. auch dafür, dass ein harter Ausfall eines Servers die Zugriffe der anderen Server nicht beeinträchtigt.

  Diese *gleichzeitige, gemeinsame Nutzung desselben Filesystems durch mehrere (viele) Server* hat ihren Preis: Cluster-Filesysteme sind *hochkomplexe Gebilde*, die *administratives Know-How* erfordern.
  Außerdem sind Zugriffe durch die zusätzlich erforderliche Synchronisation *merklich langsamer* als auf "normalen" Filesystemen.

  Solche Cluster-Filesysteme können *auf mehrere Ziele hin optimiert* werden, das "universell optimale" Cluster-Filesystem gibt es nicht:

  - In *Supercomputern* kommt es auf die *Maximierung des Durchsatzes* durch viele gleichzeitige Zugriffe und *Verteilung der Last* auf viele Server an.
  - In kommerziellen Rechenzentren steht oft die *Hochverfügbarkeit* im Vordergrund: Es darf *keinen "Single Point of Failure"* geben und *keine Ausfallzeiten* oder Verzögerungen, wenn Server "on the fly" dazu- oder wegkommen.

Beispiele für Cluster-Filesysteme sind Ceph, Lustre, OCFS2, GFS2 oder GlusterFS.

Cluster

Cluster sind *Verbünde von eigenständigen, aber mit leistungsfähigen Netzwerken gekoppelten Servern*. Insbesondere erlauben diese Netzwerke typischerweise *jedem Server* im Cluster den Zugriff auf *jeden Plattenspeicher* im Cluster:
Die Platten (außer den Systemplatten zum Booten) stecken in vielen Fällen nicht in den Servern selbst, sondern *in Storage-Systemen*, die über Netzwerke angebunden sind.

Cluster dienen vor allem 2 Zwecken:

- Der Erhöhung der Rechenleistung oder des Storage-Durchsatzes:

  "HPC-Cluster" = "High Performance Computing"-Cluster
  (landläufig "Supercomputer" genannt)

- Der Erhöhung der Ausfall-Sicherheit und der Verfügbarkeit:

  "HA-Cluster" = "High Availability Cluster"
  (die typische Art und Weise, wie Rechnersysteme in kommerziellen Rechenzentren aufgebaut sind und betrieben werden)

Wir wollen hier vor allem HA-Cluster betrachten.

- In HA-Clustern sind *alle Komponenten* (Server, Storage-Systeme, Netzwerk-Verbindungen und Netzwerk-Switches, ...) typischerweise *redundant* ausgelegt, d.h. *mehrfach vorhanden*, um einen *"Single Point of Failure" zu vermeiden*.

- HA-Cluster für kommerzielle Zwecke sind normalerweise auf *mindestens zwei Rechenzentren an zwei geografisch getrennten Standorten* (mehrere Kilometer Abstand) verteilt, um den Betrieb auch bei *Totalausfall eines Standortes* (Hochwasser, Sabotage, Infrastruktur-Schäden, ...) sicherzustellen.

  Dabei bietet *jeder Standort* eine Hardware- und Software-Ausstattung, die *auch zum alleinigen Betrieb* (ohne den zweiten Standort) ausreicht. Insbesondere werden *alle Plattendaten* auf beiden Standorten aktuell und vollständig gespeichert (d.h. *zwischen den Standorten synchron gespiegelt*) und auch *innerhalb jedes Standorts* durch *Redundanz* gesichert (d.h. zumindest ein eigenes RAID 5 pro Standort).

- Die *Vernetzung* der Standorte ist ebenfalls *redundant* auszulegen, z.B. über zwei geografisch getrennte Glasfaser-Trassen (!) bzw. verschiedene Provider.

Software-Anwendungen können in Hochverfügbarkeits-Clustern auf drei Arten betrieben werden:

- **Aktiv + Cold standby:**

  Hier läuft die Anwendung stets *nur auf einem Server* im Cluster.

  Der für Ausfälle bereitstehende *zweite Server* ist zwar gebootet, aber hat *weder die Filesysteme* gemountet, auf denen die Daten der Anwendung liegen, *noch hat er die Anwendung selbst gestartet*. Erst bei Ausfall des primären Servers werden die Anwendungsdaten-Filesysteme mit dem zweiten Server verbunden und dort die Anwendung gestartet.

  Diese Betriebsart ist nötig, wenn Anwendung und Filesysteme *nicht clusterfähig*

sind.

Das (ungeplante, normalerweise automatisierte) Verschieben der aktiven Anwendung im Fehlerfall auf einen anderen Server heißt ***"Failover"***.

In der Praxis lässt man den Standby-Server oft nicht komplett leerstehen, sondern *betreibt eine andere, unabhängige Anwendung* darauf, d.h. im Normalbetrieb läuft Anwendung A auf Server X und Anwendung B auf Server Y, und bei Ausfall eines Servers muss der andere Server beide Anwendungen gleichzeitig übernehmen.

- **Aktiv + Hot standby:**

  Auch hier läuft die Anwendung *nur auf einem Server im Benutzerbetrieb*.

  Auf dem Standby-Server läuft die Anwendung zwar, bedient dort aber *keine Clients*, sondern hält nur ihren *internen Status und ihren Datenbestand* über das Netz *synchron* mit der am primären Server aktiv laufenden Anwendung (bei Datenbanken beispielsweise mittels *"Log Shipping"*).

  Im Fehlerfall kann die Anwendung am zweiten Server daher *rascher übernehmen*, und je nach Anwendung hat sie auch einen aktuelleren internen Status (im RAM, z. B. Informationen über gerade angemeldete Benutzer und deren laufende Bearbeitungsschritte) als eine frisch gestartete Anwendung.

