Betriebssysteme

Susanne Schaller, MMSc

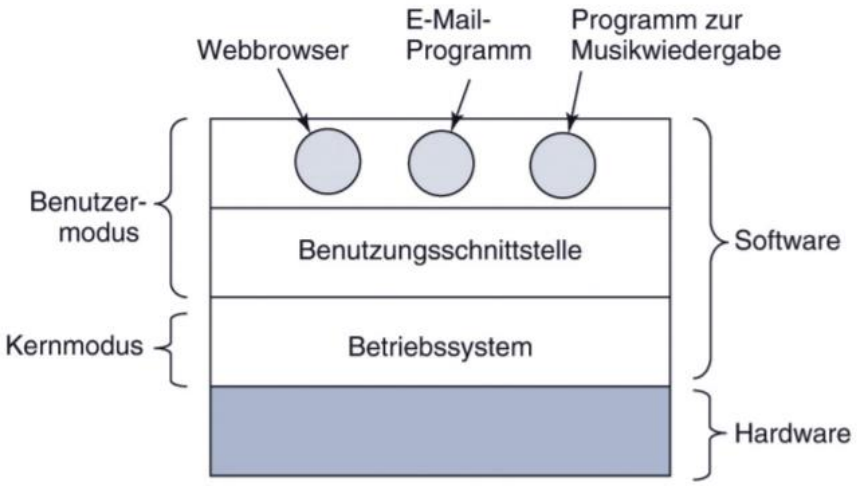
Andreas Scheibenpflug, MSc

SE Bachelor (BB/VZ), 2. Semester

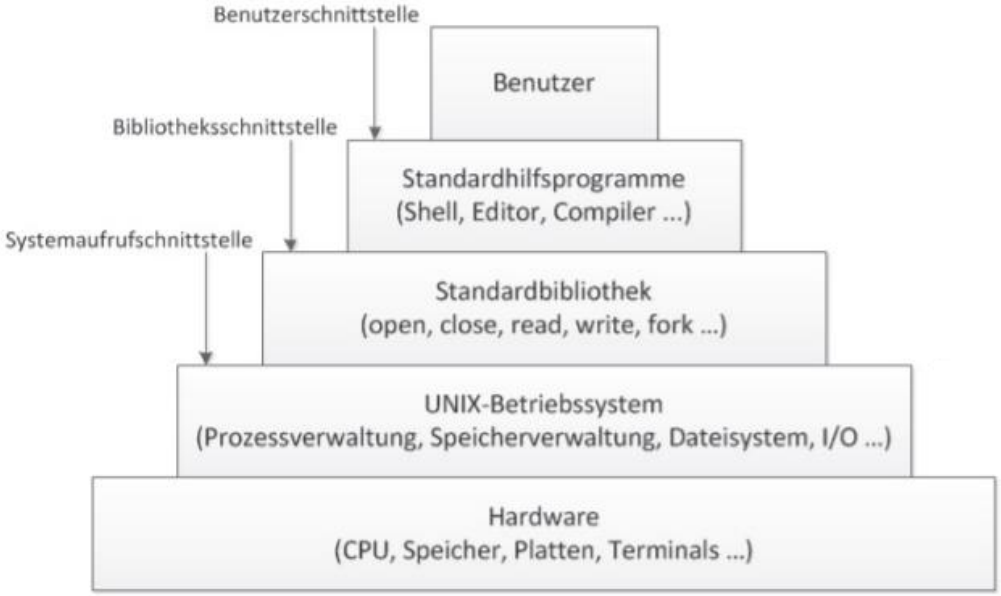
Was ist eigentlich ein Betriebssystem?

- Moderner Rechner = komplexes System
- Betriebssystem (operating system, OS) gehört zu den **komplexesten Softwaresystemen** überhaupt
- Software Engineer sollte Grundverständnis eines Betriebssystems & eines Rechners haben
- OS liegt über der blanken Hardware
- OS bildet die Basis für die gesamte übrige Software
- OS = grundlegende Software im **Kernmodus**

Lage des Betriebssystems



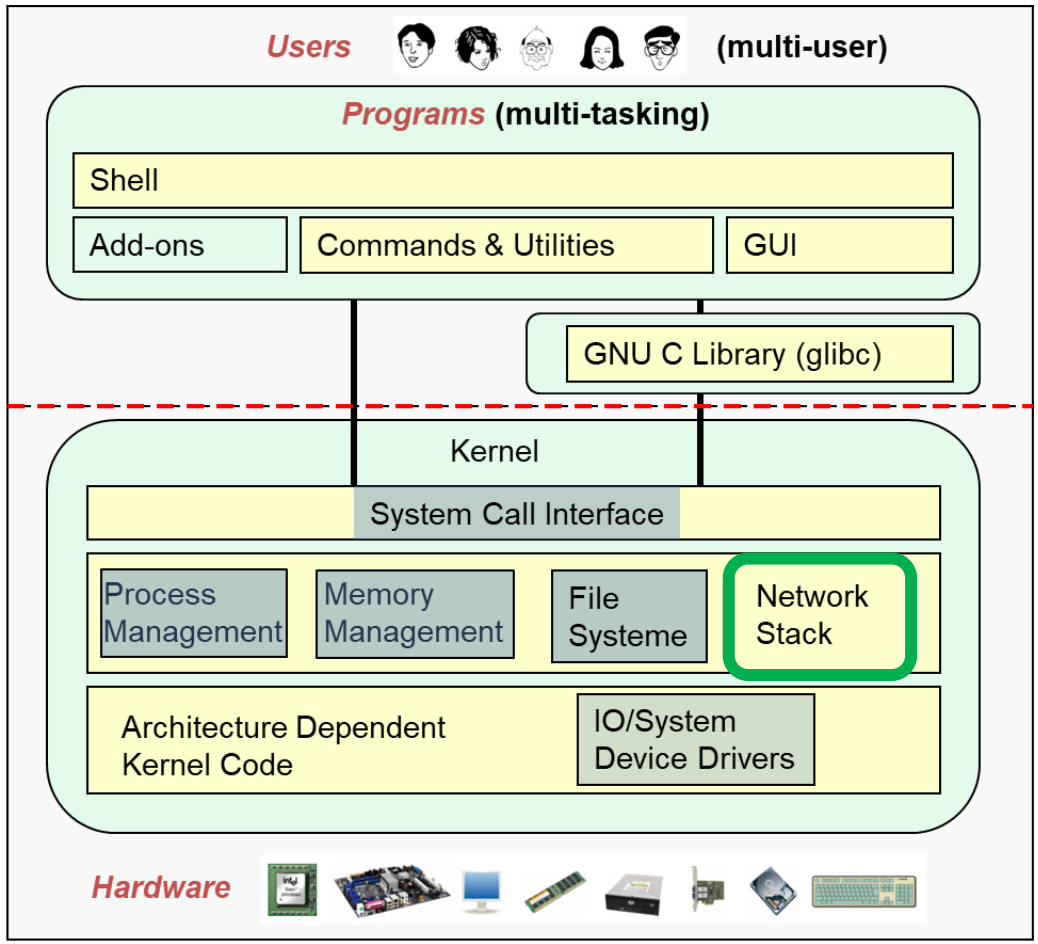
Allgemeine Einordnung eines Betriebssystems. [1]



Schichtenmodell von UNIX. [2]

Quellen:
[1] Andrew S. Tanenbaum, Moderne Betriebssysteme, 4. Auflage
[2] Erich Ehses et al., Systemprogrammierung in UNIX/Linux

Betriebssystem Architektur



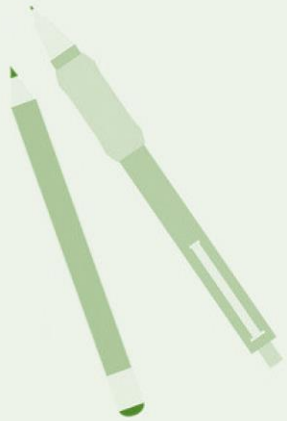
Richard Stallman started the GNU project in 1983 to create a free UNIX-like OS. He Founded the Free Software Foundation in 1985. In 1989 he wrote the first version of the GNU General Public License

Linus Torvalds, as a student, initially conceived and assembled the Linux kernel in 1991. The kernel was later re-licensed under the GNU General Public License in 1992.

Allgemeines

Prof. Gerhard Jahn

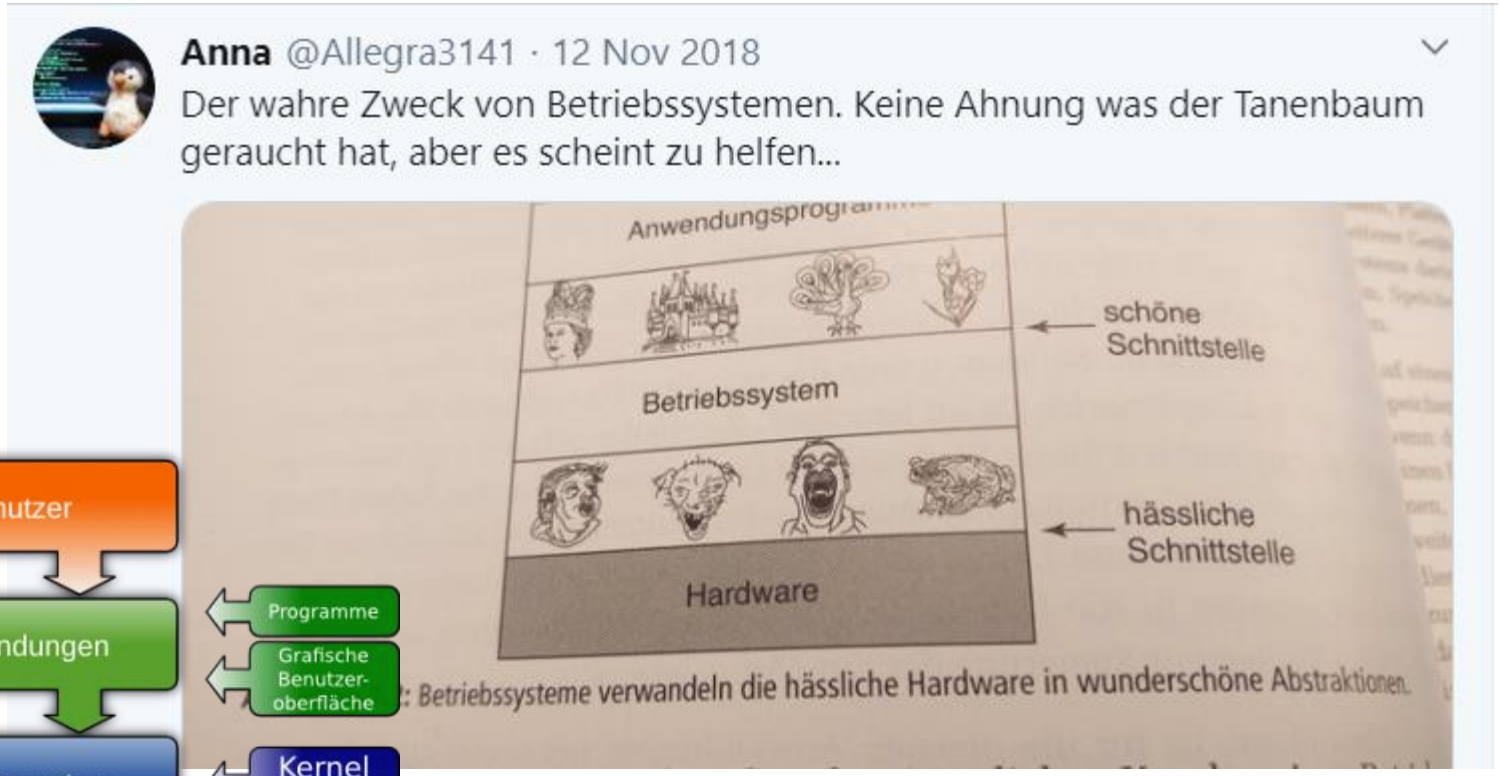
Hauptaufgaben eines Betriebssystems?

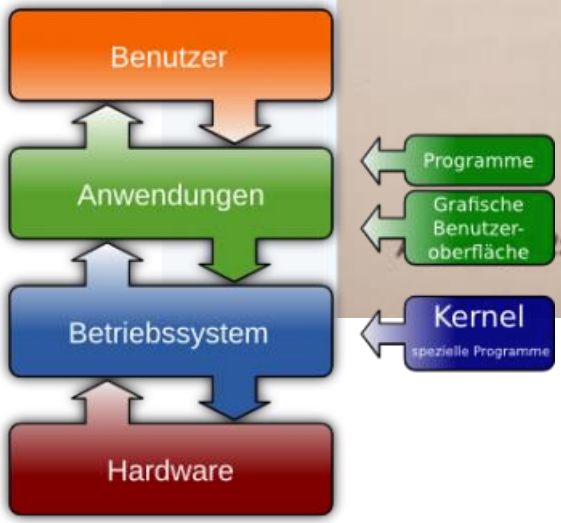


OS Aufgaben

Gezeichnet mit
excalidraw.com

Betriebssystem





Top-down-Sicht vs. Bottom-up-Sicht bei Betriebssystemen!

Größe eines Betriebssystems

- Quellcode ~ 5 Millionen Codezeilen
- Langlebig, schwer zu schreiben
- Vage Definition: „Software, die im Kernmodus läuft“
- Betriebssysteme übernehmen **zwei Aufgaben**:
  - Anwendungsprogrammierer*innen saubere Abstraktionen der Betriebsmittel
  - Hardwareressourcen zu verwalten
  - „Hardwareabstraktionsschicht“ (1. Semester EIB)
- OS enthält viele unterschiedliche Treiber zur Steuerung von Ein-/Ausgabegeräten
- OS als Abstraktionsebene: Hardware/Treiber, Dateien, System Calls,…!

Einsatzbereiche

- Desktop
- Server
  - Fileserver, Webserver, Mailserver, DB-Server
- Netze und Sicherheit
  - Firewalls, Gateways, Router
- Großrechner
  - Rechenzentren
  - Optimiert auf Zuverlässigkeit und hohen Datendurchsatz
  - Banken, Versicherungen und große Unternehmen
- Computercluster
- Supercomputer

Betriebssystemfamilie

- Insgesamt kann man in 9 Betriebssystemarten einteilen
- 1. Betriebssysteme für Großrechner:
  - Sehr hohe Ein-/Ausgabeleistung
  - Großrechner: 1000 Festplatten, Millionen von Gigabytes an Daten
  - Großrechner erleben im Moment Comeback als hoch entwickelte Webserver, Server für den E-Commerce, …
  - Betriebssysteme für Großrechner sind darauf ausgelegt viele Prozesse gleichzeitig auszuführen
  - Bieten 3 Arten der Prozessverwaltung:
    - Stapelverarbeitung
    - Dialogverarbeitung
    - TimeSharing

Betriebssystemfamilie

- 2. Betriebssysteme für Server:
  - Laufen auf Servern, große PCs, Workstations oder sogar Großrechner
  - Bedienen gleichzeitig viele Benutzer über ein Netzwerk
  - Verteilen Hardware- und Softwareressourcen an die Anwender
  - Typische Server OS: Solaris, FreeBSD, Linux, Windows Server 201x
- 3. Betriebssysteme für Multiprozessorsysteme:
  - Max. Rechenleistung, mehrere Prozessoren
  - Parallelcomputer, Multicomputer
  - Windows als auch Linux können auf Multiprozessorsystemen eingesetzt werden

Betriebssystemfamilie

- 4. Betriebssysteme für PCs
- 5. Betriebssysteme für Handheld-Computer:
  - Tablets, Smartphones, Handheld-Computer
  - Marktführer: Android (Google) und iOS (Apple)

- 6. Betriebssysteme für eingebettete Systeme:
  - Rechensysteme, die andere Geräte steuern
  - Welche Beispiele?
  - Haupteigenschaft: nur vertrauenswürdige Software ausgeführt
  - Bekannte Systeme: Embedded Linux, VxWorks, QNX
- 7. Betriebssysteme für Sensorknoten:
  - Netzwerke von winzigen Sensoren
  - Beispiele: Überwachung von Landesgrenzen, Entdeckung von Waldbränden

Betriebssystemfamilie

- 8. Echtzeitbetriebssysteme:
  - Wichtigster Faktor: Zeit bei Ressourcenvergabe
  - Harte Echtzeitsysteme
  - Weiche Echtzeitsysteme
- 9. Betriebssysteme für Smartcards:
  - Kleinste Betriebssysteme
  - Größe einer Kreditkarte
  - Besitzen eigenen Prozessor
  - Einige Smartcards sind Java-orientiert
  - Rechenleistung und Speicherplatz sehr stark eingeschränkt

Zusammenfassung

- OS sind riesige, komplexe und langlebige Systeme
- Lage Betriebssystem, Hauptaufgaben, Größe eines Betriebssystems
- Geschichtliche Entwicklung (5 Generationen)
- Einsatzbereiche von Betriebssystemen
- Betriebssystemfamilie
- Betriebssystemkonzepte

Alle Bilder sind aus Google Bildersuche und teilweise mit Hyperlinks verknüpft bzw. hinterlegt. Diese Folien sind nicht für den öffentlichen Gebrauch verwendbar.

Literatur

- Interesse geweckt?
- A. S. Tanenbaum, H. Bos: Moderne Betriebssysteme (2016)
- P. Mandl: Grundkurs Betriebssysteme: Architekturen, Betriebsmittelverwaltung, Synchronisation, Prozesskommunikation (2010)
- Tolle Übersicht zum Lernen: https://de.wikipedia.org/wiki/Betriebssystem (obwohl Wikipedia, trotzdem hilfreich)
- L. Torvalds, D. Diamond: Just for Fun: The Story of an Accidental Revolutionary (2002)
- W. Isaacson, S. Jobs: Steve Jobs (2011)
- R. C. Alexander, D. K. Smith: Fumbling the Future: How Xerox Invented, Then Ignored, the First Personal Computer (1999)

Dateisysteme

Susanne Schaller, MMSc

Andreas Scheibenpflug, MSc

SE Bachelor (BB/VZ), 2. Semester

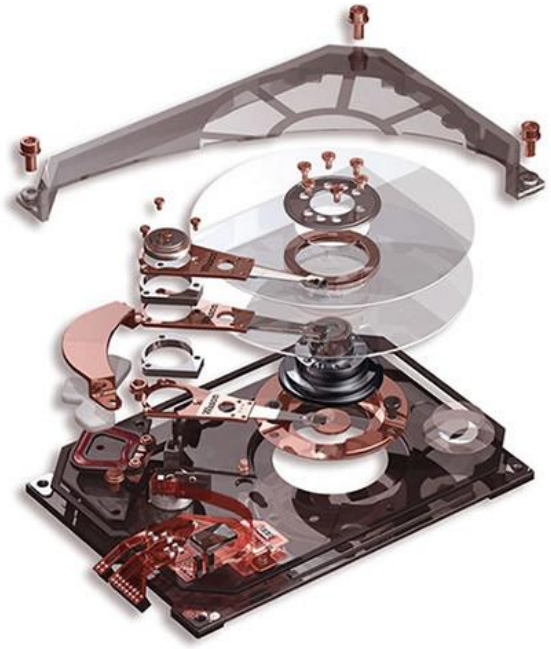
Übersicht

- Festplatten: Allgemeines
  - Diskgeometrie HDD & SSD
- Begriffe: Partitionierung, Blöcke, Cluster
- Dateisystem
  - Aufgabe
  - Dateien vs. Verzeichnisse (logische Sicht)
  - Dateiverwaltung
  - Layout eines generischen Dateisystems
  - Implementierung von Dateisystemen
  - Beispiele von Dateisystemen

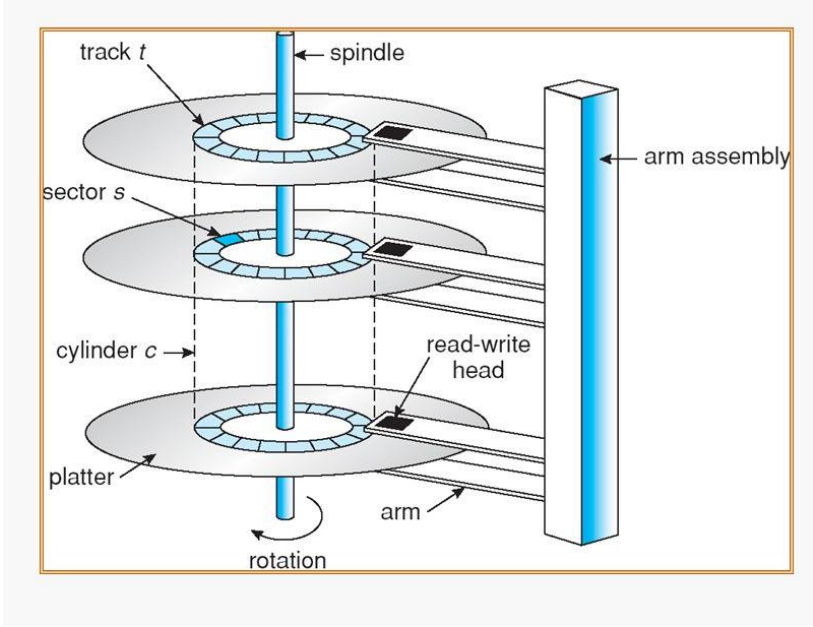
Was ist ein Dateisystem?