  Dafür sind Anwendungen nötig, die eine solche Betriebsart unterstützen, und *entweder Cluster-Filesysteme* (zum gemeinsamen Zugriff auf dieselben Daten) oder *eine eigene Instanz der Plattendaten* (insbesondere der Datenbanken) pro Server.

- **Aktiv + Aktiv:**

  Hier läuft die Anwendung im Normalbetrieb *auf beiden / mehreren / vielen Servern*, alle laufenden Instanzen der Anwendung sind *gleichwertig* und *bedienen Benutzer*, d.h. *teilen sich die Last* (bzw. die Last wird über einen *Load Balancer* automatisiert auf alle verfügbaren Server verteilt).

  Wenn die Anwendung keine eigene Datenhaltung hat und die Datenbank auf einem separaten Server (und zwar nur auf einem einzigen) läuft, ist dies relativ leicht zu realisieren. Soll hingegen auch die Datenhaltung und die Datenbank auf mehrere aktive Server verteilt werden, ist ein *Cluster-Filesystem und/oder eine Cluster-fähige Datenbank* erforderlich.

Damit die Clients die verlagerten Anwendungsserver automatisch finden (bzw. im günstigsten Fall vom Wechsel gar nichts mitbekommen), werden im Failover-Fall oft auch automatisch *die Netzwerke umkonfiguriert*, entweder auf *DNS-Ebene* (d.h. der neue Server übernimmt auch den *Netzwerk-Namen* des ausgefallenen Servers), oder auf *IP-Ebene* (d.h. der neue Server bekommt die *IP-Adressen des ausgefallenen Servers*, im Extremfall sogar die MAC-Adressen).

"Heartbeat" und "Split Brain":

Eine zentrale Frage ist, wie eine *Situation erkannt* wird, die ein Failover oder eine andere *Umkonfiguration eines HA-Clusters nötig macht* (das Failover selbst läuft dann meist vollautomatisch bzw. Skript-gesteuert).

Realisiert wird diese Erkennung meist dadurch, dass auf *allen Servern* (sowohl dem Primär-Server als auch allen Standby-Servern), die im Cluster für dieselbe Anwendung zuständig sind, *ständig der sogenannte Heartbeat-Dienst* läuft, der die *Funktion*

der anderen Server (und ev. auch die Verfügbarkeit der Anwendung darauf) *laufend über das Netz prüft* (ähnlich "ping"), und zwar redundant über mindestens zwei völlig getrennte Verbindungen (für eine davon wird mitunter eine klassische serielle Schnittstelle unabhängig von allen Netzen verwendet). Wenn der "gegenüberliegende" Server *auf keiner der Heartbeat-Verbindungen mehr erreichbar* ist, nimmt der lokale Server an, dass tatsächlich der gegenüberliegende Server ausgefallen ist und nicht nur eine Netzverbindung. Daher geht der lokale Server davon aus, dass er *jetzt allein ist* und sofort *alle Funktionen des ausgefallenen Servers als primärer Server übernehmen muss*.

Ebenso erkennt der Heartbeat-Dienst, dass die Gegenseite *wieder verfügbar* geworden ist, und leitet die automatische Rück-Konfiguration ein. Dazu gehört unter anderem, dass die *Daten* der Storage-Systeme auf beiden Seiten *wieder synchronisiert* werden, d.h. dass alle Änderungen der Plattendaten, die innerhalb der Zeit erfolgten, in denen der Cluster getrennt war (und damit auch die Storage-Spiegel nicht spiegeln konnten), automatisch auf das "rückständige" Storage-System kopiert werden.

Sollte es hingegen dazu kommen, dass tatsächlich *alle Netzverbindungen* zwischen zwei Servern *ausgefallen* sind, aber *die Server selbst noch beide laufen*, kommt es zu einer sogenannten *"Split-Brain"-Situation*: Beide Server glauben, der "einzige Überlebende" zu sein, und starten *unabhängig voneinander die Anwendung im primären Modus*, jeder Server nur *mit der am eigenen Standort gespeicherten Kopie* der gemeinsamen Daten und Datenbanken (weil ja der Spiegel-Verbund der Storage-Systeme ebenfalls auseinanderbricht, wenn alle Netzverbindungen zwischen den Standorten ausfallen).

Diese Situation ist der *"größte anzunehmende Unfall"* in einem Cluster: Wenn beide Seiten unabhängig voneinander (und im Glauben, der einzige zu sein) verschiedene Änderungen in ihrer lokalen Kopie der Filesysteme und vor allem der Datenbanken vornehmen, entstehen höchstwahrscheinlich *irreparable Inkonsistenzen* (beispielsweise werden bei automatischer fortlaufender Nummerierung dieselben Nummern für verschiedene Dinge vergeben, es werden Datenbank-Einträge mit kollidierenden Schlüsseln angelegt, oder es wird derselbe freie Platz in einem Filesystem verschiedenen neuen Files zugeteilt).

Cluster versuchen daher, diese Situation mit allen Mitteln zu *verhindern*.
Besonders unter Linux haben sich dafür radikale Maßnahmen eingebürgert:
Im Extremfall schaltet bei Ausfall des Heartbeats eine Seite über einen unabhängigen Kommunikationskanal (z.B. Handy-Netz oder serielle Leitung) irgendeiner zentralen, aber unempfindlichen Komponente der anderen Seite den Strom ab.
Wenn dies auf beiden Seiten mit unterschiedlicher Verzögerung geschieht (z.B. A killt B nach 10 Sekunden ohne Heartbeat, B (falls er das überlebt) killt A nach 25 Sekunden), so ist mit ziemlich großer Wahrscheinlichkeit sichergestellt, dass genau einer überlebt und nicht beide.

Storage Area Networks

Storage Area Networks (SAN's) *verbinden Plattenlaufwerke bzw. Storage-Systeme* ohne eigene Filesystem-Intelligenz *mit Servern*, und zwar typischerweise über *kurze* (innerhalb eines Rechenzentrums) und *mittlere* (zwischen zwei zusammengehörenden Rechenzentren) *Distanzen*.

Die Storage-Systeme stellen dabei nur *Logical Volumes über das Netz* zur Verfügung (implementieren aber z.B. RAID oder Snapshots lokal), sie "wissen" nichts von Files

und Filesystemen. Die *Netzwerk-Protokolle für SAN's* sind daher *nicht File-basiert*, sondern *rein Block-basiert* (z.B. *"lies 32 Blöcke ab Block 723 von Logical Volume 42"*).

Es gibt mehrere Varianten:

- **iSCSI** (*"Internet SCSI"*) "verlängert" das klassische, seit Jahrzehnten standardisierte *lokale Platten-Zugriffs-Protokoll SCSI über TCP/IP*. Es ist die *preisgünstige Variante* für normale *Ethernet- und TCP/IP-Infrastrukturen* und wird oft zwischen Servern und kleinen bis mittleren Storage-Systemen eingesetzt.
- **FC** (*"Fibre Channel"*) ist die *klassische Rechenzentrums-SAN-Technologie*.
  Es umfasst eigene, *nicht Ethernet-kompatible Netzwerk-Hardware*
  (Glasfaser-basiert, mit eigenen Adaptern und Switches)
  und *eigene, nicht TCP/IP-kompatible Protokolle*, beides optimiert für Storage-Systeme
  (es ist allerdings möglich, IP "huckepack" über FC-Netze zu leiten und umgekehrt FC-Protokolle über Ethernet bzw. IP zu tunneln).

  Es sind zwar Server-Plattenlaufwerke mit direktem FC-Anschluss verfügbar,
  aber normalerweise werden große Storage-Systeme mittels FC angebunden,
  die über eigene Intelligenz verfügen und ihre Platten intern "ganz normal"
  (mit SAS oder SATA) anbinden.
- **Infiniband** bietet ebenfalls *spezielle Protokolle und Interface-Karten* zur Einbindung von Storage-Systemen. Infiniband ist eine Hardware-Technologie mit *eigenen Adaptern und Switches zum internen Datentransfer in Supercomputern* und als Interconnect in Server-Clustern, die um rund eine Größenordnung höhere Geschwindigkeit und vor allem wesentlich geringere Latenz als Netzwerke bietet.
  Es verwendet im Bereich bis zu 10 m spezielle Kupferkabel und darüber Glas.
  Es bietet z.B. Knoten-zu-Knoten-DMA, d.h. RAM-Daten-Transfer zwischen Knoten.
- **NBD** (*"Network Block Device"*) ist wie iSCSI ein Protokoll zum *Zugriff auf Block Devices über normale TCP/IP-Netze*. Es wird aber selten für Storage-Systeme eingesetzt, sondern vor allem dann, wenn *Linux-Server sich gegenseitig ihre lokalen Platten zur Verfügung stellen* wollen (z.B. Server A lagert ein Spiegel-Volume seines lokalen RAID-Verbundes auf Server B aus und umgekehrt).

  DRBD (*"Distributed Replicated Block Devices"*) ist eine auf Linux implementierte *Weiterentwicklung des NBD-Konzeptes*, die speziell für diesen Zweck,
  d.h. *zum Aufbau verteilter RAID-Verbünde aus den lokalen Platten der einzelnen Server* in hochverfügbaren Server-Clustern, optimiert wurde.
  Es unterstützt diverse RAID-Operationen direkt im Treiber:
  Wenn beispielsweise die Verbindung zu einem Server unterbrochen wird,
  resynchronisiert DRBD nach Wiederherstellung der Verbindung
  automatisch wieder die betroffenen RAID-Volumes
  (und kopiert dabei nur die geänderten Blöcke).

  Solche Lösungen mit lokal angebundenen Platten sind zwar nicht so leistungsfähig und komfortabel wie ein SAN mit Servern und großen Storage-Systemen, aber wesentlich preisgünstiger.

Virtuelle Maschinen und Container

Der Begriff "virtuelle Maschine" (VM) wird in der Informatik in der Praxis für zwei sehr unterschiedliche Dinge verwendet:

- Bei der ***Implementierung von Programmiersprachen*** ist die "virtuelle Maschine" (z.B. die *Java-VM* oder die *"Common Language Runtime"* von Microsoft für .net) *ein Stück Software*, das *Programme dieser Programmiersprachen ausführt*.

  "Virtuell" ist daran vor allem der *Prozessor*:
  Programme dieser Programmiersprachen werden nicht in Binärcode einer realen Prozessor-Architektur compiliert. Stattdessen erzeugt der Compiler *portablen Binärcode für eine imaginäre Prozessor-Architektur* mit eigenem Befehlssatz, und die virtuelle Maschine führt diesen Code aus, indem sie diese Befehle in irgendeiner Form emuliert oder auf die Befehle des realen Prozessors abbildet.

- Auf ***Betriebssystem-Ebene*** ist eine "virtuelle Maschine"

  *ein vom **"Host-Betriebssystem"***
  in Software emulierter Hardware-Rechner,
  *auf dem ein eigenes **"Gast-Betriebssystem"***
  wie auf einem echten Rechner gebootet und ausgeführt wird.

  Die Aufgabe des "Host-Betriebssystems" ist dabei weniger, einen CPU-Befehlssatz zu emulieren, denn in 99 % der Fälle verwendet das *Gastsystem dieselbe CPU-Architektur wie das Host-System*, d.h. die *Software des Gastsystems kann direkt auf der realen CPU ausgeführt werden* (es gibt aber sehr wohl die Möglichkeit, z.B. unter Linux auf einem x86-Hostsystem ein ARM-Gastsystem zu booten).