- Schnittstelle zwischen dem Betriebssystem und den Partitionen auf den Datenträgern
- Ablageorganisation
- Aufgabe: **Dateien** sollen gespeichert werden können, sollen leicht wieder gefunden werden und sicher abgespeichert werden.

Diskgeometrie HDD

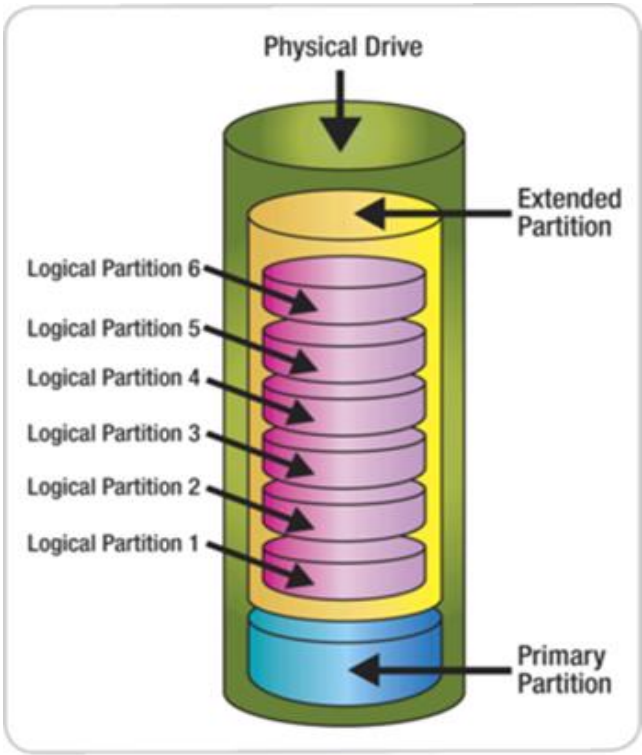


- Mechanisches Speichermedium
- Daten werden auf die rotierende Scheiben geschrieben
- Besteht aus Sektoren, Zylindern, Sektoradressen, Schreib-Lese-Kopf
- Entwickler IBM
- Vorgänger: Magnetband (siehe Geschichte)



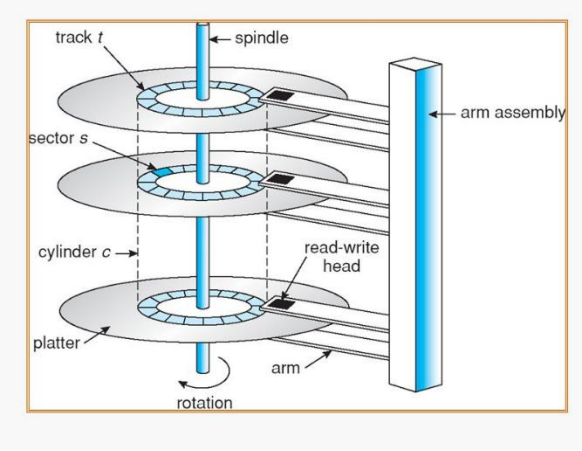
Partitionen und Partitionstypen

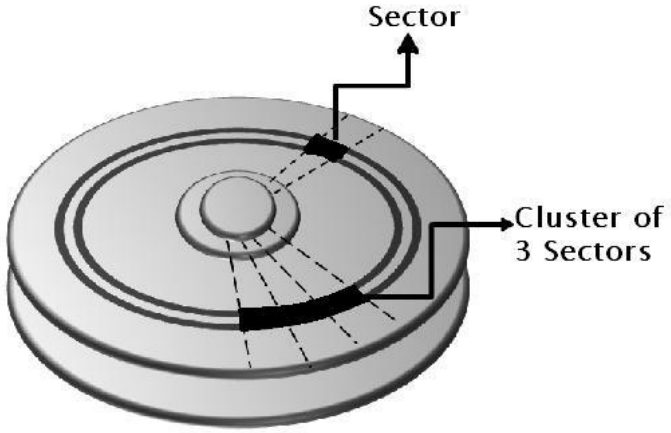
- Partitionen:
  - Einteilung einer Harddisk in einen oder mehrere Bereiche hintereinanderliegender Sektoren
- Partitionstypen (erfolgt mithilfe eines Partitionierungsprogramms)
  - Primäre Partition:
    - Kann nicht mehr weiter unterteilt werden
  - Erweiterte Partition:
    - Kann alleine nicht benutzt werden
    - Unterteilung in logische Laufwerke
- Maximal 4 primäre Partitionen oder 3 primäre und 1 erweiterte Partition sind möglich
- Erweiterte Partitionen können in sogenannte logische Laufwerke unterteilt werden



Blöcke / Cluster an HDD

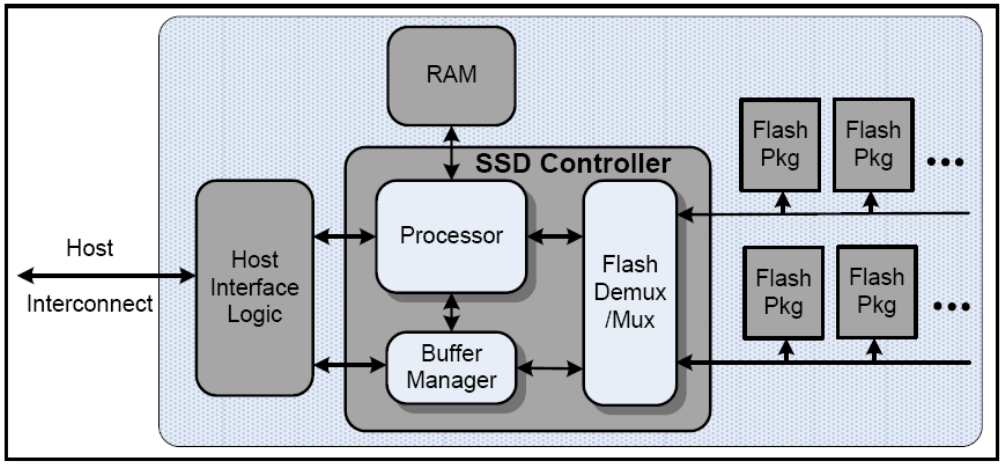
- Mehrere physikalisch aufeinander folgende Sektoren werden zu sogenannten Clustern (Windows) oder Blöcken (Unix) zusammengefasst
- Dadurch können beim Datentransfer die Bewegungen der Lese- / Schreibköpfe reduziert werden.



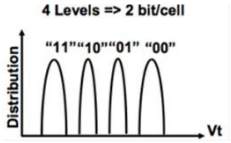


Diskgeometrie SSD

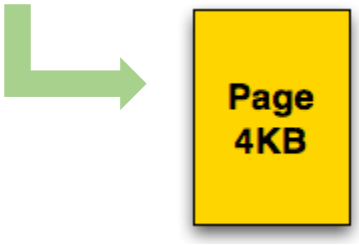
- Speichermedium das wie eine herkömmliche magnetische Festplatte eingebaut und angesprochen werden kann
- Keine rotierenden Scheiben
- Besteht aus Controllerchip, Flash-Bausteinen, Cache
- Flash-Bausteine beinhalten Speicherzellen (NAND-Flash Zellen)



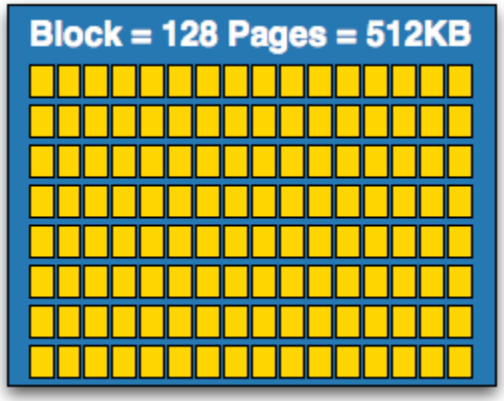
Speichereinheiten SSD - Beispiel



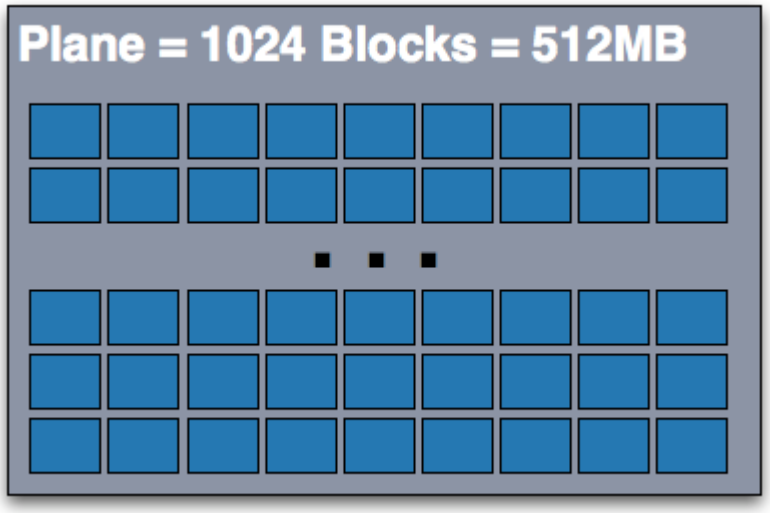
NAND-Flash Zellen werden zu einer Page (4K) gruppiert



Pages werden zu einem Block (512K) gruppiert

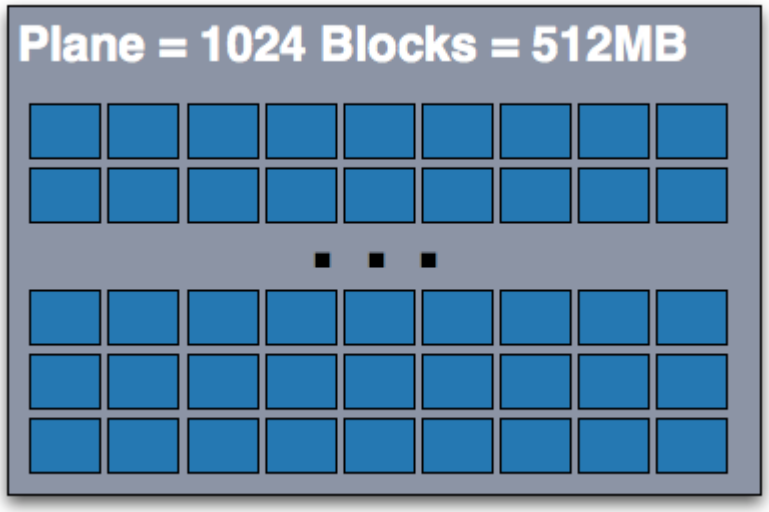


Blocks werden zu einer Plane gruppiert

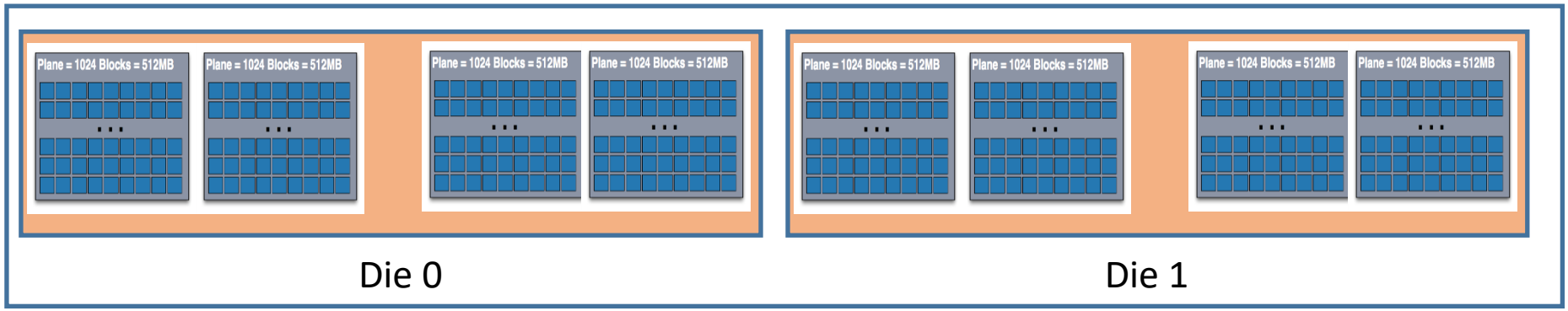


Speichereinheiten SSD

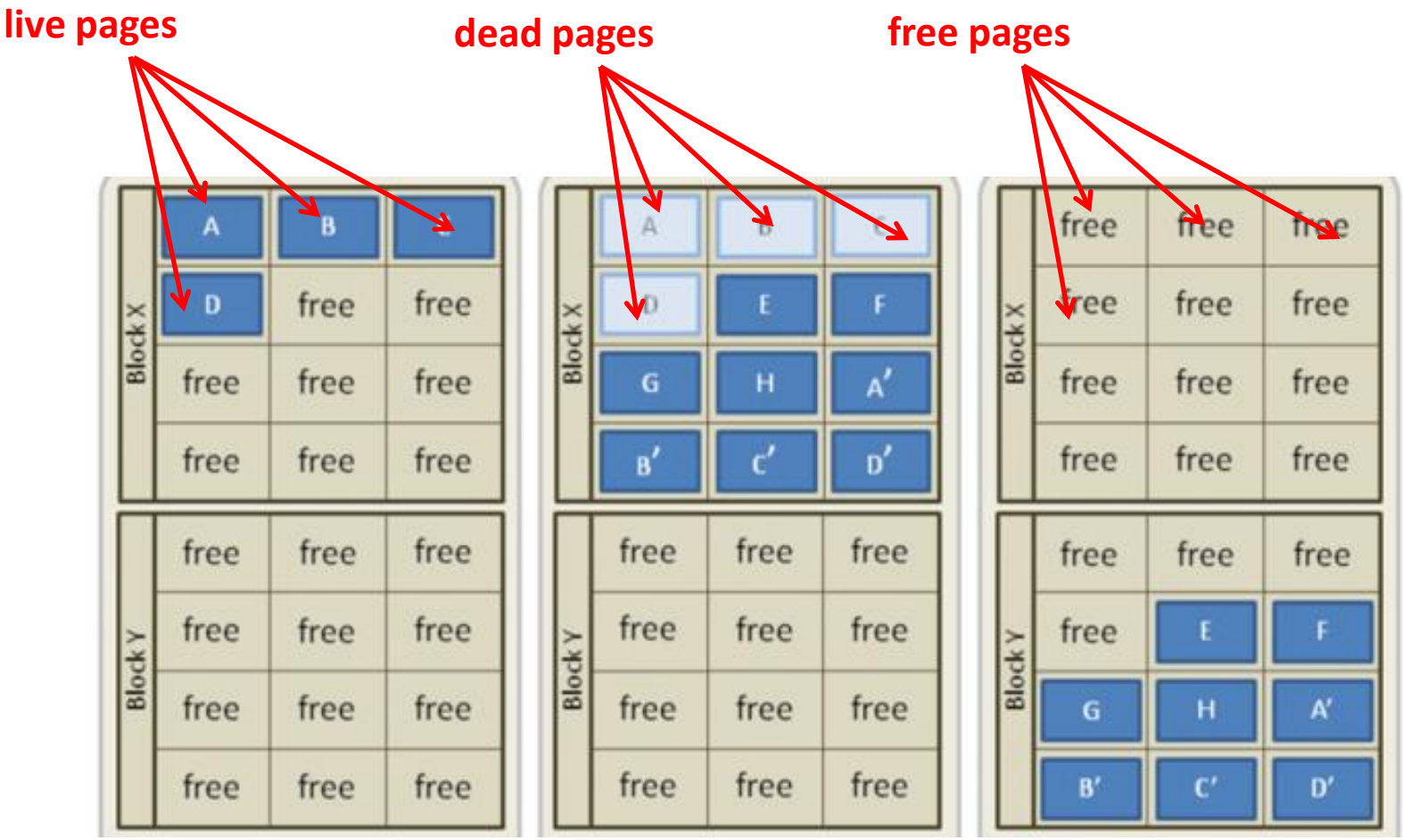
Arrays of Blocks werden
zu einer Plane gruppiert



- Je vier Planes werden zu einem Die gruppiert
- Jedes Die hat 2 GB Speicher
- Zwei Dies ergeben ein 4 GB Flash Package

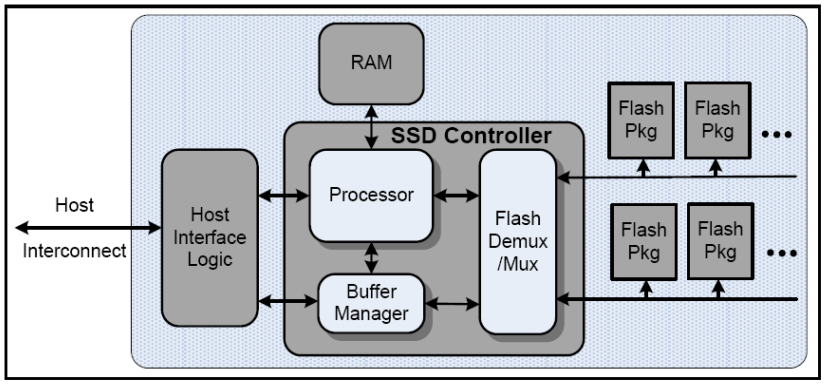


SSD: Zustände Pages



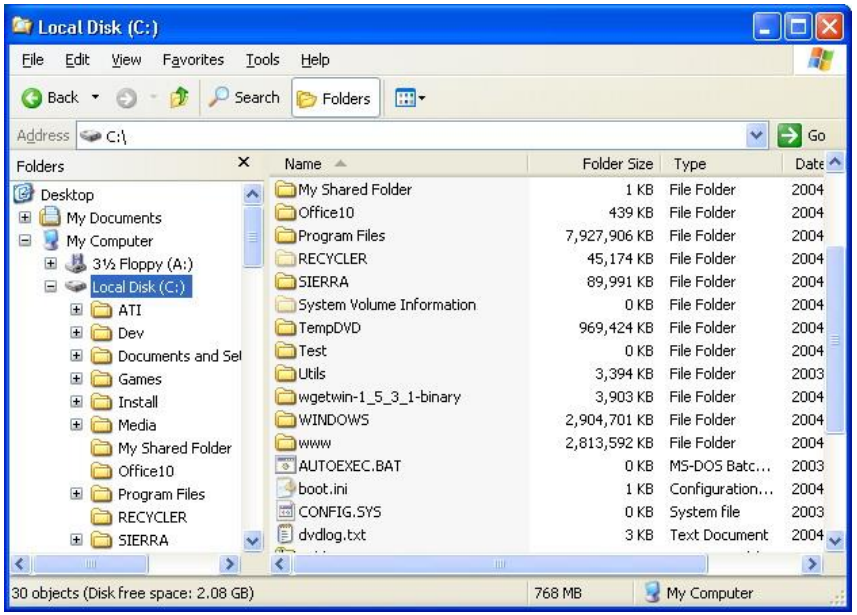
Diskgeometrie SSD

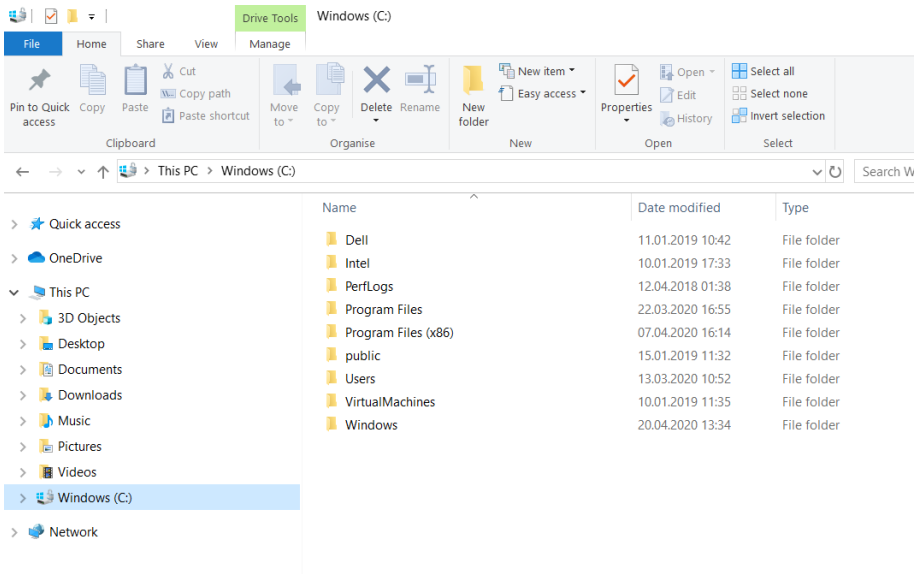
- SSD soll gleichmäßig „altern“
- SSD Controller verwendet Wear Levelling:
  - Umorganisieren der Pages
  - Neu ankommende Daten werden in die bisher am wenigsten beanspruchten Blöcke geschrieben
  - Controller verwendet Tabellen:
    - zur Zuordnung der Blockadressen, die das Dateisystem verwendet
    - zur Zuordnung der tatsächlichen Pageadressen im Flash Speicher (Dateisystem hat keine Ahnung mehr, wo Daten tatsächlich gespeichert sind)