  Vielmehr besteht die Aufgabe der VM-Implementierung und des Host-Betriebssystems darin, *dem Gastsystem eine komplette System-Hardware* (I/O-Geräte, CPU-Statusregister, MMU, Interupt-System, ...) *vorzutäuschen*, obwohl das Gastsystem keinen direkten und vollständigen Zugriff auf die Host-Hardware bekommt, sondern die System-Ressourcen wie RAM und I/O-Geräte vom Hostsystem verwaltet werden. Hier ist also *die gesamte Hardware incl. I/O-Geräten*, die das Gastsystem sieht, *virtuell*.

Hier befassen wir uns im Folgenden nur mit der zweiten Bedeutung von VM's.

Das Konzept ist älter als allgemein vermutet: Es wurde schon vor 1970 erstmals von IBM für den IBM/360-Großrechner angeboten, um auf einem zentralen Rechner jedem Benutzer allein und unabhängig von anderen sein eigenes Betriebssystem bzw. "einen Rechner für ihn allein" zu geben.

Heute werden VM's für viele Zwecke eingesetzt:

- Sie dienen dazu, um *auf einem Betriebssystem ein anderes Betriebssystem laufen zu lassen*.
- Sie dienen dazu, vor allem in großen Rechenzentren die Server-Hardware möglichst effizient zu nutzen, indem man *einen Rechner mit mehreren VM's auslastet*.

Insbesondere *Cloud-Provider* benötigen eine Möglichkeit, *jedem Kunden seine "eigenen Rechner"* zu geben, die der Kunde selbst verwalten kann, und die von den Cloud-Instanzen anderer Kunden völlig unabhängig, getrennt und geschützt sind, aber je nach aktueller Belastung die Cloud-Instanzen mehrerer / vieler Kunden gleichzeitig auf demselben Server laufen zu lassen (weil eine Kunden-VM oft viel zu wenig Last erzeugt, um einen Server ständig allein damit effizient auszulasten).

- Neben der effizienten Systemnutzung dienen VM's auch der *Verfügbarkeit*: Bei Störungen oder geplanten Abschaltungen kann eine laufende VM in Sekundenbruchteilen *eingefroren*, auf einen File *kopiert*, *auf eine andere Maschine verlagert* und dort weiter ausgeführt werden.
- Es ist oft schwierig, viele Anwendungen gemeinsam auf einem einzigen System zu betreiben: Die Anwendungen erfordern häufig verschiedene Versionen des Systems, der Datenbank oder irgendwelcher Libraries, oder sie verlangen unterschiedliche Systemeinstellungen.

  In solchen Fällen kann es praktischer sein, *jede Anwendung für sich allein in einer VM* zu installieren: Die VM kann dann optimal an die Anwendung angepasst werden.

  Auch datenmäßig sind die Anwendungen *besser voreinander geschützt*, wenn sie *in getrennten Systemen* laufen, insbesondere bei Sicherheitslücken.
- Oft werden *"Wegwerf-VM's"* benötigt, z.B. um Software-Installationen, Upgrades oder kritische Systemkonfigurations-Änderungen vor der Durchführung am Produktivsystem in einem Spielzeug-System zu testen.
- In manchen Fällen ist es nötig, *sehr viele verschiedene Systemkonfigurationen* parallel vorzuhalten, die aber nur sporadisch benötigt werden (beispielsweise im Support eines SW-Anbieters Kopien der Installationen wichtiger Kunden, oder in der Qualitätssicherung eines SW-Entwicklers Installationen aller unterstützten Plattformen in allen unterstützten Konfigurationen).

Der Betrieb mit VM's hat aber auch Nachteile:

- Dasselbe System läuft in einer VM *ineffizienter als auf nackter Hardware*:
  - Die CPU muss laufend *zwischen Gast und Host hin- und herschalten* (vor allem bei jedem I/O), manche Dinge werden aufwändig emuliert, I/O-Daten werden oft einmal mehr umkopiert (vom Gast zum Host und vom Host zum I/O-Gerät) bzw. durchlaufen im Fall von Netzwerkverkehr eine zusätzliche Routing-Ebene, manche Host-Features (z.B. Grafik-Beschleuniger oder intelligente Netzwerk-Karten) sind vom Gast aus nicht nutzbar, Page Faults und MMU-Operationen müssen oft sowohl im Gast-Beriebssystem als auch im Host-Betriebssystem behandelt werden usw..
  - Auch *speichermäßig* sind VM's ineffizienter: Es liegen ja *zumindest zwei komplette Betriebssyteme* im Speicher, und auch viele Daten des Betriebsystems gibt es (zumindest teilweise) doppelt, z.B. Pagetables oder Disk-Caches. Bei einem Betrieb dutzender VM's auf einem Server multipliziert sich die Anzahl der Betriebssysteme und Daten im Speicher entsprechend (manche Host-Betriebssysteme unterstützen daher automatisches *"Samepage Merging"*: Der Host prüft periodisch, ob es im RAM Pages mit identem Inhalt gibt, und legt diese nach dem Copy-On-Write-Prinzip mittels MMU zusammen).

Je nach Anwendung, VM-Implementierung und Hardware kann der *Effizienzverlust* durch VM's *von wenigen Prozent bis zu einer Größenordnung* reichen.

Grundsätzlich gilt, dass im Laufe der Jahre viele Hardware-Features in moderne Prozessoren eingebaut wurden, die die Implementierung von VM's wesentlich effizienter als früher machen, weil sie die VM-Software vereinfachen: Das betrifft beispielsweise eine eigene Hardware-Prioritätsebene für den Hypervisor (auf x86: Der Gast läuft nach wie vor auf Ring 0, hat dort aber nicht mehr alle Hardware-Rechte, und der Host bekam eine neue Rechte-Ebene, die den bisherigen Ring 0 kontrollieren kann), Virtualisierungs-Unterstützung bei den Pagetables und der MMU sowie bei den Interrupts, oder die I/O-MMU, die es erlaubt, den direkten Zugriff von Gast-Systemen auf einzelne I/O-Devices gezielt zu erlauben oder abzufangen.

- Durch VM's (und Container, siehe unten) muss die Systemadministration nicht mehr ein System pro Server verwalten, sondern viele, manchmal hunderte. Diese müssen *alle gepflegt und aktuell gehalten* werden. Allein schon die Buchführung, welche Softwarekomponenten in welchen Versionen in welcher VM installiert sind, kann zur Herausforderung werden (u.a. weil Container-Anbieter oft gar nicht angeben, welche Komponenten in ihren Containern vorinstalliert sind).

  Gerade im Zusammenhang mit Updates sind VM's daher ein *Sicherheits-Risiko*: Die Gefahr, dass bei einer Vielzahl von VM's auf irgendeiner VM vergessen wird, einen sicherheitskritischen Patch zeitnah einzuspielen, ist groß (z.B. weil die VM länger nicht mehr benutzt wurde, oder weil man schlicht vergessen hat, dass irgendein seltenes Stück Software irgendwann einmal auch auf dieser VM installiert wurde).

Für die Realisierung eines VM-Host-Systems gibt es drei Möglichkeiten:

- Der Host kann ein ***spezieller VM-Hypervisor*** sein: Ein solcher Hypervisor hat *nur den Zweck*, die *Hardware zu verwalten und die VM's zu starten* und zu überwachen. Er ist nicht dazu gedacht, auch die direkte Ausführung von Programmen zu erlauben oder eine normale Anwender-Benutzeroberfläche zu bieten. Der Hypervisor ist daher ein *abgespecktes, spezialisiertes Betriebssystem*.
- Der Host kann ein ***um spezielle VM-Mechanismen erweitertes "normales" Betriebssystem*** sein, mit *"normaler" Benutzeroberfläche* und der Möglichkeit, *parallel Anwendungen wie bisher* auszuführen und *gleichzeitig VM's* laufen zu lassen.
- In gewissen Sonderfällen kann der VM-Host auch als *ganz normale Anwendung* am Host-Betriebssystem gestartet werden, ohne Teil des Betriebssystems zu sein bzw. ohne spezielle VM-Mechanismen im Betriebssystem zu benötigen:
  - Das *Gast-System* wurde so modifiziert, dass es *nicht direkt auf die Hardware* zugreift und auch *ohne eigene Hardware-Privilegien* läuft.

    Es gibt z.B. einen eigenen "User Mode Linux"-Kernel, der speziell dafür gebaut ist, unter einem anderen Linux-Kernel als ganz normale Anwendung ohne Hardware-Rechte zu laufen.
  - Das *Gast-System* ist *für eine andere Prozessor-Plattform* gebaut, d.h. die Ausführung incl. aller Hardware-Zugriffe wird *komplett in Software emuliert*, anstatt die Gast-Anwendungen direkt am Prozessor auszuführen.
  - Das *VM-Host-Programm* hat ein paar *spezielle Device-Treiber* im Host-

Betriebssystem installiert, um die Hardware-Zugriffe des Gastes abzufangen.

Das *Gast-Betriebssystem* (das ein anderes System als das Host-Betriebssystem sein kann, z.B. Windows unter Linux oder Linux unter Windows)

- ***… glaubt, dass es auf "echter" Hardware läuft***
  (und "weiß" daher nichts vom Host-Betriebssystem)
- ***… und dass es das ganze System für sich allein hat***
  (obwohl ev. viele VM's gleichzeitig laufen und jede vom Host nur einen Teil des RAM, des Plattenplatzes, der CPU-Zeit usw. zugewiesen bekommt).

I/O in VM's

Konzeptionell werden alle Zugriffe des Gastes auf I/O-Geräte, auf MMU- und CPU-Kontrollregister, auf den Interrupt-Controller usw. vom Host abgefangen und emuliert.

Konkret gibt es folgende I/O-Möglichkeiten:

- Wenn das *Gast-Betriebssystem unverändert* (auch mit unveränderten Device-Treibern für reale I/O-Devices) laufen soll, und wenn die Host-Hardware *keine I/O-Virtualisierung* unterstützt, dann löst *jeder einzelne Zugriff des Gastsystems auf ein I/O-Gerät* automatisch eine *Umschaltung in das Host-System* aus.
  Das Host-System *emuliert dann das Verhalten des jeweiligen I/O-Chips* (auch wenn dieser Chip am Host gar nicht existiert), d.h. macht softwaremäßig genau das, was dieser Chip bei dem Zugriff gemacht hätte (inclusive des dafür nötigen I/O's).

  Als die Virtualisierung auf x86 begann, war es z.B. jahrelang üblich, dass in Gastsystemen zum Zugriff auf Platten, Bänder und CD-Laufwerke der Treiber für den SCSI-Adapter AHA-1542 installiert wurde (einer damals weit verbreiteten Karte für den ISA-Bus) und das Host-System sich zum Gast hin wie eine solche Karte mit angeschlossenen Plattenlaufwerken verhielt, selbst wenn die Platten im Host ganz anders (ev. sogar ohne SCSI-Bus) angeschlossen waren.
- Die zweite Variante ist, dass *im Gastsystem spezielle I/O-Treiber* für die virtuelle Maschine installiert werden, die *alle I/O-Zugriffe direkt an das Host-System durchreichen* (deutlich effizienter als bei einer Emulation des I/O-Chips), anstatt auf irgendeine I/O-Hardware zuzugreifen.
- Die dritte Variante (nur für bestimmte I/O-Geräte) ist, dass *das Host-System dem Gast-System direkten Zugriff auf die "echte" Hardware erlaubt* ("Passthrough").
  Dies erfordert hostseitige Hardware-Unterstützung in Form einer I/O-MMU, die einzelne I/O-Adressen in den virtuellen I/O-Bereich einer VM weiterleiten kann.
- Ein zentrales Thema ist der ***Plattenspeicher***, den die virtuellen Maschinen sehen:
  - Hier kann dem Gast entweder *ein ganzes physisches Laufwerk oder ein Logical Volume* am Host als Block Device zur Verfügung gestellt werden (in den meisten Fällen exklusiv).
  - Viel wichtiger ist aber die Möglichkeit, dass die VM's *ganz normale Files am Host als Logical Volume* verwenden (ebenfalls im Normalefall exklusiv, d.h. jede VM hat ihren eigenen "Platten-File" am Host):
    Die "Platten" der einzelnen VM's können dann am Host wie normale Files verwaltet, gesichert, zwischen Hosts kopiert usw. werden.