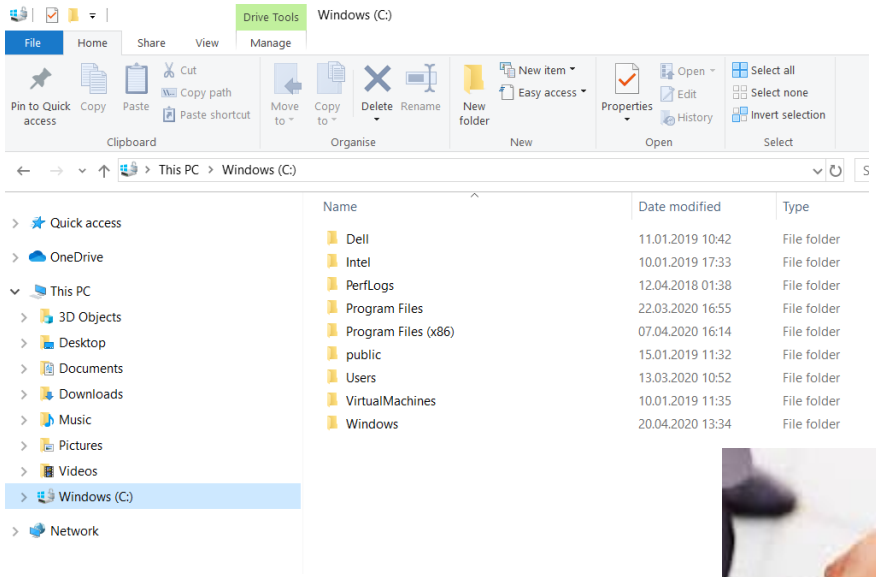
Dateisystem

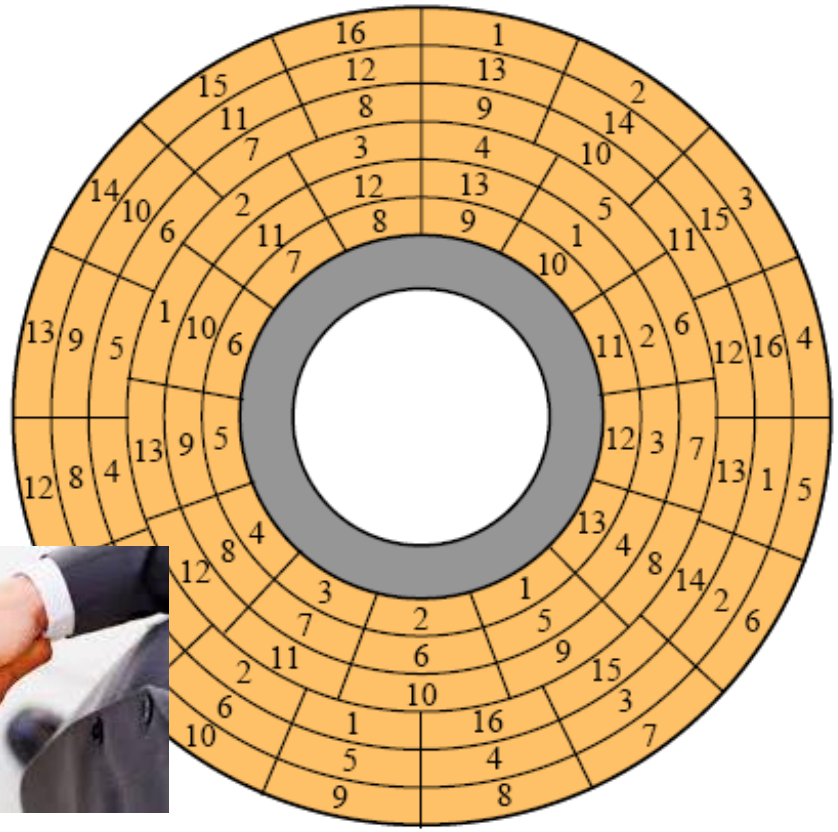
- Nach der Einteilung einer Festplatte in Partitionen muss jede Partition mit einem Dateisystem formatiert werden, bevor sie zur Datenspeicherung benutzt werden kann.
- Dateisystem kann sich auf eine Verzeichnisstruktur beziehen oder auf Partitionen
- Dateisysteme benutzen Datenstrukturen und Funktionen zur persistenten Speicherung der Daten





Physische Sicht des Dateisystems

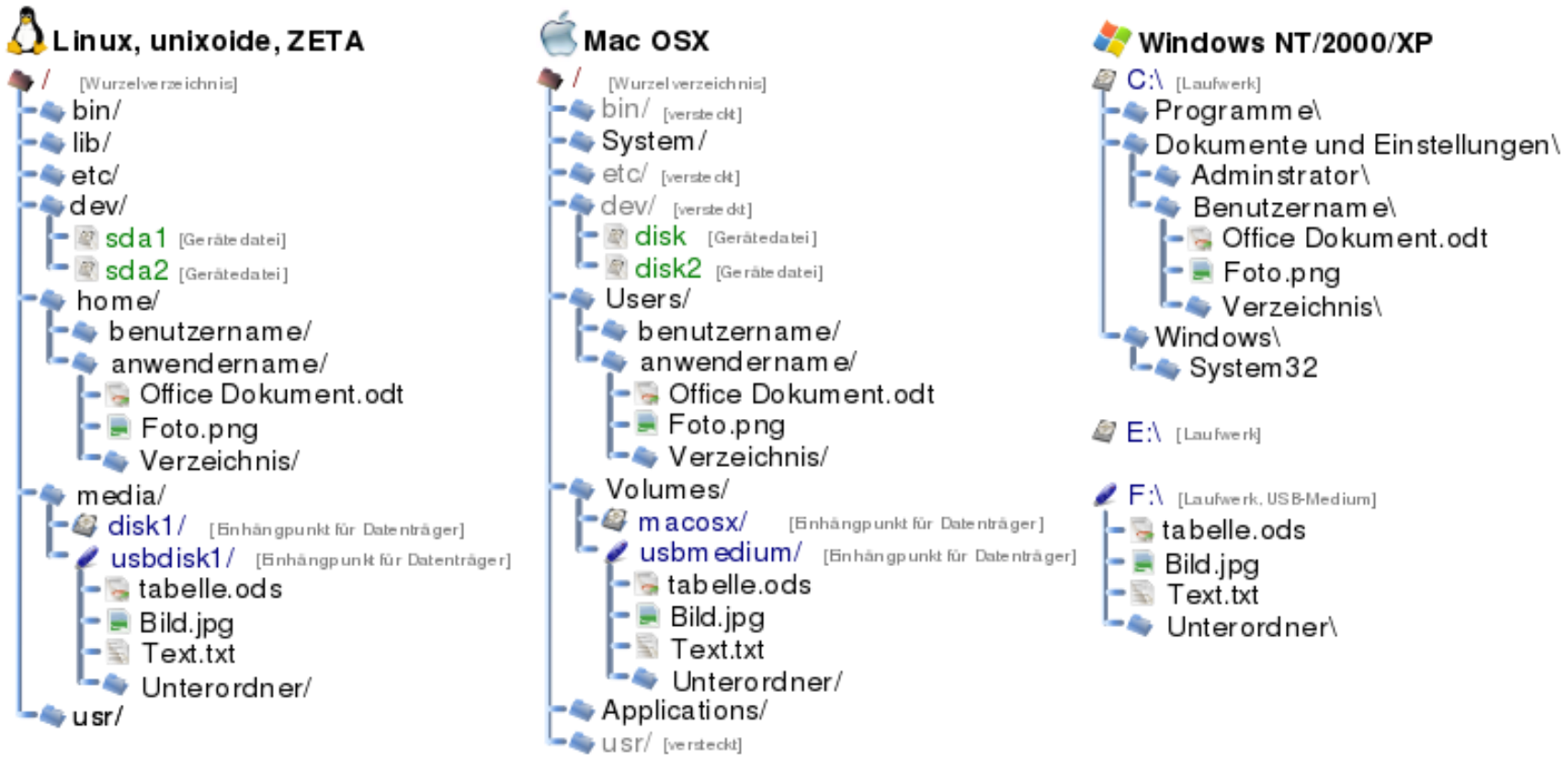




Aufgabe eines Dateisystems

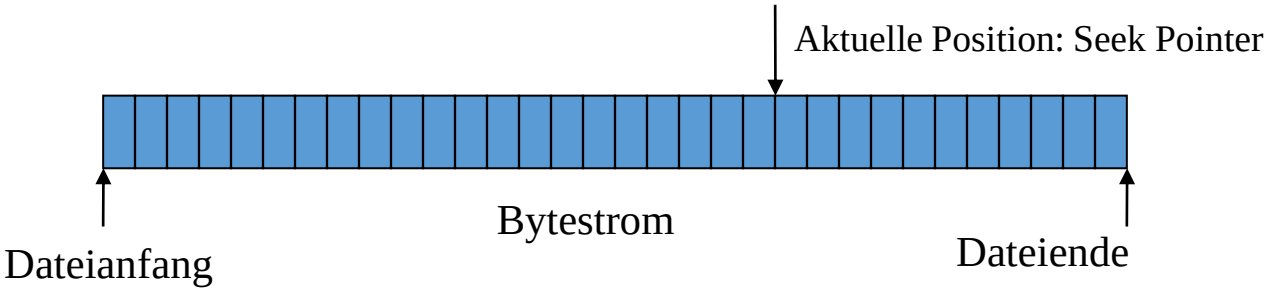
- Dateisystem ist das am meisten sichtbare Konzept eines Betriebssystems (siehe Explorer)
- Es verwendet folgende Objekte:
  - Dateien
  - Verzeichnisse zur Strukturierung von Dateien
  - Laufwerke für Partitionen (Windows)
- Persistente Speicherung großer Informationsmengen
- Gleichzeitiger Zugriff auf Daten durch mehrere Prozesse (siehe Prozessmanagement) muss möglich und geregelt sein
- Berechtigungskonzept für den Zugriff auf Daten ist notwendig

Logische Sicht des Dateisystems



Logische Sicht: Datei

- Jede Datei besteht aus:
  - Dateiattribute (Metadaten): Datum, Zeitpunkt, Dateiname, Dateierweiterung (Windows: Zuordnung Programm, mit dem die Datei zu bearbeiten ist; Unix: Endung nur eine Konvention), Dateigröße (in Bytes), Benutzer- und Zugriffsinformationen, Markierungsbits oder Flags
  - Nutzdaten: werden als Bytestrom gespeichert
  - Struktur des Bytestroms selbst ist für das Dateisystem nicht relevant, sondern nur für die Anwendung, die die Nutzdaten organisiert
  - Dateizugriff: sequenziell (früher), zufällig



Dateitypen

- Reguläre Dateien (regular file)
- Verzeichnisse (directory)
- Zeichendateien (character special file)
  - Device File für direkte, serielle Kommunikation mit Geräten (meist nicht gecached), z.B. USB
- Blockdateien (special block file)
  - Device File für blockorientierte Geräte (Daten werden im OS gecached und in Blöcken (parallel) übertragen), z.B. Festplatten

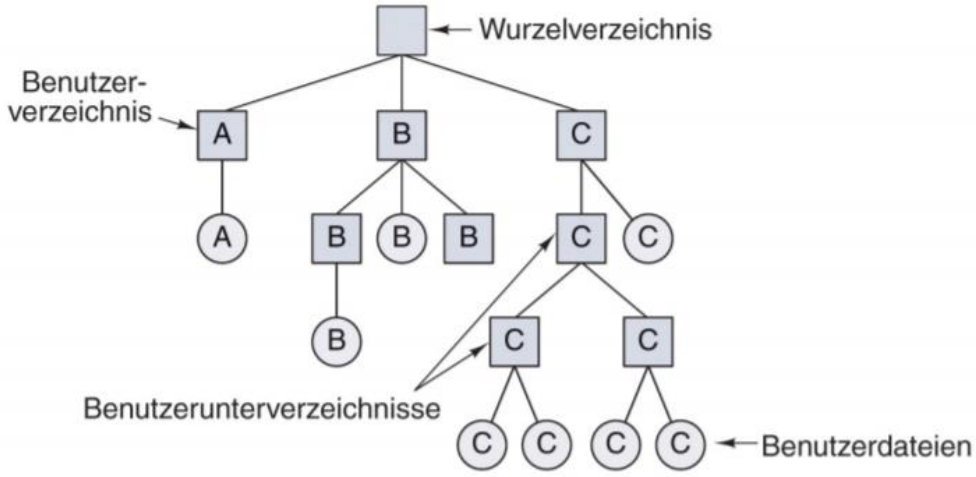
- Dateien existieren, um Informationen zu speichern, die später wieder abgerufen werden. Dafür stehen unterschiedliche Operationen für Speicherung und Abfrage zur Verfügung!

Dateinamen

- MS-DOS: Dateinamen bestehen aus
  - Dateinamen (bis zu 8 Zeichen, beginnend mit Buchstaben)
  - Dateierweiterung: bis zu 3 Zeichen
  - Keine Unterscheidung zwischen Groß- und Kleinschreibung
  - z.B.: PROG.C, HALLO.TXT
- UNIX / Linux: Dateien bestehen aus
  - Bis zu 255 beliebige Zeichen
  - Dateierweiterung: nur Konvention, kein Bestandteil der Namensregeln
  - Sehr wohl Unterscheidung zwischen Groß- und Kleinschreibung
  - z.B.: prog.c, Prog.c, prog.c.Z, etc.

Logische Sicht: Verzeichnisse

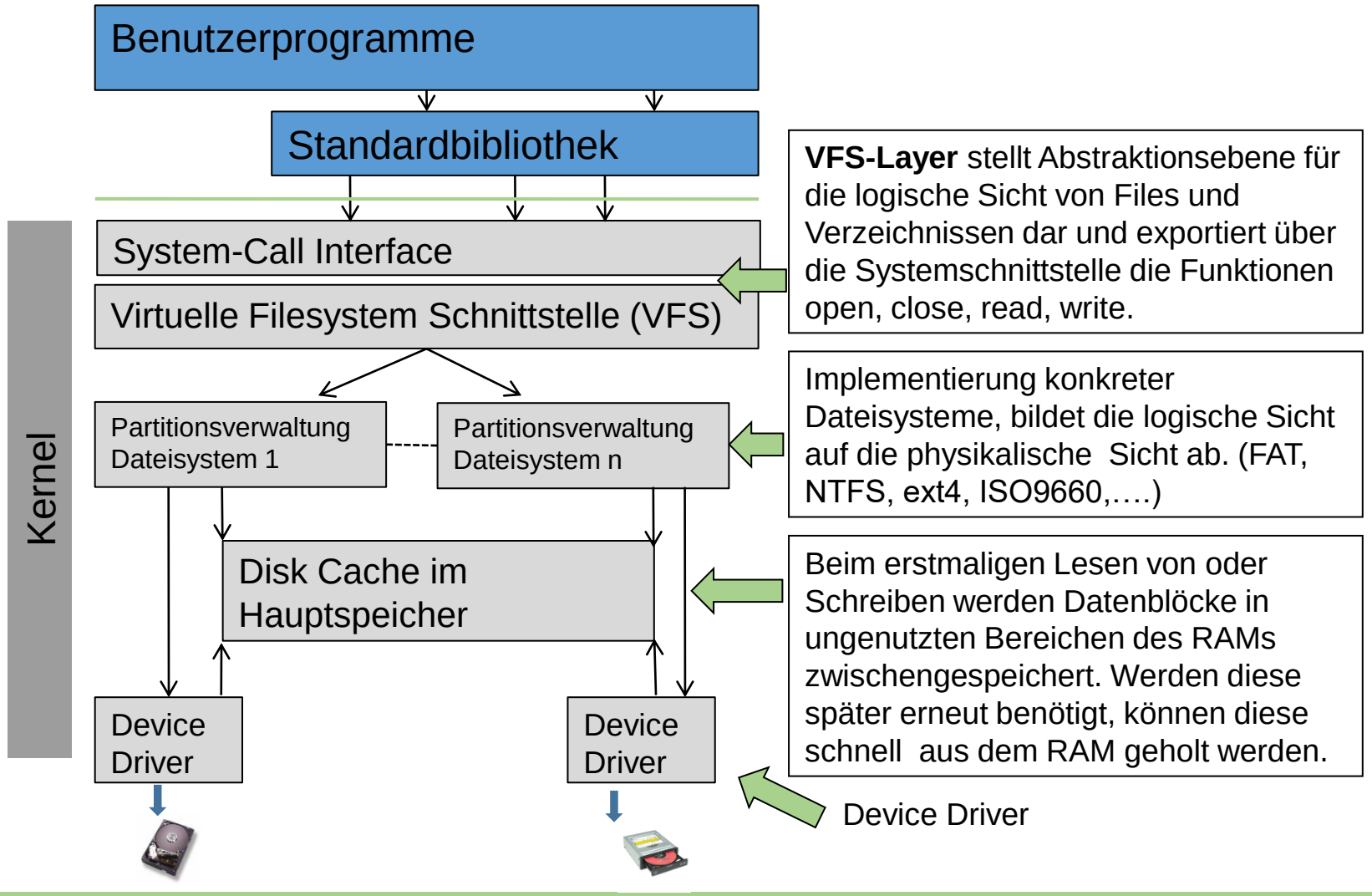
- Verzeichnisse werden verwendet zur thematischen Gruppierung von Dateien durch den Benutzer oder dem Betriebssystem
- Hierarchische Verzeichnissysteme
- Verzeichnisse können Unterverzeichnisse enthalten, dadurch entsteht eine Baumstruktur
- Verzeichnisse werden von verschiedenen Dateisystemen mit unterschiedlichen Datenstrukturen implementiert



Dateiverwaltung als Service für OS

- Folgende Operationen werden zur Verfügung gestellt (vom System Call Interface des OS):
  - Datei Funktionen: create, delete, open, close, read, write, append, seek, rename
  - Verzeichnis Funktionen: create, delete, opendir, closedir, rename
- Verwaltungsfunktionen:
  - Formatierung: Erzeugen eines leeren Dateisystems auf einer Partition
  - Konsistenzprüfung beim Rechnerstart
  - Backup Funktionen für Datensicherung
  - Falls gewünscht: Komprimierung und Verschlüsselung der Dateien

Schichten des Dateisystems im Kernel



Partition: Layout eines generischen Dateisystems

Partition mit Sektoren

Für alle Bereiche sind entsprechende Daten-Strukturen vorzusehen, die über die Sektoren der Partition zu legen sind.

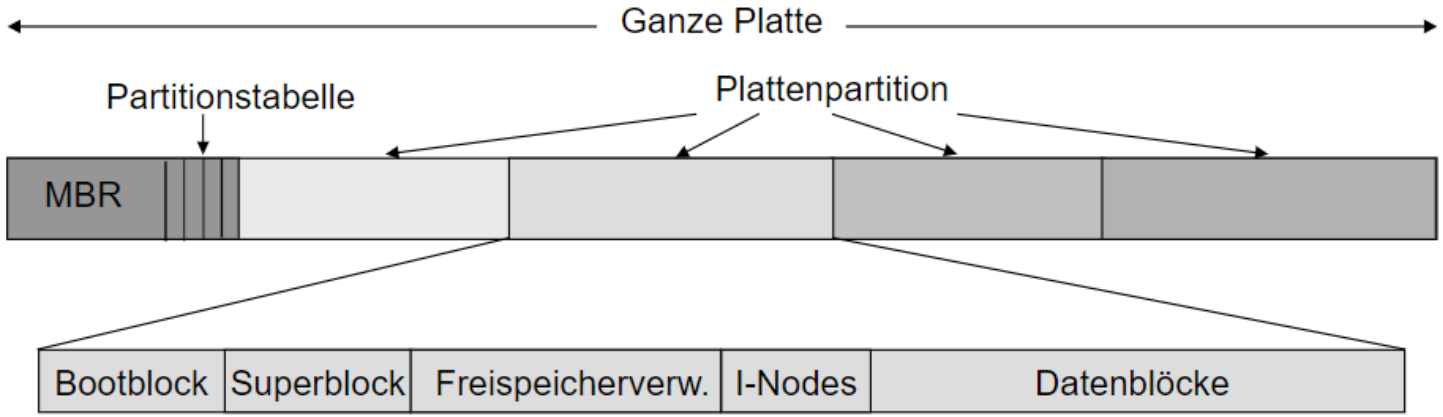
Exkurs: Was passiert beim Hochfahren eines Computers?

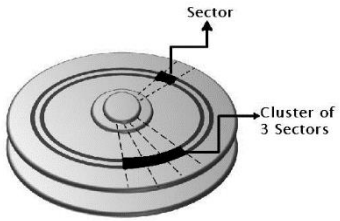
- Hochfahren an sich ist ein Prozess = Booten
- PC: besitzt Motherboard und darauf Programm: BIOS
- BIOS: Programm und enthält Ein-/Ausgabesoftware
- BIOS bestimmt das Gerät, von dem das Betriebssystem geladen werden soll

- Default: BIOS lädt MBR

Layout eines möglichen Dateisystems

- Platte: Sektor 0 reserviert für MBR
- MBR = Programm, lokalisiert als Erstes die aktive Partition, liest deren ersten Block, den Boot-Block und führt ihn aus, Boot-Block enthält OS → Start des OS!



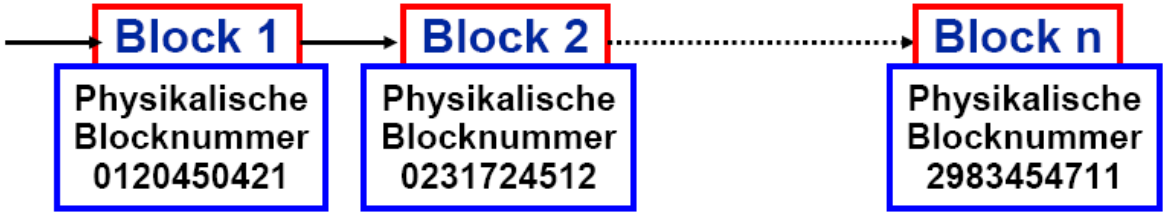


Was passiert beim Hochfahren eines Computers?

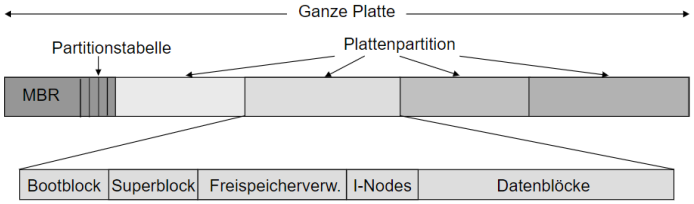
- Wenn BIOS Festplatte bestimmt hat, dann Sektor 0, MBR und Partitionstabelle nachgeschaut was als aktive Partition eingestellt ist
- Nächstes Programm: Bootlader
- Bootlader liest dann das OS von der aktiven Partition ein und startet es
- OS fragt BIOS nach Konfigurationseinstellungen
- Für jedes gefundene Gerät wird nach Gerätetreibern gesucht
- Wenn alle Treiber gefunden wurden, lädt das OS diese in den Kern
- Danach: Interne Tabellen initialisiert, nötige Hintergrundprozesse eingerichtet, Login-Programm bzw. GUI auf den angeschlossenen Terminals gestartet.

Partitionsverwaltung

- Eine Partition stellt für ein Dateisystem ein Array von fortlaufend nummerierten Blöcken dar, die frei oder belegt sein können
- Jeder Block kann direkt über seine Logical Block Address (LBA) adressiert werden
- Das Dateisystem muss seine eigene Verwaltungsinformation auf der Partition speichern
- Das Dateisystem sieht eine Datei/Verzeichnis als Folge von Blöcken, die physikalisch nicht aufeinander folgen müssen



Unix-Style Layout Dateisystem



- Nach Bootblock kommt der Superblock
- Superblock:
  - Schlüsselparameter des Dateisystems (Name des Dateisystemtyps, Schutzmarkierungen)
- Danach kommt Freispeicherverwaltung:
  - Information über die freien Blöcke im Dateisystem
- I-Nodes
  - Liste von Datenstrukturen
- Abschluss bilden die Verzeichnisse und Dateien = Datenblöcke

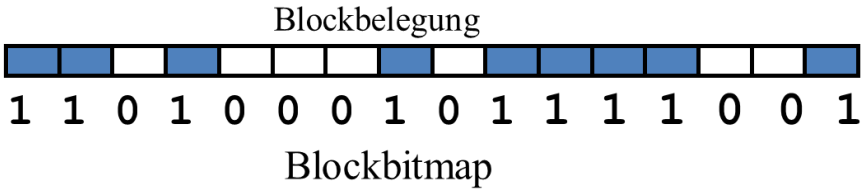
Generisches Dateisystem

Für alle Bereiche sind entsprechende Daten-Strukturen vorzusehen, die über die Sektoren der Partition zu legen sind.

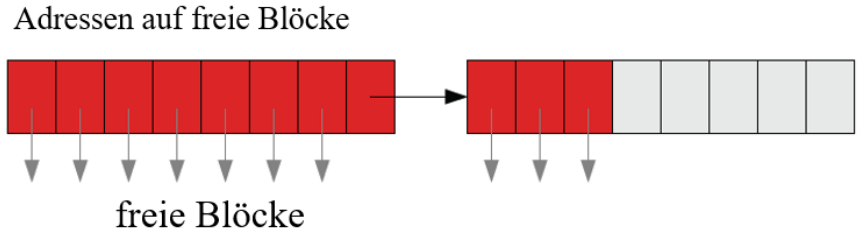
- Allgemeine Informationen des Dateisystems:
  - Kennung des spezifischen Dateisystems, Größe in Blöcken, Dirty-Bit: Indikator, ob eine Überprüfung oder ggf. Reparatur beim Systemstart durchzuführen ist
  - Freie Positionen im Inhaltsverzeichnis
- Inhaltsverzeichnis für Dateien/Verzeichnisse
  - Beinhaltet für jede Datei/Verzeichnis eine Datenstruktur, die Dateiattribute und Referenzen auf Datenblöcke beinhaltet
- Freispeicherverwaltung, Bad-Blocks Verwaltung
- Datenbereich
  - Beinhaltet die eigentlichen Datenblöcke (= Nutzdaten) für Dateien/Verzeichnisse und die leeren Datenblöcke

Freispeicherverwaltung

- Ziel: Welche Plattenblöcke sind mit welchen Dateien assoziiert
- 2 Möglichkeiten:
  - Bitmaps
  - Verkettete Listen



1 – Block/Cluster ist belegt, oder BAD
0 – Block/Cluster ist noch frei

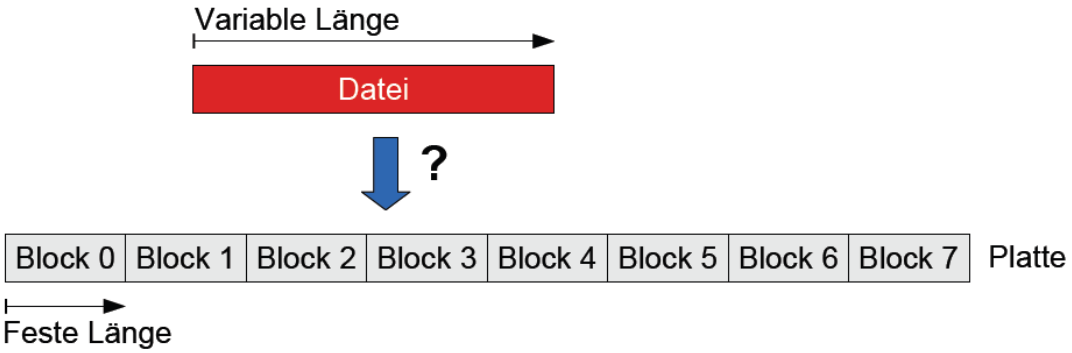


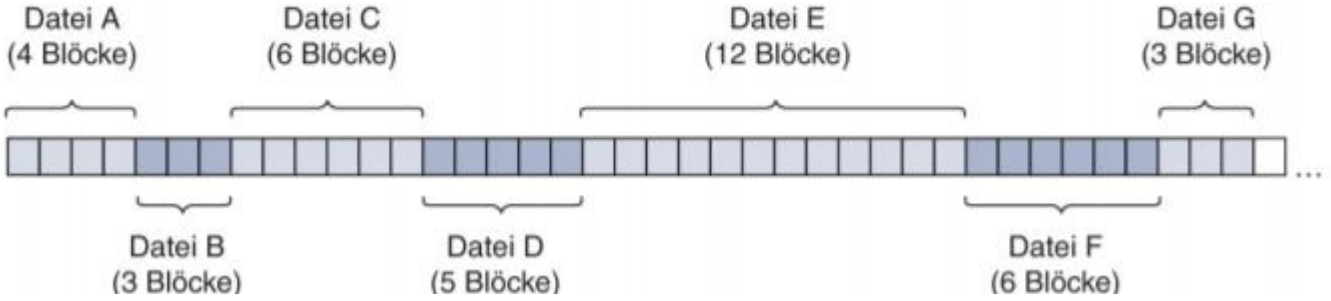
Liste enthält Einträge auf freie Blöcke
Rot – freie Blöcke
Grau – BAD Block (kaputt), oder schon belegt mit Daten

Inhaltsverzeichnis für Dateien/Verzeichnisse

- Dateien bestehen aus mehrere Blöcke und das OS muss verwalten, welche Blöcke zu welchen Dateien gehören bzw. welche Dateien sich in den entsprechenden Verzeichnissen befinden
- Dafür gibt es **Inhaltsverzeichnisse** oder auch **Verzeichniseinträge** genannt
- Es gibt unterschiedliche Implementierungen wie Dateien bzw. Verzeichniseinträge abgespeichert und verwaltet werden:
  - Kontinuierliche Speicherung (contiguous allocation)
  - Verkettete Speicherung (linked allocation)
  - Indizierte Speicherung (indexed allocation)

Kontinuierliche Speicherung

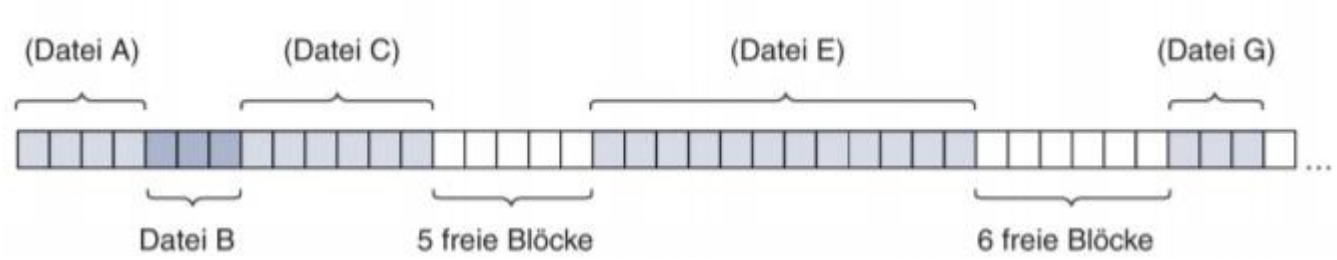




Auszug aus Tanenbaum et. al

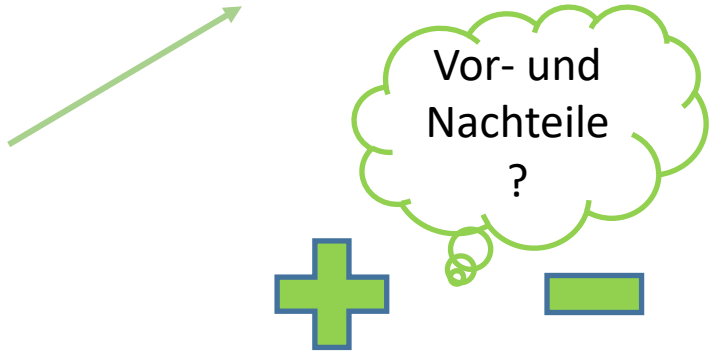
Kontinuierliche Speicherung

- Dateien benötigen oft mehr als nur einen Datenblock:



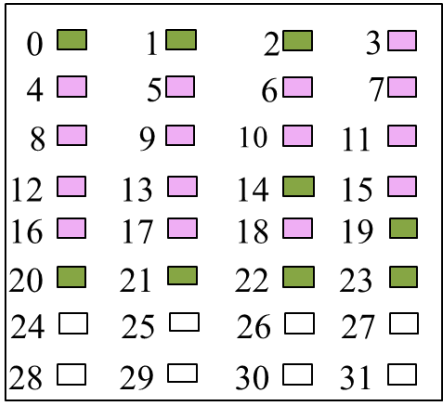
Verzeichnis

Filename	Startblock	Länge
Datei A	0	4
Datei B	4	3
Datei C	7	6
…	…	…



Kontinuierliche Speicherung mit Extents

- Unterteilen einer Datei in Folge von Extents
- Extents werden kontinuierlich gespeichert
- Pro Datei muss Startblock und Länge jedes einzelnen Extents gespeichert werden
- Alle Extents einer Datei werden als Runlist bezeichnet



Filename	Startblock	Länge
Test.doc	0	3
	14	1
	19	5
Doku.dat	3	11
	15	4

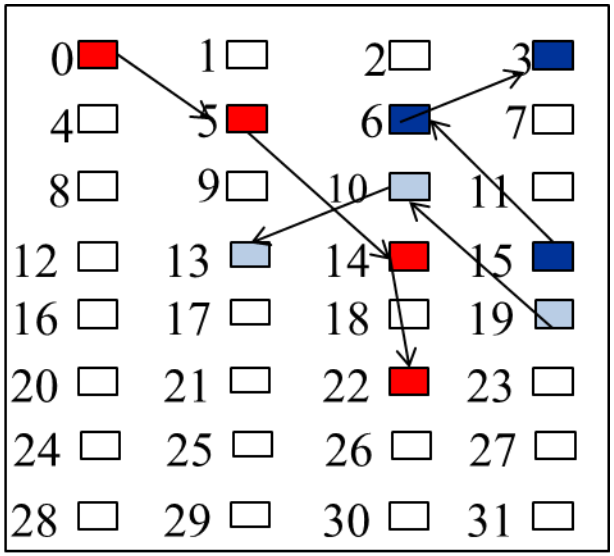
Runlist

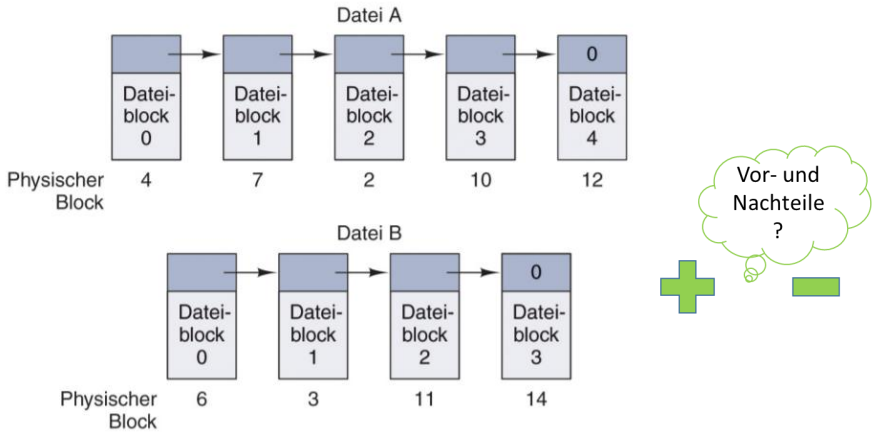
Runlist

Verzeichnis

Verkettete Speicherung

- Beispiel für Blockgröße 512 Bytes:
  - Die ersten oder die letzten vier Bytes im Block bezeichnen die Blocknummer des nächsten Blocks
  - 508 Bytes Nutzdaten + 4 Byte Zeiger auf nächsten Block



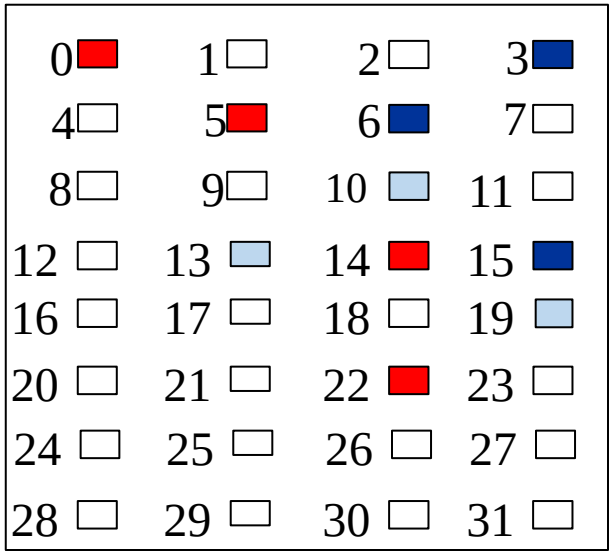


Filename	Startblock	Endblock
file1	0	22
text	15	3
list.dat	19	13

Verzeichnis

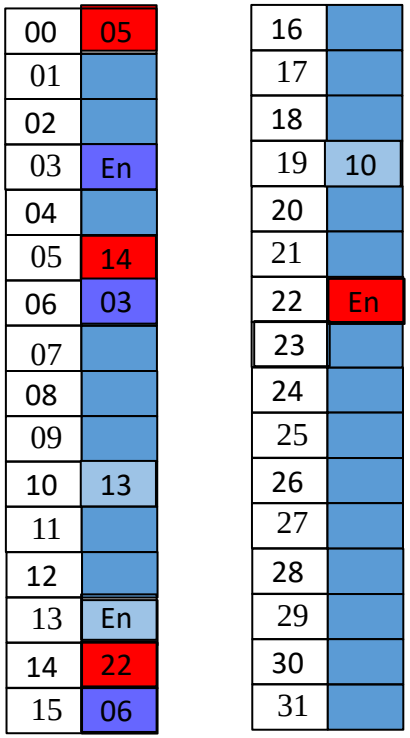
Verkettete Liste mit FAT

- Verkettung aller Dateien wird in speziellen Blöcken ausgelagert – in eine sogenannte File Allocation Table (FAT)

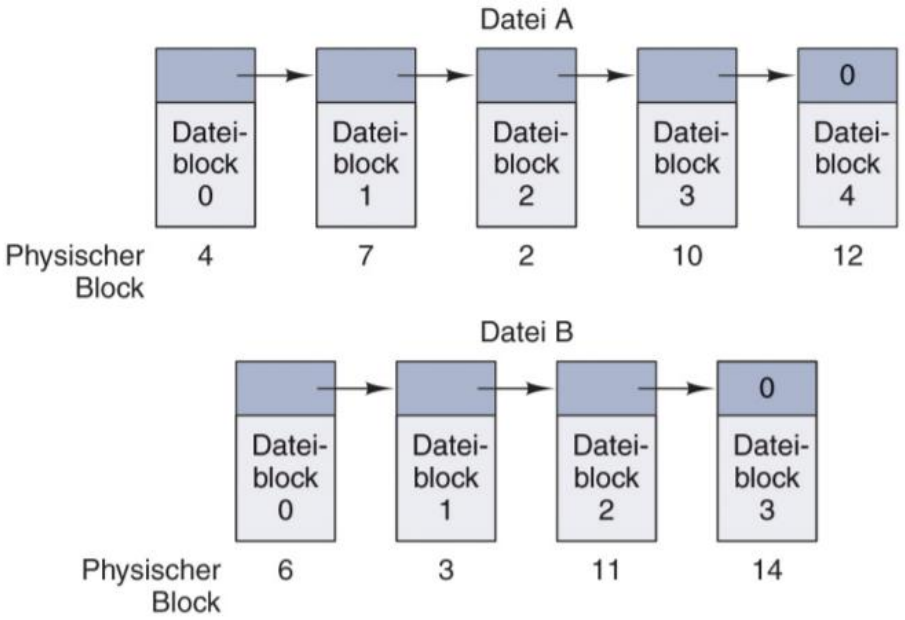


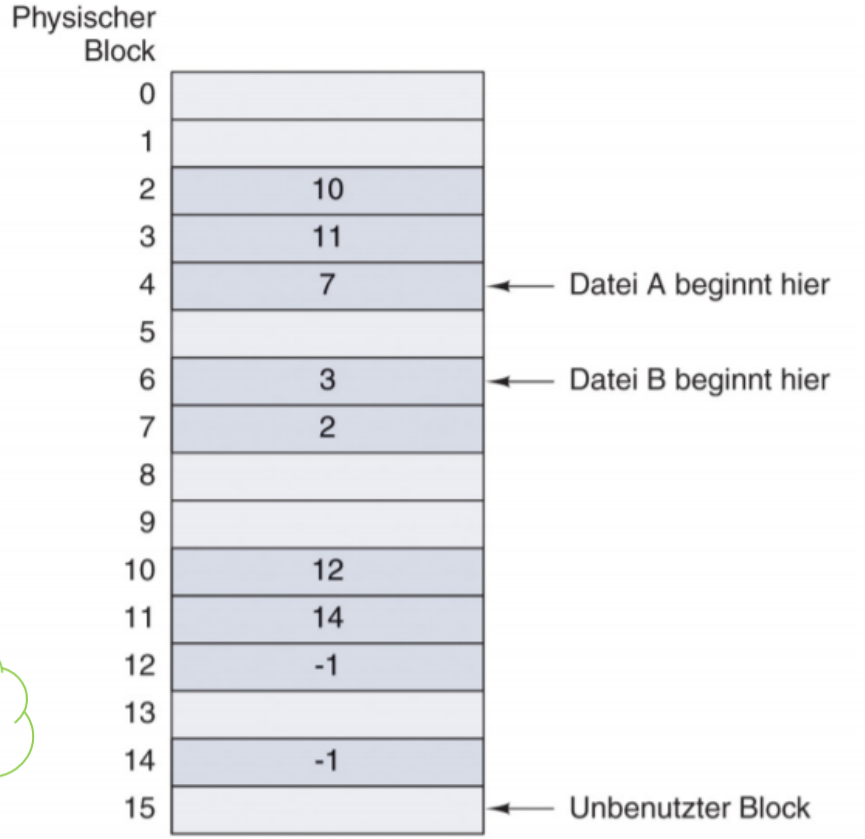
Filename	Startblock
file1	0
text	15
list.dat	19