Vor allem für Cloud-Anbieter ebenfalls wichtig ist die Fähigkeit, diese Files am Host nicht gleich fix in der Größe des Logical Volumes im Gast anzulegen, sondern *Thin Provisioning* zu nutzen (diese Files also z.B. als *"Sparse Files"* zu speichern), d.h. am Host nur so viel Platz wirklich zu belegen, wie am Gast bisher tatsächlich genutzt bzw. beschrieben wurde.

- Die dritte Möglichkeit ist, im Gast einen *speziellen Treiber* zu installieren, der die *Zugriffe auf File-Ebene statt auf Block-Ebene an den Host* durchreicht. Der Gast "sieht" dann also *ausgewählte File-Systeme des Hosts* und kann darauf zugreifen, gleichzeitig mit dem Host und anderen Gästen.

- Ein weiteres spezielles Thema ist das ***Netz***: Im Normalfall stellen VM-Hosts jeder VM ein oder mehrere *"virtuelle Netzwerkadapter"* mit *eigenen* (für jeden VM-Gast anderen) *IP- und MAC-Adressen* zur Verfügung. Wie diese Adapter untereinander und mit den realen Adaptern verbunden sind, ist *im Host konfigurierbar*.

  Das Host-System spielt also in diesem Fall *softwaremäßig Switch oder Router zwischen den virtuellen Netzen der VM's und den realen Netz-Karten*.

  Der Host kann aber auch *mehrere Adressen verschiedener virtueller Maschinen* direkt nach draußen *an eine reale Karte binden* (die meisten Karten können das), die Karte verhält sich dann logisch wie mehrere Netzwerkinterfaces an einem einzigen Netzwerkanschluss.

  Manche Server-Netzwerkkarten (z.B. von Intel) haben sogar hardware-mäßige Virtualisierungs-Unterstützung, sodass man sie via I/O-MMU gleichzeitig direkt an mehrere VM's durchreichen kann.

- Die *am schlechtesten virtualisierte I/O-Karte* ist die ***Grafik-Karte***, sowohl bei Verwendung als *Grafikkarte*, als auch bei Verwendung als *Rechen-Beschleuniger* in Servern, als auch für die Nutzung der Hardware-Video-Decoder und -Encoder: Einerseits sind Grafik-Prozessoren hochkomplex und sehr performance-kritisch (d.h. auf direkten Zugriff der Grafik-Anwendung auf die GPU und das Grafik-RAM angewiesen), andererseits sind die Grafik-Karten der führenden Hersteller bezüglich Architektur und Ansteuerung untereinander völlig inkompatibel.

  Außerdem sind Grafik-Prozessoren (im Unterschied zu den CPU's) von der Architektur her *noch nicht wirklich Mehrbenutzer-fähig*: Es fehlt an Möglichkeiten (vor allem an *Sicherheits- und Rechte-Mechanismen*) zur Aufteilung der Grafik-Cores in mehrere Gruppen für getrennte Benutzer und Anwendungen, und es fehlt vor allem an einer MMU für das Grafik-RAM, die einzelne Grafik-Cores zuverlässig daran hindern würde, auf den Grafik-Speicher anderer Anwendungen (oder gar beliebige Hauptspeicher-Bereiche!) zuzugreifen.

  Es erscheint daher unmöglich, die Grafikkarten-Hardware-Zugriffe eines "echten" Grafikkarten-Treibers in einem Gastsystem abzufangen und schnell genug durch den Host zu emulieren. Ebenso wenig erfolgversprechend erscheint es, einen Virtualisierungs-Grafiktreiber zu schreiben, der die Grafikprozessor-Befehle und Grafikspeicher-Zugriffe vom Gast an das Hostsystem durchleitet: Einerseits bräuchte man pro Grafikkarten-Hersteller einen solchen Treiber, und andererseits wäre die Performance vermutlich unbefriedigend.

  Bisher war es also üblich, dass die Grafik in Gast-Systemen völlig unbeschleunigt und mehr oder weniger mit dem Treiber einer normalen VGA-Karte aus den

Anfängen des PC's lief und die Grafikfähigkeiten der Karte im Host ungenutzt blieben.

Aktuell werden zwei Ansätze verfolgt:

- Die *Grafikkarte hardwaremäßig an das Gastsystem durchleiten*:
  Für die exklusive Nutzung der Grafikkarte durch ein einziges Gastsystem funktioniert das derzeit schon recht gut, aber die Nutzung der Karte durch mehrere Gäste oder durch Gast und Host gleichzeitig steckt noch in den Anfängen (d.h. man braucht einen Schirm für den Gast und einen für den Host).

  Es wird auch diskutiert, wie "ausbruchsicher" solche Lösungen sind, weil das Gastsystem am Grafik-Prozessor Programme laufen lassen könnte, die in Speicherbereiche des Host-Systems eingreifen.

- Ein *Grafik-Protokoll auf höherer, hersteller-unabhängiger Ebene* festlegen, das der Treiber im Gast an den Host durchleitet, und das dann im Host mit Hilfe der Grafik-Hardware ausgeführt wird. Auch das funktioniert (zumindest für alte Versionen von OpenGL und DirectX) in einigen kommerziellen VM-Implementierungen zumindest prinzipiell, wenn auch mit Performance-Verlusten.

Container

Um die *Ineffizient von VM's* zu reduzieren, vor allem in dem Fall, dass *der Host und alle Gäste ohnehin unter derselben Betriebssystem-Version* laufen, kam man bald auf die Idee, nur mehr ein einziges Betriebssystem laufen zu lassen (d.h. keine logisch eigenständigen Maschinen mit einem Gast-Beriebssystem mehr), aber dieses Betriebssystem so umzubauen,

dass es zu den Anwendungen hin so aussieht,
als ob jede Anwendung in einer eigenen VM laufen würde:

- Die "Anwendungen" für solche Systeme heißen Container:

  Jeder solche Container enthält ein *komplettes, eigenes Filesystem* mit *demselben Inhalt wie bisher das Filesystem einer VM* (also ein Filesystem mit einer kompletten Softwareinstallation: Anwendungscode und -daten, alle Libraries, alle Systemkonfigurationsfiles usw.), *mit Ausnahme des Betriebssystems selbst*.

  Jede Anwendung bringt also ihre eigene *komplette Umgebung* mit.

  Im Unterschied zu VM's, die meistens vor Ort vom Kunden selbst erstellt werden (bzw. bei Cloud-Diensten vom Cloud-Anbieter), sind *Container portabel* und werden oft *schon vom Software-Anbieter erstellt*, d.h. der SW-Anbieter *liefert einen fertig installierten, laufbereiten Container* aus.

  In der Praxis sind Container aus Effizienzgründen oft mit *Overlay-Filesystemen* realisiert, die auf mehrere Schichten aufgespalten sind:
  Beispielsweise die unterste Filesystem-Schicht mit der System-Grundinstallation und den allgemeinen Libraries, eine Schicht mit der Anwendung, eine Schicht mit den systemspezifischen Konfigurationsdaten des aktuellen Systems, und die oberste Schicht mit allen Dateien, die sich im laufenden Betrieb ändern.

Das vereinfacht Updates, macht Backups effizienter (weil nur die oberen Schichten gesichert werden müssen), und kann vor allem viel Platz sparen:
Die unteren Schichten können readonly sein und von mehreren Containern gemeinsam benutzt werden.

- Das *Betriebssystem legt von allen seinen für die Anwendung sichtbaren Datenstrukturen eine eigene Kopie pro laufendem Container* an
  und verwaltet auch *alle I/O-Devices für jeden Container getrennt*.

  Jeder Container sieht also nur seine eigenen Platten- und Netzwerk-Devices bzw. Filesysteme und hat seine eigenen Benutzer, Gruppen und Rechte (auch "root" hat in einem Container nur für den eigenen Container Administrationsrechte und nicht für das gesamte System, da jeder Container seinen eigenen "root"-Benutzer hat). Ein Container sieht auch nur die Prozesse, die innerhalb dieses Containers gestartet wurden, und kann auch nur auf innerhalb des Containers angelegtes Shared Memory zugreifen.

Es gibt dadurch viel weniger doppelte RAM-Inhalte, keine Emulation irgendwelcher I/O-Geräte oder MMU-Funktionen mehr, und vor allem keine Umschaltung zwischen Host- und Gastsystem mehr.

Inter-Prozess-Kommunikation

Bei der *Programmierung von parallelen Anwendungen* ist nicht die Parallelität an sich (d.h. die Erzeugung mehrerer paralleler Threads oder Prozesse) das Problem, sondern der Datenaustausch bzw. der Zugriff auf gemeinsame Daten:
Ein *ungeordneter gleichzeitiger Zugriff* würde meist entweder zu *inkonsisteneten Daten* oder zu vom zeitlichen Ablauf abhängigen, *nichtdeterministischen Ergebnissen* führen (oder zu beidem).

Im einfachsten Fall lagert die Anwendung das Problem auf eine *Datenbank* (die für die Einhaltung der ACID-Bedingung auch bei vielen gleichzeitigen Zugriffen sorgt) oder auf das *Filesystem* (das zumindest konsistente Metadaten-Updates garantiert und auch Mechanismen zum exklusiven Zugriff auf File-Inhalte bietet) aus.

Ansonsten gibt es zwei wesentliche Kommunikations-Mechanismen:

- Beim ***Message Passing*** kommunizieren die parallelen Prozesse, indem sie sich *gegenseitig Nachrichten zusenden*, d.h. A schickt durch *Aufruf spezieller Funktionen* bestimmte Daten an B, und B empfängt diese Daten (bzw. eine Kopie davon) durch Aufruf spezieller Funktionen.

  Die Vorteile dieses Konzeptes sind Folgende:

  - Es ist auch einsetzbar, wenn A und B *keinen physisch gemeinsamen Speicher* (RAM) haben, d.h. wenn sie z.B. auf verschiedenen Knoten eines Clusters laufen.
  - Es ist *relativ leicht zu erlernen, zu verstehen, fehlerfrei anzuwenden* und nachzuvollziehen, weil es der *Programmierung von Netzwerk-Kommunikation sehr ähnlich* ist.