Verzeichnis



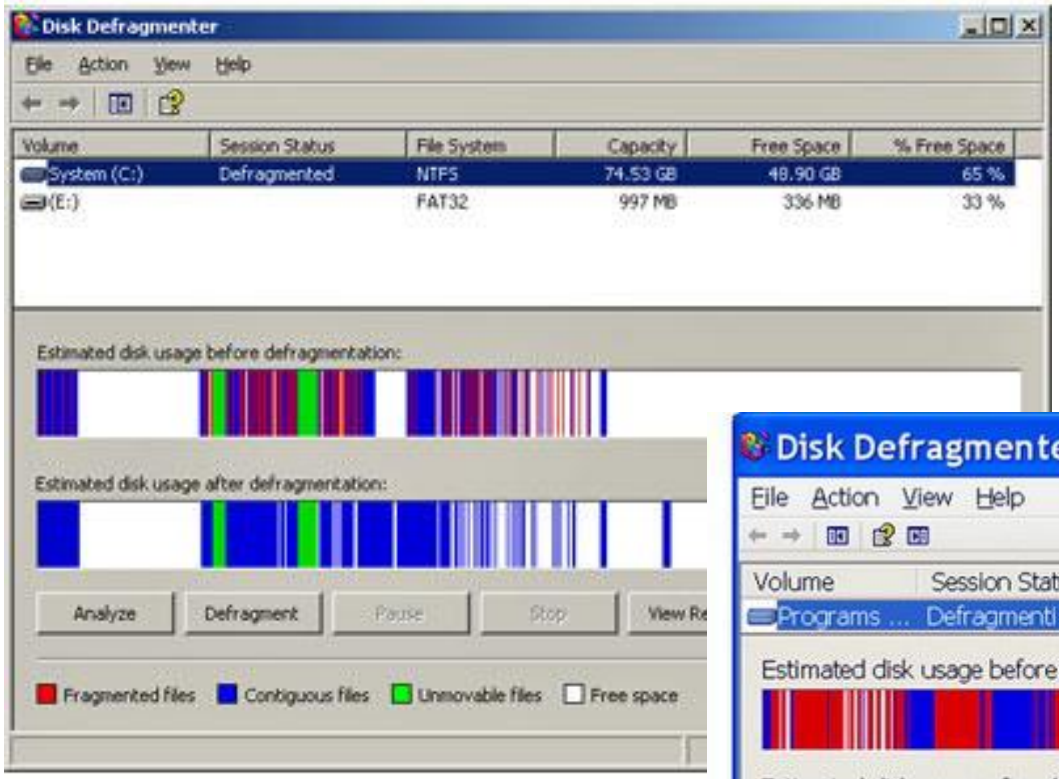
Gegenüberstellung verkettete Liste: mit und ohne Tabelle

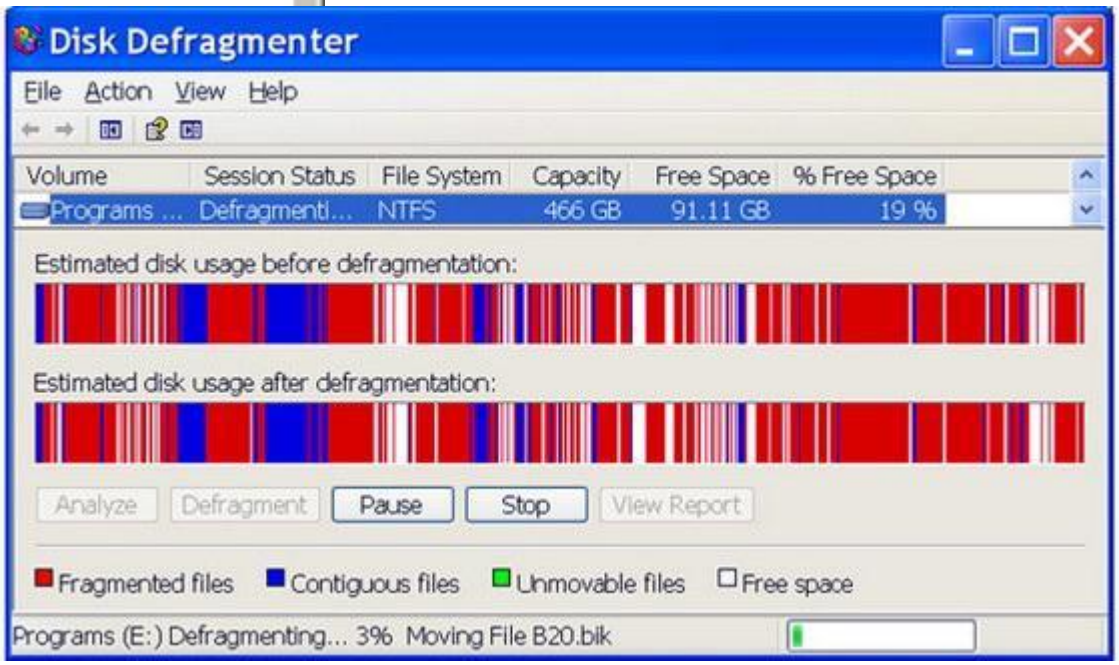




Auszug aus Tanenbaum et. al

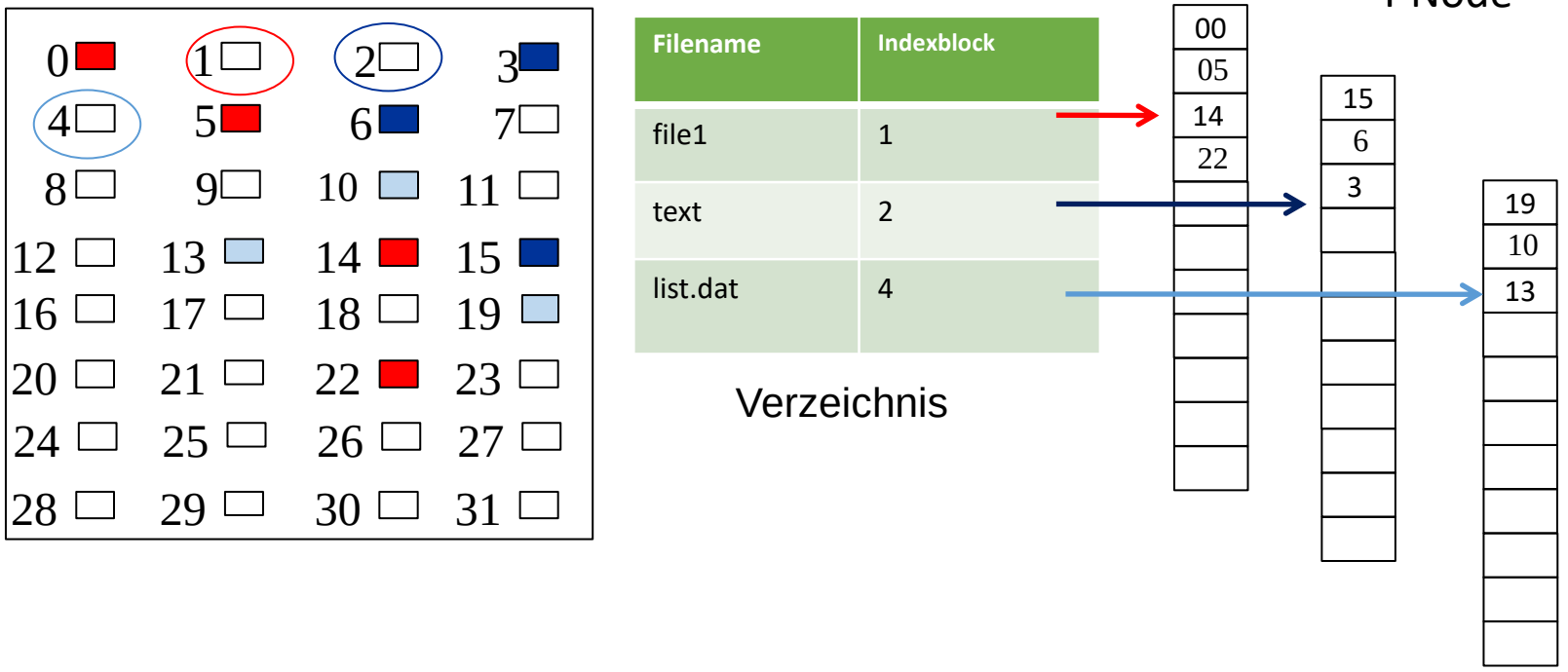
Defragmentierung



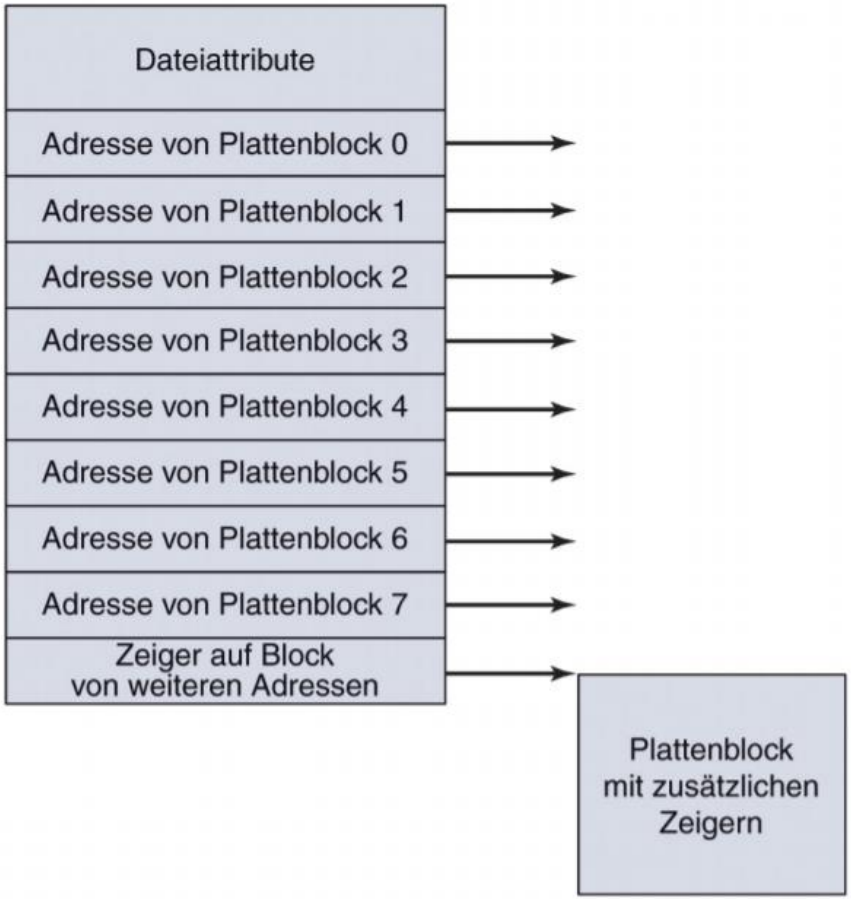


Indizierte Speicherung

- Eigener Block für jede Datei und jedes Verzeichnis, der die LBAs auf die Nutzdaten enthält



Nutzdaten: indizierte Speicherung

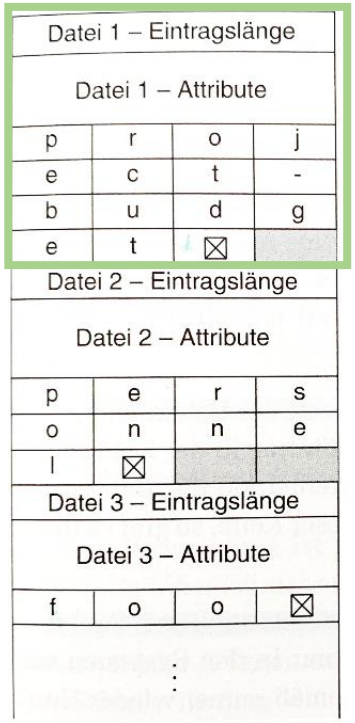


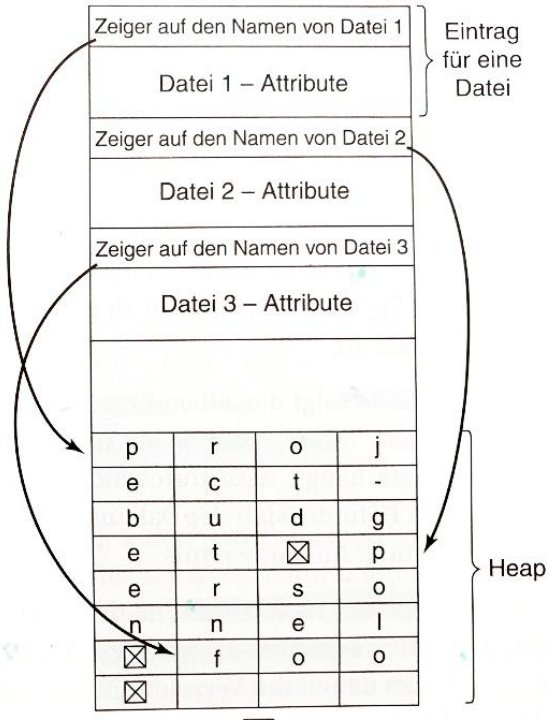
Implementierung von Verzeichnissen

- Für das Öffnen einer Datei benutzt das OS den angegebenen Pfadnamen um den Verzeichniseintrag auf der Platte zu finden
- Verzeichniseintrag stellt Information dar, die für das Auffinden der Plattenblöcke benötigt wird
- Je nach System kann das die Plattenadresse der gesamten Datei sein, die Nummer des ersten Blocks, oder die Nummer des I-Nodes

Implementierung von Verzeichnissen

- 1. Möglichkeit: Bestimmung einer maximalen Länge
  - Problem?
- Alternative: jeder Verzeichniseintrag enthält eine feste Menge
- 2 Möglichkeiten, mit langen Dateinamen in Verzeichnissen umzugehen:
  - linear oder
  - mit einem Heap (Arbeitsspeicher)





Dateisystem-Erweiterung: Journaling

- Journaling oder wiederherstellbare Dateisysteme
- Jede Modifikation im Dateisystem wird **bevor** sie physikalisch auf der Festplatte ankommt in einer speziellen Datei – einer Logdatei - protokolliert
  - Jede Modifikation wird in Suboperationen zerlegt, dann wird eine Modifikation als **Transaktion** bezeichnet
    - Beispiele für Modifikationen: Anlegen einer neuen Datei, Löschen einer Datei, Schreiben in eine Datei, Löschen von Sätzen einer Datei, etc.
  - Beispiel: Suboperationen beim **Löschen einer Datei**
    - **Markiere Transaktionsbeginn**
    - Freigeben der belegten Datenblöcke durch Setzten der Bits in der Bitmap des Dateisystems (falls Bitmaps verwendet werden)
    - Freigeben des Eintrags der Datei
    - Löschen des Dateieintrags im Verzeichnis
    - **Ende der Transaktion (= Commit)**

Dateisystem-Erweiterung: Journaling

- Nach einem Systemabsturz startet ein Recovery Utility (z.B.: chkdsk), das auf Grund der Logdatei erkennt, welche Suboperationen einer Transaktionen bis zum Systemabsturz vorgenommen wurden und welche nicht.
- Es *entfernt* alle Suboperationen (**Undo**), wenn **kein Transaktionsende** gefunden wurde.
- Es *komplettiert* alle Suboperationen auf der Festplatte (**Redo**), wenn **das Transaktionsende** gefunden wurde.
- Metadaten-Journaling
  - lediglich die Konsistenz der Metadaten wird garantiert
- Full-Journaling
  - auch die Konsistenz der Nutzdaten gewährleistet

Implementierungen & Beispiele I

- Konkrete Implementierung von Filesystemen
- Windows:
  - Microsoft FAT
    - FAT12, FAT16, FAT32, exFAT
  - Microsoft NTFS
    - Alle Informationen zu Dateien in einer Master File Table (MFT) gespeichert
    - Freispeicherverwaltung: Bitmaps
    - Indizierte Speicherung – wie ein I-Node nur das die Knoten MFT-Dateisätze

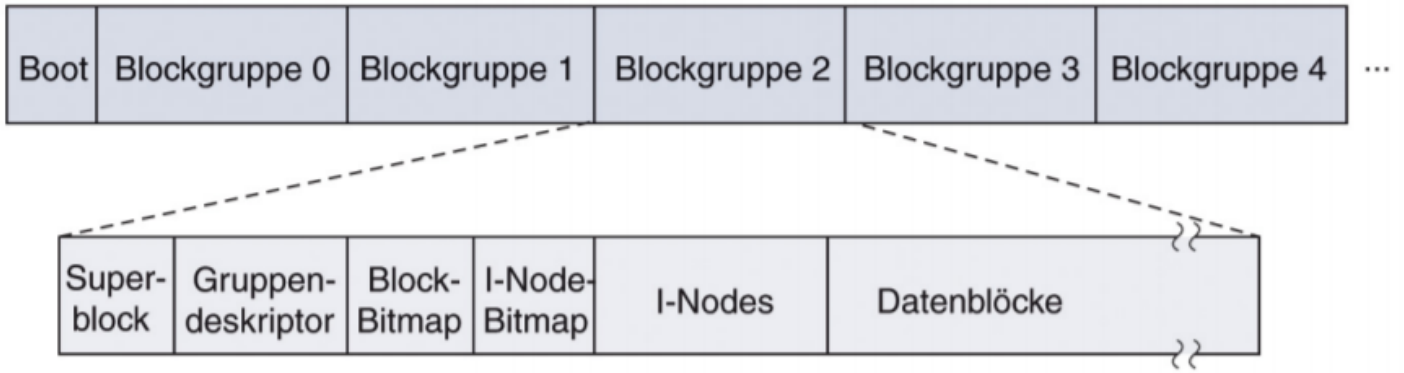
Implementierungen & Beispiele II

- **Unix/Linux/*BSD/Solaris:**
  - ext2, ext3, ext4, UFS, ZFS
  - Unix/Linux – Inode-Liste, arbeitet mit einer abgewandelten Form von Indizierung (Inode-basiertes Dateisystem)
- **macOS:**
  - APFS, HFS+
  - Inode-basiertes Dateisystem, ist optimiert für SSD, Flashspeicher
  - Wird auch bei iPhones, iPad
  - Implementierung von Verzeichniseinträgen: Baumstruktur um schnelle Suchen zu ermöglichen

Beispiel Linux: Dateisysteme

- Ext: extended-Dateisystem
  - Erstes Dateisystem, Schicht für mehrere Dateisystem wurde eingeführt VFS
  - Ext3 wurde Ext2 um Journaling Funktionalität ergänzt, Datenstrukturen zur Partitionsverwaltung blieben unverändert
- Ext4 stellt 48 Bit (früher 32 Bit) für die LBA Adressierung zur Verfügung und arbeitet mit Extents
  - Verwaltungsdaten werden über gesamte Partition verteilt
  - Man versucht Verwaltungsdaten und Nutzdaten räumlich zusammenzufassen