  Das Konzept hat aber auch Schwächen:

  - Die Kommunikation über Message Passing ist *aufwändig*:
    Jeder Datenaustausch erfordert auf beiden Seiten einen *Systemaufruf* und das mehrfache *Kopieren* von Daten, eventuell auch einen *Prozess-Wechsel*.
    Außerdem liegen die Daten bei beiden Prozessen getrennt im Speicher, belegen also zwei Mal Platz.
  - Message Passing ist *nur für "einfache" Daten* einfach:
    Dynamische Datenstrukturen müssen vor dem Versand aufwändig serialisiert und beim Empfänger wieder rekonstruiert werden. Ganz grundsätzlich können Pointer nicht sinnvoll über Messages übertragen werden:
    Es müssen stattdessen immer alle Daten, auf die die Pointer zeigen, mit der Message mitkopiert werden.
  - Es handelt sich *nicht wirklich um "gemeinsame" Daten*, d.h. nicht alle Beteiligten "sehen" zu jedem Zeitpunkt automatisch alle Daten, und schon gar nicht im selben Zustand.
    Damit lässt sich nicht jede Form des Datenaustausches einfach realisieren:
    Schon bei "einer schickt an alle" sind viele Nachrichten nötig, und spätestens bei "zwei oder mehr aktualisieren wechselseitig dieselben Daten in nicht vorher festgelegter Reihenfolge" wird es mit Message Passing konzeptionell schwierig.

- Bei ***Shared Memory*** sehen alle Beteiligten tatsächlich *denselben Speicher*

(d.h. dieselben RAM-Inhalte) und können *ganz normal* (d.h. lesend und schreibend, durch Variablen- und Pointer-Zugriffe und mittels Zuweisungen) *darauf zugreifen*.

- Bei *mehreren Threads innerhalb eines Prozesses* ist das konzeptionell ohnehin automatisch der Fall (weil ja alle Threads im *selben virtuellen Adressraum* laufen und damit die *globalen und statischen* sowie alle *dynamisch angelegten Daten* für alle Threads eines Prozesses *gemeinsam* sind,
  nur die Stacks mit den lokalen Variablen sind getrennt).
- Für gemeinsamen Speicher zwischen *verschiedenen Prozessen* bzw. Programmen stellt das *Betriebssystem* Mechanismen zur Verfügung, die im Wesentlichen bewirken, dass *dieselben RAM-Pages* gleichzeitig *in die virtuellen Adressbereiche mehrerer Prozesse* eingeblendet werden (mittels Pagetables und MMU).

  Unter Unix/Linux gibt es dafür zwei standardisierte Mechanismen:

  - **`mmap`** (das die *Daten eines Files* in den Adressbereich eines oder mehrerer Prozesse einblendet), wahlweise in Verbindung mit "normalen" Files, Devices oder Pseudo-Files wie **`/dev/zero`**, "anonymen" Speicherbereichen oder POSIX Shared Memory Objekten.
  - ***Klassisches Unix System V IPC Shared Memory***
    (seit rund 40 Jahren: System Calls **`shmget`**, **`shmat`**, …).

Shared Memories haben mehrere Vorteile:

- Der Zugriff darauf ist *einfach und schnell*
  (genauso einfach und genauso schnell wie auf "normale" Daten im Speicher).
- Im Shared Memory können *auch dynamische Datenstrukturen* usw. abgelegt werden.

Es hat aber auch zwei wesentliche Nachteile:

- Es funktioniert nur dann, wenn alle Beteiligten auf Cores laufen, die *physisch Zugriff auf dasselbe RAM* haben, nicht jedoch z.B. auf verschiedenen Knoten eines Clusters. Der Anzahl der Prozessoren, die mit demselben RAM verbunden werden können, sind durch die Physik jedoch enge Grenzen gesetzt (typischerweise unter hundert Cores, im Vergleich zu hunderttausenden Cores in Clustern mit Message Passing).

  Es gibt zwar Konzepte, die gemeinsamen Speicher über Systemgrenzen hinweg per Message Passing simulieren (z.B. unter Ausnutzung von Page Faults), aber deren Geschwindigkeit ist um mehrere Größenordnungen geringer.
- Damit gemeinsame Daten bei parallelen Zugriffen *konsistent* bleiben, sind spezielle Vorkehrungen nötig:
  - Der "klassische" Mechanismus dafür ist *"Locking" bzw. "Mutual Exclusion"*, z.B. mittels Semaphoren oder kritischen Regionen.
  - Eine Alternative dazu sind Zugriffe mittels *"atomarer Operationen"* (die das Locking mehr oder weniger auf die Hardware-Ebene verlagern).
  - Intel unterstützt in seinen jüngeren Prozessor-Generationen das Konzept des *"Transactional Memory"* in Hardware, womit ein *Zugriffs- und Konsistenzmodell ähnlich dem von Datenbanken* implementiert werden kann.

Allen diesen Konzepten ist gemeinsam, dass sie

- … jedenfalls *Unterstützung direkt durch die Hardware benötigen* (d.h. spezielle Maschinenbefehle), und in den vielen Fällen auch durch das Betriebssystem. Damit geht ein Teil des Geschwindigkeitsvorteils von Shared Memory verloren, weil für die Synchronisation der Zugriffe erst recht wieder *Systemaufrufe und Prozesswechsel* nötig sind.
- … *viel schwerer verständlich und nachvollziehbar* als Message Passing sind: Die Programmierung mit Shared Memory ist *äußerst fehleranfällig*, und die Programme verhalten sich in vielen Fällen nicht reproduzierbar bzw. nichtdeterministisch (d.h. Verhalten und Ergebnis hängen von zeitlichen Zufällen ab). Zu den klassischen Problemen gehören ***Deadlocks*** (mehrere Prozesse warten reihum aufeinander, das Programm hängt) und ***Race Conditions*** (das Ergebnis eines Programms ist unterschiedlich, je nachdem, in welcher Reihenfolge mehrere parallele Abläufe auf dieselben Daten zugreifen).

Zu den Konzepten und Problematiken von Shared Memory gibt es *separate Unterlagen*.