Beispiel Linux Plattenlayout



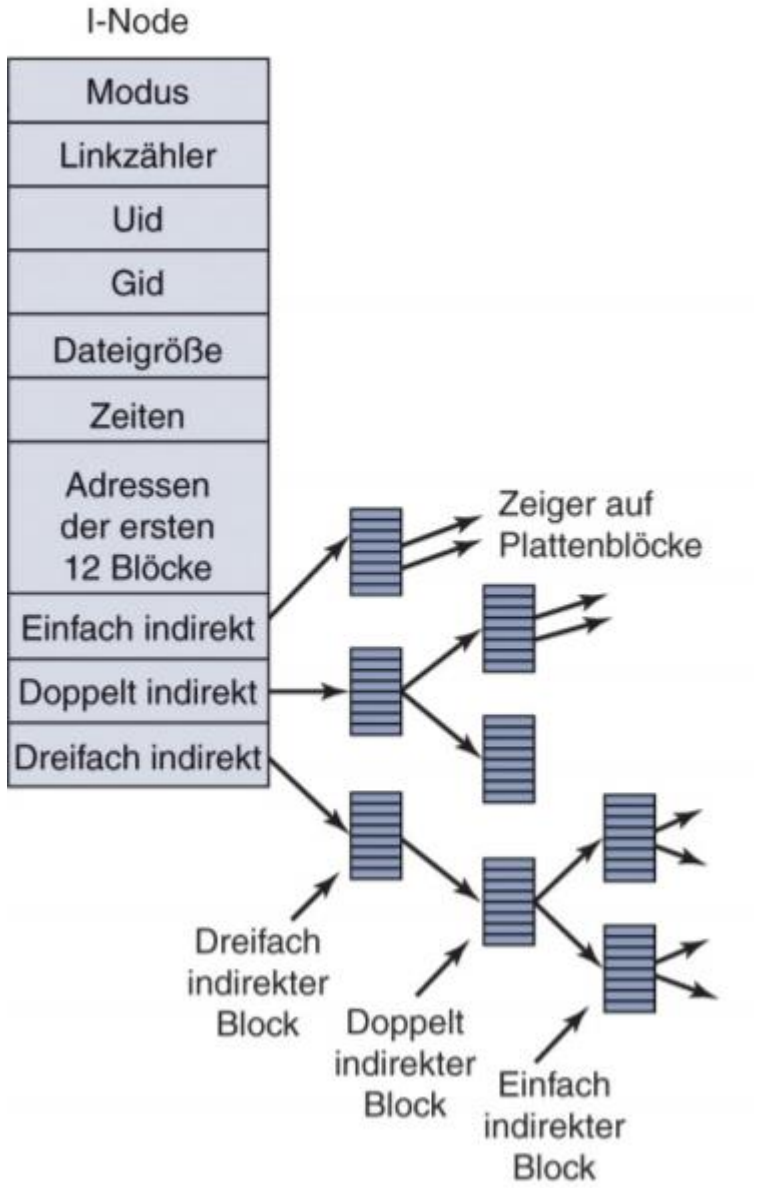
- Partition ist in Gruppen von Blöcken aufgeteilt
- Der erste Block in einer Gruppe ist der Superblock mit Infos über das Layout des Dateisystems (wie #Inodes, #Plattenblöcke, Zeiger auf die Liste der freien Blöcke)
- Gruppendeskriptor enthält die Infos über die Position der Bitmaps für I-Nodes und freie Blöcke in der Gruppe und die Anzahl der Verzeichnisse
- Dateien und Verzeichnisse werden nach Möglichkeit innerhalb einer Gruppe gespeichert (Wieso?)

Linux: Inodes

- Inode List:
  - Die Inode List enthält die Inodes
  - entspricht dem Inhaltsverzeichnis des Dateisystems
- Inode:
  - Für jedes Objekt (Datei, Verzeichnis,..) wird ein eigener Inode angelegt,
    - Datenstruktur, in der die Attribute des Objekts und die LBAs der Nutzdaten gespeichert werden.
  - Jeder Inode ist 128 Bytes groß

Datenstruktur Inode

- Jeder Inode beginnt mit Metainformation der Datei/Verzeichnis
- 12 direkte Pointer auf Datenblöcke
- 3 weitere indirekte Pointer:
  - Indirekter Block
  - Doppelt indirekte Blöcke
  - Triple indirekte Blöcke



Linux: Beispiel 1 für maximale Filegröße

- Annahmen:
  - Größe eines logischen Blocks: 1 KB (2 Sektoren)
  - Adresslänge LBA: 32 Bit (4 Byte)
    - – > 256 LBA‘s in einem Block
- dann:
  - 12 direkt adressierte Datenblöcke á 1 KB: 12 KB Dateigröße
  - 1 indirekter Block mit 256 LBA‘s zu Datenblöcken: 256 KB + 12 KB = max 268 KB Dateigröße
  - 1 doppelt indirekter Block: **?** Dateigröße
  - 1 dreifach indirekter Block: **?** Dateigröße

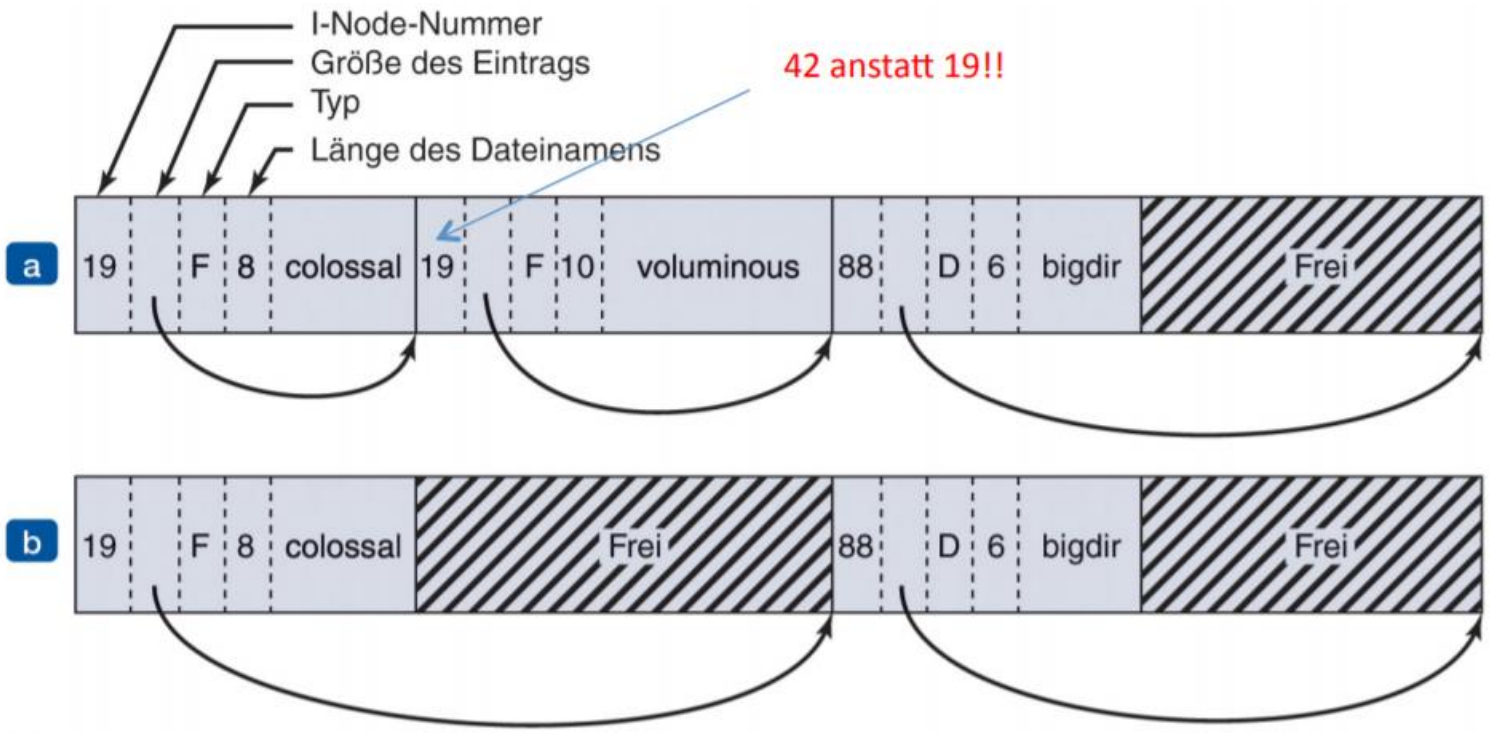
Linux: Beispiel 2 für maximale Filegröße

- Annahmen:
  - Größe eines logischen Blocks: 4 KB (8 Sektoren)
  - Adresslänge LBA: 32 Bit (4 Byte)
    - – > 1024 LBA‘s in einem Block
- dann:
  - 12 direkt adressierte Datenblöcke á 4 KB: 48 KB Dateigröße
  - 1 indirekter Block mit 1024 LBA‘s zu direkten Datenblöcken: 4096 KB + 48 KB = max 4144 KB Dateigröße
  - 1 doppelt indirekter Block: **?** Dateigröße
  - 1 dreifach indirekter Block: **?** Dateigröße

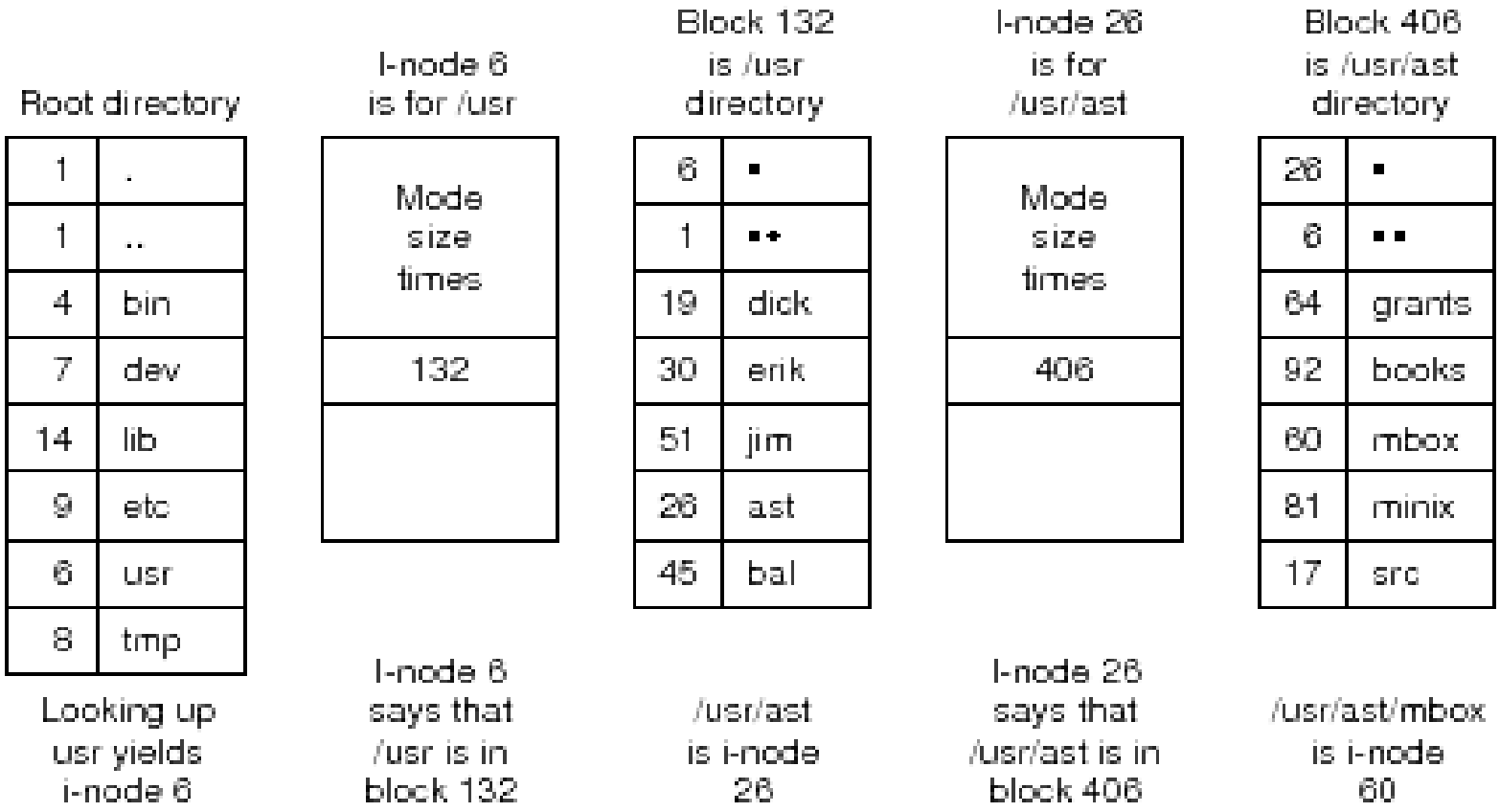
Linux: Implementierung von Verzeichnissen

- Implementierung als verkettete Liste oder mit zusätzlichem Index in Form eines H-trees
- Verkettete Liste (hier beschrieben):
  - Jedes Verzeichnis besteht aus einer Liste sogenannter **directory entries.** Das ist eine Datenstruktur, die für jedes Objekt des Verzeichnisses angelegt wird.
  - Die Liste der directory entries wird bei einer normalen Datei über die LBA's im dazu gehörigen Inode des Verzeichnisses verwaltet.
  - Die Länge eines **directory entries** hängt von der Länge des Filenamens ab.
  - Insgesamt wird in jedem **directory entry** folgendes gespeichert:
    - Inode nummer des Objekts
    - Länge des directory entries
    - Länge des Namens des Objekts
    - Typ des Objekts
    - Name des Objekts

Linux: Aufbau einer Verzeichnisdatei



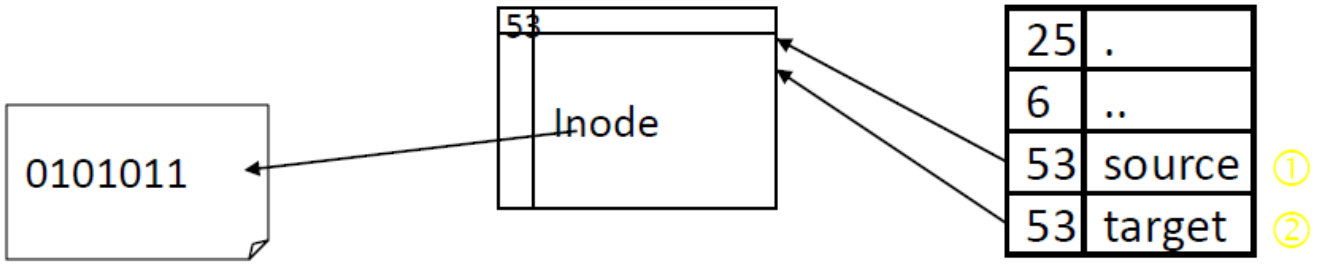
Linux: Zugriff über Pfadangabe



Beispiel: Zugriff auf `/usr/ast/mbox`

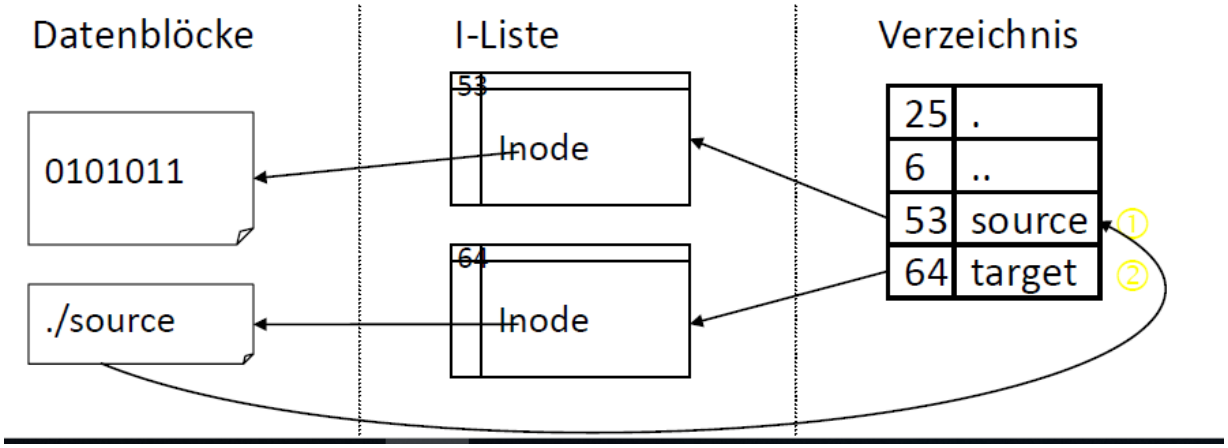
Arten von Links

- Hard Links:
  - Verschiedene Dateinamen teilen sich ein und denselben Inode
  - Link-Zähler im Inode wird erhöht, das bedeutet es wird die Anzahl der Namenseinträge mitgezählt
  - Beim Löschen einer Datei wird Anzahl der Einträge im Inode um eins vermindert
  - Wenn Einträge null sind, wird Plattenplatz freigegeben



Links II

- Soft Links:
- Verschiedene Dateinamen haben verschiedene Inodes
- Soft link ist ein Verzeichniseintrag der nicht auf einen Inode sondern über einen Pfadnamen auf eine Datei verweist
- Für den Soft link wird ein eigener Inode erzeugt



Ziele Dateisystem: Partitionsverwaltung

- Die Partition ist aus Sicht des DS so zu verwalten, dass
  - die Positionierungszeit der Lese-/Schreibköpfe minimal ist (besonders bei mechanischen Festplatten wichtig)
  - d.h. dass alle Blöcke einer Datei möglichst nahe beieinander liegen sollten (Fragmentierungsproblem)
  - die vorhandenen Diskkapazität bestmöglich ausgenutzt wird (Blockungsfaktor)

Zusammenfassung DS

- HDD & SSD
- Sektoren, Blöcke
- Logische vs. physische Sicht von Dateien und Verzeichnissen
- Layout eines Dateisystems
- Beispiele von Dateisystemen
- Ausgewähltes Beispiel: Linux

Betriebssysteme

Interrupts und System Calls

Susanne Schaller, MMSc
Andreas Scheibenpflug, MSc

SE Bachelor (BB/VZ), 2. Semester

Die HW sagt ‚Hallo‘

- Konzeptionell ist ein Interrupt (Unterbrechung) das Aufzeigen eines Ereignisses in einer HW Komponente
- Dieses Ereignis wird von der Hardware an die CPU geschickt
  - Wird als Unterbrechungsanforderung bzw. Interrupt Request (IRQ) bezeichnet
- Die CPU/das Betriebssystem sollte dann mit einer Aktion darauf reagieren
  - Diese Aktion wird als Unterbrechungsroutine bzw. Interrupt Handler bezeichnet
  - Der Interrupt Handler wird vom OS/Treibern zur Verfügung gestellt
- Vorteil: Keine verschwendete CPU Zeit durch Polling der HW
  - Polling: Periodische Abfrage der HW nach Änderungen

Beispiel – Ablauf Interrupt

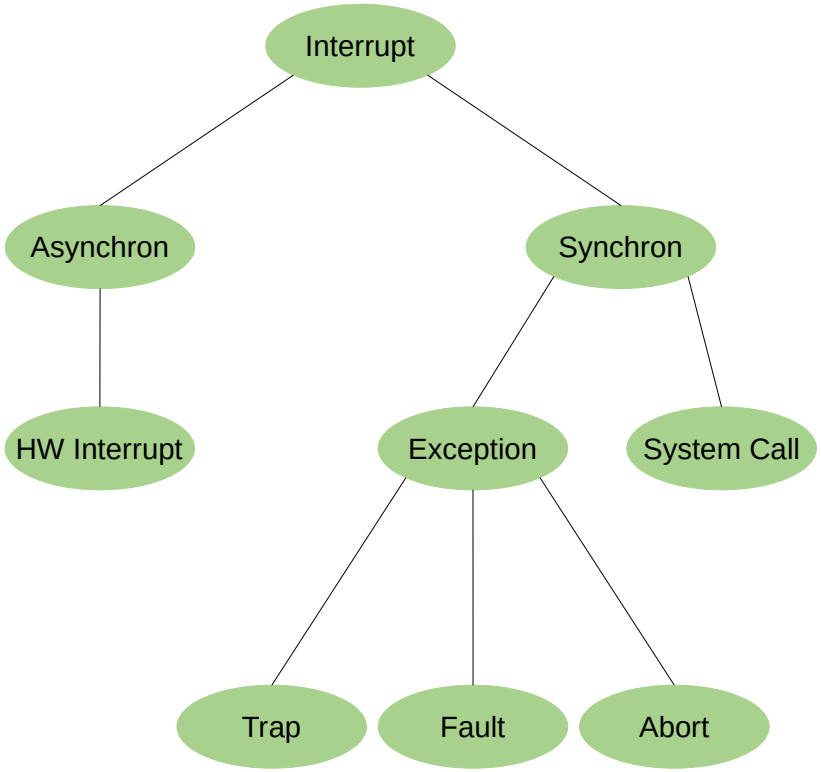
- BenutzerIn bewegt die Maus
- Die Maus erkennt die Bewegung und schickt die Daten an die CPU, z.B.
  - in welche Richtung wurde die Maus wie viel bewegt
  - welche Tasten wurden gedrückt
- Die CPU erkennt den Interrupt und unterbricht die Ausführung des aktuellen Prozesses
- Auf den Interrupt wird mit einem Interrupt Handler des Maus Treibers reagiert
- Interrupt Handler verarbeitet die Daten und führt weitere Aktionen durch (z.B. werden Prozesse (GUI) über die aktualisierten Daten informiert)
- Nach der Ausführung des Interrupt Handlers wird der unterbrochene Prozess fortgesetzt

Quellen von Interrupts

- Neben der Quelle HW gibt es noch andere Möglichkeiten wie ein Interrupt ausgelöst werden kann
- Prozessor: Wird durch die CPU bei der Ausführung eines Programms ausgelöst
  - z.B. Division durch Null, Adressfehler (Segfaults), ...
- Software: Ein Programm löst einen Interrupt aus, um einen Service des OS zu verwenden
  - Damit wird in den Kernel Mode gewechselt
  - Das OS führt die Aufgabe aus und returniert zum aufrufenden Programm
  - Wird als System Call bezeichnet (z.B. Einlesen einer Datei, Laden von Daten aus dem Internet, Verwenden einer Semaphore oder eines Mutex,...)

Kategorisierung (~ intel)

- Asynchron
  - Tritt parallel zur Ausführung der aktuellen Instruktion auf
- Trap
  - Instruktion löst Trap aus
  - Ausführung wird unterbrochen
  - Interrupt wird vom OS behandelt
  - Ausführung wird bei der nachfolgenden Instruktion fortgesetzt
  - Beispiel: Breakpoint, System Call,...
- Fault
  - Eine Exception die behandelt werden kann
  - Instruktion löst Fault aus
  - Ausführung wird unterbrochen
  - Interrupt wird vom OS behandelt
  - Instruktion die den Fault ausgelöst hat, wird wiederholt
  - Beispiel: Div. durch Null, Page Fault (siehe Foliensatz zu Speicherverwaltung)
- Abort
  - Es liegt ein gravierender Fehler vor
  - Die Ausführung der Anwendung wird abgebrochen
  - Beispiel: Hardware Fehler, Illegale/Inkonsistente Werte in System Tabellen



Ablauf Interrupt - HW

- Geräte sind über einen Bus mit dem (Programmable) Interrupt Controller (PIC) verbunden
- Der Interrupt Controller agiert als Multiplexer, priorisiert die eingehenden Interrupt Requests (IRQs) und setzt ein Bit (IEB) um der CPU zu signalisieren, dass ein Interrupt anliegt
- CPU überprüft nach jeder ausgeführten Instruktion dieses Bit. Ist es gesetzt dann
  - Sichert die CPU den Zustand des aktuellen Prozesses (Instruction Pointer, Register, Stack,...mehr dazu im Foliensatz Prozesse)
  - Signalisiert die CPU dem PIC dass sie bereit ist und erhält die Nummer des Interrupts vom PIC
  - Die Nummer ist einem Interrupt Handler zugeordnet
  - Der Interrupt Handler wird ausgeführt
  - Nach Beendigung des Interrupt Handlers wird der zuvor gestoppte Prozess wieder geladen und weiter ausgeführt

Interrupt Handler

```
/*
 * Linux IRQ vector layout.
 *
 * There are 256 IDT entries (per CPU - each entry is 8 bytes) which can
 * be defined by Linux. They are used as a jump table by the CPU when a
 * given vector is triggered - by a CPU-external, CPU-internal or
 * software-triggered event.
 *
 * Linux sets the kernel code address each entry jumps to early during
 * bootup, and never changes them. This is the general layout of the
 * IDT entries:
 *
 *  Vectors   0 ...  31 : system traps and exceptions - hardcoded events
 *  Vectors  32 ... 127 : device interrupts
 *  Vector  128         : legacy int80 syscall interface
 *  Vectors 129 ... LOCAL_TIMER_VECTOR-1
 *  Vectors LOCAL_TIMER_VECTOR ... 255 : special interrupts
 *
```

- Werden von Treibern/OS zur Verfügung gestellt
- Werden vom OS für Interrupts registriert
- Beispiel Linux: Interrupt Descriptor Table (IDT)
  - Tabelle im Hauptspeicher die von Linux verwaltet wird
  - Enhält das Mapping Interrupt (Nummer) → Interrupt Handler
  - Wird zur Boot Zeit befüllt
  - Dem Prozessor wird über ein Register mitgeteilt, wo sich diese Tabelle im Hauptspeicher befindet
  - Die CPU verwendet diese Tabelle um Interrupt Handler zu laden
  - Mehrstufige Implementierung
    - Interrupt Handler sollen möglichst schnell sein, da Interrupts blockiert werden
    - Top Half: Interrupt Handling (Interrupts blockiert)
      - Es werden z.B. die Daten von der HW (Netzwerkpakete, Datenblöcke,…) gelesen und in einen Puffer gespeichert
    - Bottom Half: Weiterführende Aufgaben (Interrupts wieder aktiviert)
      - Können asynchron ausgeführt werden
      - Verarbeitung der gelesenen Daten, Weiterleitung der Daten an darüber-liegende Schichten (Protokollschicht, Dateisystem,…)

Beispiel Timer Interrupt

- Programmable Interval Timer (PIT)
  - PIT ist im Mainboard des Computers untergebracht
  - Er erzeugt Interrupts in periodischen Intervallen (n mal pro Sekunde)
  - Liefert Ticks: Zahl der gesendeten Timer-Interrupts pro Sekunde, meistens 50, 60 oder 100 Ticks → ca. alle 10 ms ein Interrupt
  - Intervall-Länge ist programmierbar
  - Timer-Interrupt läuft mit hoher Priorität
- Interrupt Handler für den Timer Interrupt
  - Aktualisiert die Uptime (Zeit, die seit dem Systemstart vergangen ist) alle 10 ms (= 1 Tick)
  - Aktualisiert die Systemzeit
  - Überprüft die Zeitscheibe des aktuellen Prozesses – falls abgelaufen, wird der Scheduler gestartet, der die CPU einem anderen Prozess zuteilt (mehr dazu im Foliensatz Prozesse)
  - Aktualisiert Ressourcen-Nutzungsstatistik

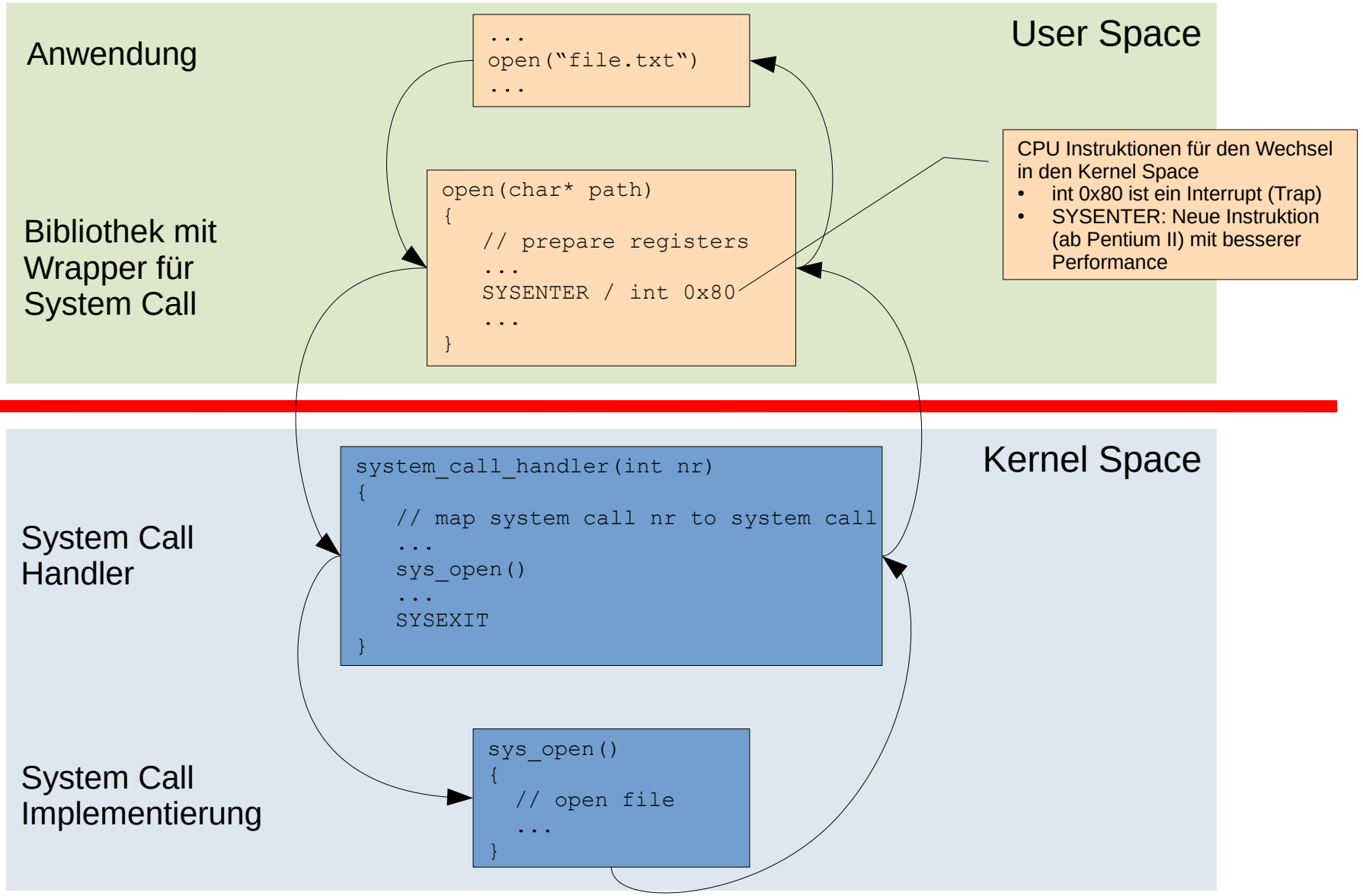
System Calls

- Ein OS stellt dem/der AnwendungsentwicklerIn eine Reihe von Funktionalitäten zur Verfügung
  - System Calls erlauben diese Funktionalitäten einfach in Anwendungen zu verwenden
  - Abstrahieren die Komplexität der darunter liegenden HW
  - Ermöglichen Aktionen, die für Anwendungsprogramme nicht erlaubt sind (z.B. direkter Zugriff auf die HW)
- Beispiele
  - Abstrahierte Formen der Ein- und Ausgabe auf Geräte, z.B.
    - Lesen/Schreiben auf der Festplatte
    - Senden/Empfangen von Paketen über die Netzwerkschnittstelle
  - Interaktion mit Prozessen, dem Arbeitsspeicher,... z.B.
    - Starten von neuen Prozessen
    - Synchronisieren/Interaktion mit anderen Prozessen
    - Anfordern/Freigeben von Arbeitsspeicher

System Calls

- Im Gegensatz zu Prozessen von BenutzernInnen läuft das OS in einem privilegierten Modus
  - Prozesse im User Mode (Space) haben eingeschränkte Rechte um Fehler zu verhindern, die andere Programme oder das OS negativ beeinflussen könnten
  - Das OS läuft im Kernel Mode (Space) und hat den kompletten CPU Befehlssatz, Arbeitsspeicher und Zugriff auf die HW zur Verfügung
  - Diese unterschiedlichen Modi sind CPU-unterstützt, z.B. bei Intel
    - Ring 0 für Kernel Mode
    - Ring 3 für User Mode
- Dieser Übergang von User Space in Kernel Space wird mit einer speziellen CPU Instruktion angestoßen, welche einen Interrupt (Trap) auslöst
  - Der aufrufende Prozess wird dabei unterbrochen, bis der Kernel die Bearbeitung des System Calls beendet hat und das Ergebnis zurückliefert
  - Der Aufruf eines System Calls funktioniert semantisch wie ein normaler Funktionsaufruf

System Call - Ablauf



System Calls in Linux

- Linux bietet ~ 500 System Calls an
  - Manche System Calls werden nicht auf allen Plattformen unterstützt
- System Calls werden im Normalfall nicht direkt aufgerufen
  - Werden über C Bibliotheken zur Verfügung gestellt (z.B. glibc)
  - Diese Funktionen sind meist „dünne Wrapper“ für den tatsächlichen System Call und machen oft nicht mehr als
    - Funktionsargumente und die Nummer des System Calls in die richtigen Register kopieren (wie vom Kernel erwartet)
    - In den Kernel Mode wechseln (den Trap ausführen)
    - Abholen des Return Codes und Rückgabe an den Aufrufer
  - Meist sind die Namen von System Calls und Wrapper ident (z.B. `fork(..)`, `open(..)`, `chdir(..)`, `poll(..)`,...)

Bsp. Mouse IRQ + System Calls

- Maus Treiber erhält IRQs über den PIC von der Maus
- Der Treiber schreibt die Daten des IRQs in eine Datei in /dev
- Verwenden der System Calls `open(..)` und `read(..)` zum Auslesen und Anzeigen der Daten im User Space

```
x=2, y=1, left=0, middle=0, right=0
x=4, y=3, left=0, middle=0, right=0
x=3, y=2, left=0, middle=0, right=0
x=3, y=3, left=0, middle=0, right=0
x=2, y=1, left=0, middle=0, right=0
x=3, y=2, left=0, middle=0, right=0
x=2, y=2, left=0, middle=0, right=0
x=2, y=1, left=0, middle=0, right=0
x=4, y=1, left=0, middle=0, right=0
x=1, y=0, left=0, middle=0, right=0
```

```c
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>

// gcc mouse.c -o mouse
// sudo ./mouse
int main(int argc, char** argv)
{
    int fd, bytes;
    unsigned char data[3];
    const char *device_path = "/dev/input/mouse1";
    int left, middle, right;
    signed char x, y;

    // open device file for mouse
    fd = open(device_path, O_RDONLY);
    if(fd == -1)
    {
        printf("ERROR Opening %s\n", device_path);
        return -1;
    }

    while(1)
    {
        // read blocks until new data is available and can be read
        bytes = read(fd, data, sizeof(data));

        if(bytes > 0)
        {
            left = data[0] & 0x1; // 00000001
            right = data[0] & 0x2; // 00000010
            middle = data[0] & 0x4; // 00000100

            x = data[1];
            y = data[2];
            printf("x=%d, y=%d, left=%d, middle=%d, right=%d\n",
                   x, y, left, middle, right);

        }
    }
    return 0;
}
```

https://stackoverflow.com/a/23317086

Zusammenfassung

- Interrupts
- Ablauf eines Interrupts
- Interrupts aus Sicht des OS
- Funktionsweise System Calls

Betriebssysteme

Prozesse, Threads und Scheduling

Susanne Schaller, MMSc
Andreas Scheibenpflug, MSc

SE Bachelor (BB/VZ), 2. Semester

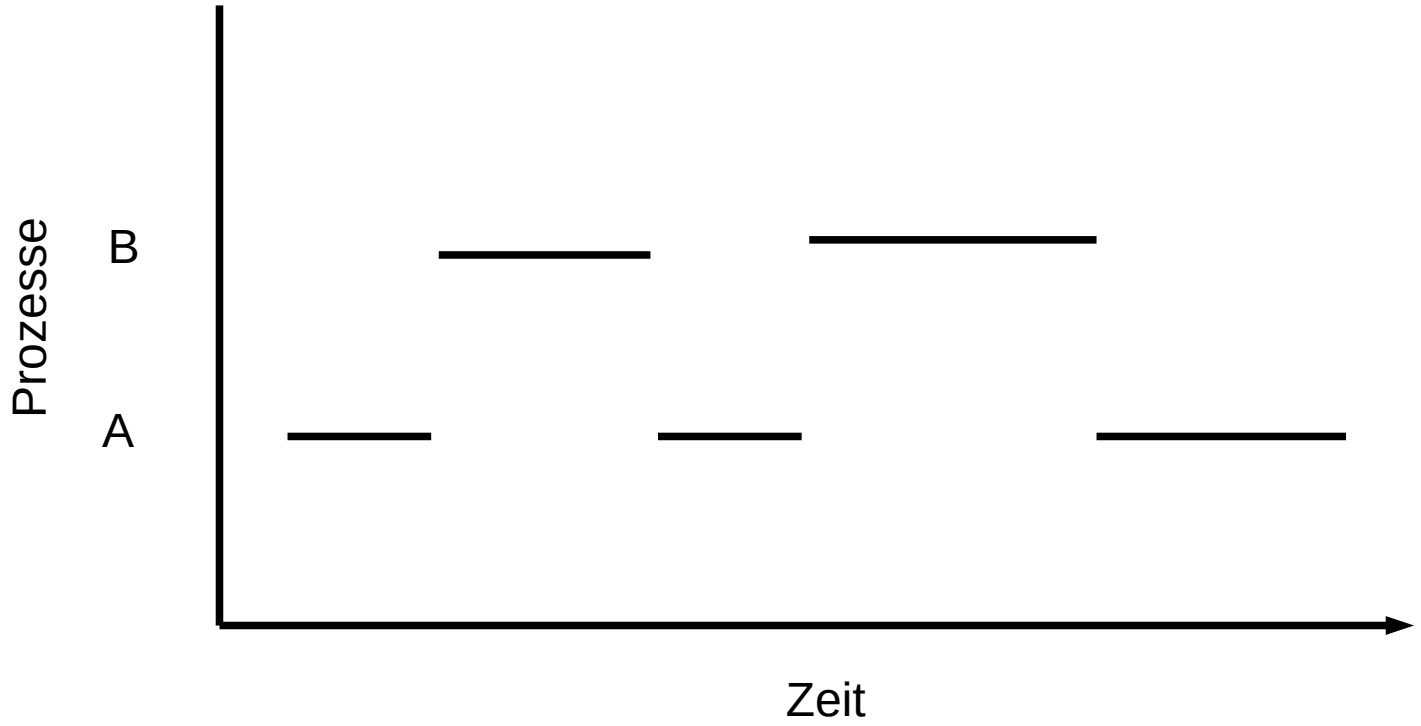
Prozess

- Abstraktion eines laufenden Programms
- Erlauben Pseudoparallelität wenn z.B. nur ein Prozessor vorhanden ist
- Gewöhnlich laufen auf einem Computer mehr Prozesse als Prozessoren/Kerne verfügbar sind
- OS kümmert sich darum, dass jeder Prozess CPU Zeit bekommt (Scheduling)
- Für den/die BenutzerIn sieht es so aus, als ob alle Prozesse gleichzeitig laufen

Prozessmodell

- Ein Prozess ist eine Instanz eines Programms zur Laufzeit und besteht aus
  - Code/Befehlszähler (Instruction Pointer)
    - Instruktion die als nächstes ausgeführt wird
  - Register
  - Variablen (Daten)
- Dem OS steht ein Prozessor zur Verfügung
- OS führt eine Menge an Prozessen pseudo-parallel aus
  - Jeder Prozess hat die CPU für eine begrenzte Zeit zur Verfügung
  - OS teil Prozessen die CPU zu und wechselt die Prozesse aus
  - Zu einem bestimmten Zeitpunkt wird immer nur ein Prozess ausgeführt

Prozessmodell



Prozesserzeugung

- Vier Ereignisse die zum Erzeugen eines neuen Prozesses führen
  - Initialisierung des OS
    - Beim Starten des OS werden Prozesse erzeugt
  - System Calls
    - Prozess, der einen anderen Prozess erzeugt (z.B. Linux: `fork()`, Windows: `CreateProcess()`)
  - Durch den/die BenutzerIn
    - z.B. Starten eines Programms in der Shell
  - Start eines Stapeljobs
    - z.B. Batchprocessing bei Großrechnern (Mainframes)
- Technisch gesehen meist Variante 2 (System Calls)

Prozesserzeugung - Beispiel

- `fork` dupliziert den Prozess inkl. Speicher
- Rückgabewert von `fork` ist
  - 0 im Kindprozess
  - PID des Kindes im Elternprozess
- `execve` erlaubt das Ausführen eines Programms
  - Ersetzt den aktuellen Prozess inkl. Speicher durch das auszuführende Programm

```c
#include <stdio.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>

// gcc fork.c -o fork
// ./fork
int main(void)
{
        pid_t pid_new;
        int status;
        char *args[] = {"uname", "-a", NULL};
        char *env[] =  {NULL};

        pid_new = fork();

        if (pid_new == 0)
        {
                printf("Child process says Hello\n");
                execve("/usr/bin/uname", args, env);
                printf("Never executed code if execve is successful\n");
        }
        else
        {
                printf("Original process says Hello!\n");
                wait(&status);
                printf("Original process: Child finished, exiting...\n");
        }
        return 0;
}
```

```
Original process says Hello!
Child process says Hello
Linux ThinkPad-L390 5.4.0-31-generic #35-Ubuntu SMP Thu May 7 20:20:34 UTC 2020
x86_64 x86_64 x86_64 GNU/Linux
Original process: Child finished, exiting...
```

Prozessbeendigung

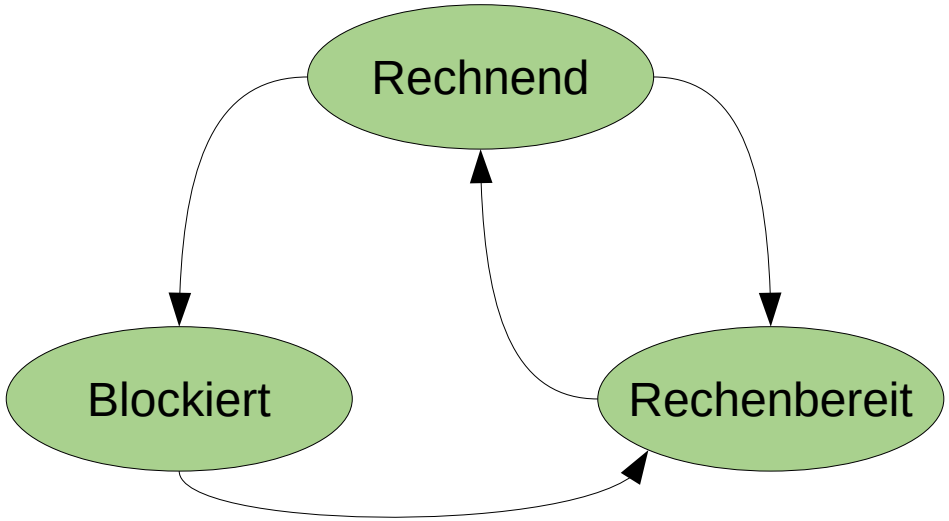
- Vier Ereignisse, die zum Beenden eines Prozesses führen
  - Normales Beenden (freiwillig)
    - Aufgabe beendet (z.B. Linux: `exit(0)`, Windows: `ExitProcess()`)
  - Beenden aufgrund eines Fehlers (freiwillig)
    - Programm stellt Fehler fest und beendet mit z.B. `exit(-1)`
  - Beenden aufgrund eines schwerwiegenden Fehlers (unfreiwillig)
    - z.B. Programmierfehler, Division durch Null, Zugriff auf ungültige Speicheradressen,...
  - Beenden durch einen anderen Prozess (unfreiwillig)
    - z.B. Linux mit `kill()` oder Windows mit `TerminateProcess()`

Prozesszustände

- Prozesse müssen oft warten auf
  - Ein- und Ausgabe auf Geräten
    - z.B. Lesen vom Dateisystem, `HTTP GET`, …
  - Andere Prozesse
    - z.B. `cat file.txt | grep Linux | sort`
- Dieser Zustand wird als „blockiert“ bezeichnet
- Im Gegensatz zum Stoppen eines rechenbereiten Prozesses weil das OS dies entscheidet
  - z.B. Umschalten zu einem anderen Prozess weil aktueller Prozess seinen Anteil an Rechenzeit verbraucht hat

Prozesszustände

- Drei Zustände
  - Rechnend
    - Prozess läuft auf der CPU und führt Befehle aus
  - Rechenbereit
    - Prozess ist bereit aber gestoppt, weil gerade ein anderer Prozess rechnet
  - Blockiert
    - Nicht lauffähig, da auf ein Ereignis (z.B. Benutzereingabe, Datei eingelesen,...) gewartet wird



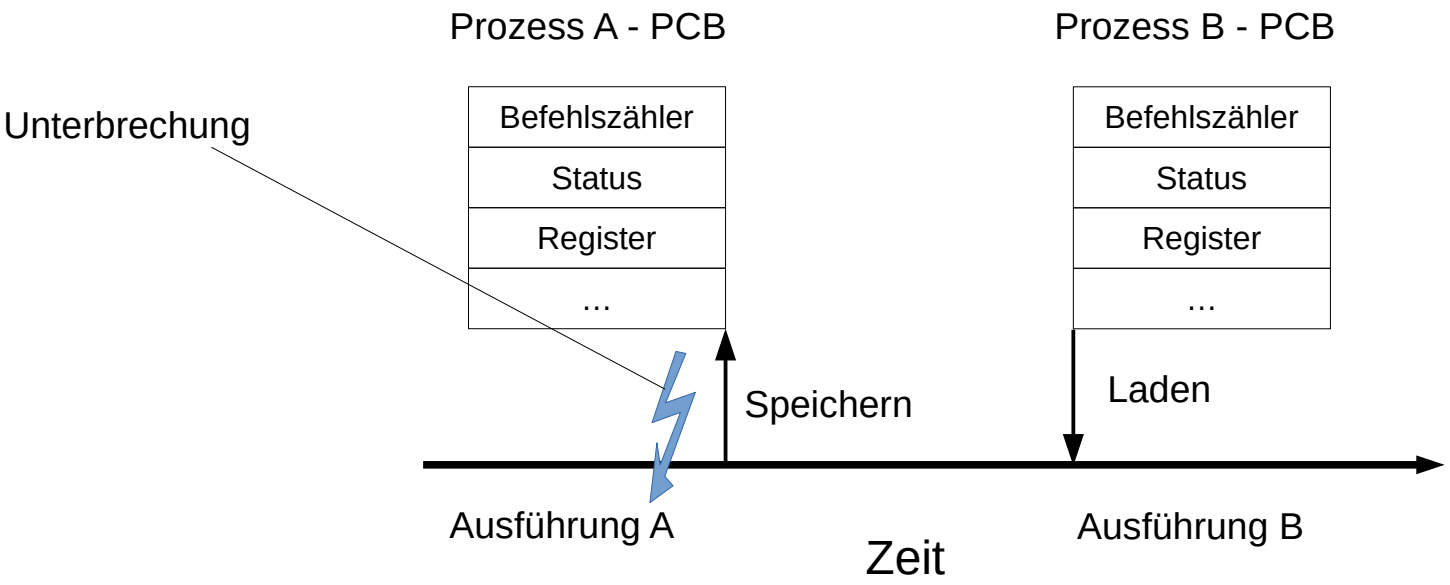
Prozesstabelle

- OS besitzt eine Prozesstabelle, welche eine Liste aller Prozesse enthält
- Ein Eintrag in dieser Liste wird als Prozesskontrollblock (PCB) bezeichnet
  - Enthält alle Informationen (Zustand) zu einem Prozess
  - Ermöglicht damit das Fortsetzen von Prozessen
  - Ermöglicht Scheduling Entscheidungen zu treffen

Prozessverwaltung	**Speicherverwaltung**	**Dateiverwaltung**
Register	Zeiger auf Textsegment	Arbeitsverzeichnis
Befehlszähler	Zeiger auf Datensegment	Dateideskriptor
Stackpointer	Zeiger auf Stacksegment	User Id
Programmstatuswort	...	Gruppen Id
Prozess ID (PID)		...
Benutzte CPU Zeit		
Prozesszustand		

Prozesswechsel

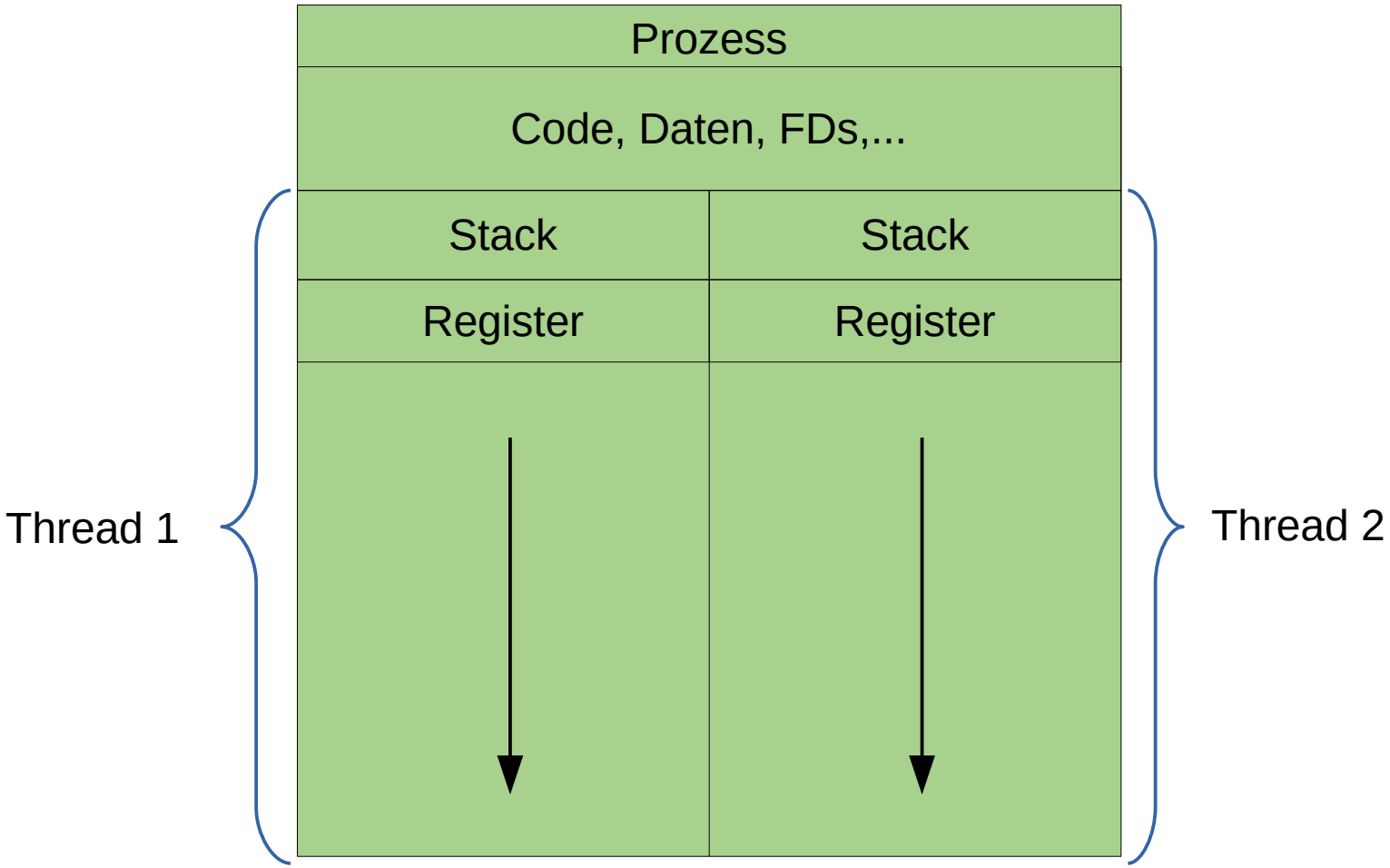
- Anzahl Prozesse > Anzahl CPUs (in unserem Modell == 1)
  - OS muss zwischen Prozessen wechseln, da jeder Prozess CPU Zeit zur Ausführung benötigt
- Prozesswechsel - Context Switch
  - Laufender Prozess wird unterbrochen
  - Scheduler wählt einen anderen, rechenbereiten Prozess aus und führt diesen aus
- Informationen zu unterbrochenem Prozess müssen gespeichert werden
  - Damit dieser später wieder ausgeführt werden kann
  - Zustand wird im PCB gespeichert



Threads

- Jeder Prozess hat seinen eigenen Adressraum („Speicher“)
  - CPU führt eine Instruktion nach der anderen sequentiell aus
- Ein Prozess kann mehrere Threads starten
  - Threads sind parallel laufende Ausführungspfade
  - Im selben Adressraum wie der Prozess
- Vorteile
  - Einfacheres Programmiermodell (im Vergleich zu z.B. IPC)
  - Leichtgewichtiger als Prozesse
  - Performance, z.B. 1 Thread rechnet während ein 2. auf I/O wartet
  - Performance bei Multicore Prozessoren

Threads


Prozess
Code, Daten, FDs,...
Stack
Stack
Register
Register
Thread 1
Thread 2

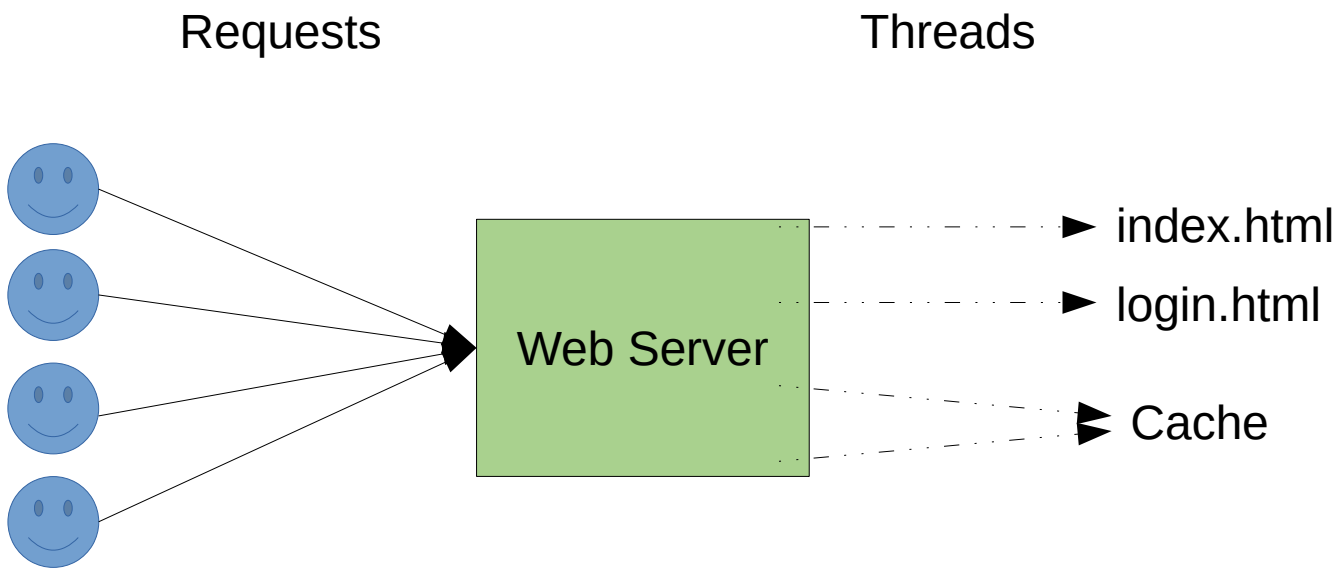
Threads

- Implementierung
  - Im OS: OS hält Thread Tabelle ähnlich zur Prozesstabelle mit z.B.:
    - Stack
    - Register
    - Befehlszähler
    - Threadzustand
  - Im User Space: Implementiert Funktionalität in Laufzeitsystem (Bibliothek)
    - Bessere Performance
    - Probleme bei Blocking I/O, Page Faults,… → andere Threads sind auch blockiert
  - Hybrid: Kombination aus OS und User Space Threads

Threads - Implementierungen

- Windows
  - „Echte“ OS Threads, Verwenden der `CreateThread(...)` Funktion
  - Jeder Prozess besteht aus mindestens einem Thread
  - Hybrid Threads mit User-Mode Scheduling
- Linux
  - Threads werden durch das OS verwaltet
  - Es wird aber die Implementierung der Prozessverwaltung verwendet
  - Das bedeutet beinahe kein Unterschied zwischen Prozessen und Threads aus Kernel Sicht
  - Verwendung der Funktion `clone(…)` und setzen der Flags für das Erlauben des Zugriffs auf die File Deskriptoren etc. im Prozessadressraum
  - PID bleibt allerdings gleich

Threads - Beispiel



Threads - Beispiel

- Diskussion: Implementierungsmöglichkeiten
  - 1 Prozess, sequenzielle Abarbeitung
  - Mehrere Prozesse, Parallelität
  - 1 Prozess, mehrere Threads (pro Request?)
  - 1 Prozess, Non-Blocking IO
  - ...

POSIX Threads - Beispiel

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

// gcc -pthread threads.c -o threads
// ./threads

void *say_hello_thread(void* id)
{
        printf("Hi! I'm thread %d\n", id);
        pthread_exit(NULL);
}

int main(void)
{
        static int nr_of_threads = 20;
        pthread_t threads[nr_of_threads];

        for(int i = 0; i < nr_of_threads; i++)
        {
                pthread_create(&threads[i], NULL, say_hello_thread, (void*) i);
        }

        for(int i = 0; i < nr_of_threads; i++)
        {
                pthread_join(threads[i], NULL);
        }

        return 0;
}
```

```
Hi! I'm thread 0
Hi! I'm thread 3
Hi! I'm thread 2
Hi! I'm thread 5
Hi! I'm thread 1
Hi! I'm thread 4
Hi! I'm thread 6
Hi! I'm thread 7
Hi! I'm thread 8
Hi! I'm thread 9
Hi! I'm thread 10
Hi! I'm thread 11
Hi! I'm thread 13
Hi! I'm thread 12
Hi! I'm thread 14
Hi! I'm thread 16
Hi! I'm thread 17
Hi! I'm thread 18
Hi! I'm thread 19
Hi! I'm thread 15
```

Threads/IPC

- Interprocess Communication
- Drei Themen
  - Informationsweitergabe
    - Austausch von Informationen zwischen Prozessen
  - Synchronisation
    - Warten auf andere Prozesse
  - Abhängigkeiten/Reihenfolgen
    - Mehrere Prozesse, die in einer definierten Reihenfolge eine Aufgabe abarbeiten sollen (→ Race Conditions)
- Nur der erste Punkt betrifft Threads nicht →selber Adressraum
- Daraus ergeben sich eine Reihe von Problemen und Lösungen: Race Conditions, Critical Regions, Mutex, Semaphoren → LVA VPS

Scheduling

- Bei PCs / mobilen Geräten laufen mehr Prozesse als CPUs zur Verfügung stehen
- Die Prozesse müssen sich daher die Ressource CPU teilen
- Scheduler ist Teil des OS
  - Entscheidet welcher Prozess als nächstes ausgeführt wird
  - Diese Berechnung der Entscheidung wird auch als Scheduling bezeichnet und durch den Scheduling Algorithmus, den der Scheduler implementiert, getroffen
  - Das Resultat der Berechnung führt zu einem Context Switch

Ziele

- Wir werden uns auf interaktive Systeme fokusieren
  - Im Gegensatz zu Batchverarbeitungssystemen oder Echtzeitsystemen
- Ziele
  - Fairness
    - Vergleichbare Prozesse sollten auch den gleichen Anteil an Rechenzeit bekommen
  - Policy Enforcement
    - Unterstützt das OS Policies (z.B. Prioritäten, Deadlines), sollten diese auch eingehalten werden
  - Balance
    - Optimale Ausnutzung der HW Ressourcen
    - Gleichmäßige Auslastung der CPU und der I/O Geräte ist zu bevorzugen
  - Response Time
    - Minimierung der Zeit zwischen einer Benutzereingabe und einer dementsprechenden Rückmeldung
    - Vorrang gegenüber z.B. Hintergrundtasks (Daemons/Services)

Things to consider

- Context Switch ist nicht gratis
  - Wechsel in den Kernel Space, Speichern des PCBs,…
  - Viele Context Switches kosten Performance
    - Steht im Gegensatz zu Ziel Response Time
- Verhalten von Prozessen
  - CPU-bound: Verbringen die meiste Zeit mit Berechnungen
  - I/O-bound: Verbringen die meiste Zeit mit Ein- und Ausgabe
  - Scheduler muss auf das Verhalten von Prozessen Rücksicht nehmen
    - z.B. Bevorzugung von I/O-bound Prozessen, damit diese die Festplatte „beschäftigen“, während CPU-bound Prozesse die CPU beschäftigen → Ziel Balance

Zeitpunkte

- Es gibt mehrere Ereignisse die einen Context Switch auslösen
  - Wenn ein neuer Prozess erzeugt wird
    - Ausführung des Elternprozesses oder des neuen Prozesses
  - Beendigung eines Prozesses
  - Wenn ein Prozess blockiert
    - Blockierendes I/O (System Call), Semaphore, Mutex,…
  - Auftreten eines I/O Interrupts
    - Wurde eine I/O Operation fertig, kann ein Prozess, der auf diese wartet, weiter ausgeführt werden (blockierend → rechenbereit)
    - Oder es könnte trotzdem der aktuelle Prozess weiter ausgeführt werden weil er z.B. eine höhere Priorität hat

Context Switch – Am Beispiel eines Interrupts

- Ein IRQ tritt auf
  - CPU prüft nach der Ausführung der aktuellen Instruktion ob ein IRQ aufgetreten ist
- CPU stoppt Ausführung des aktuellen Prozesses und legt Instruction Pointer etc. auf einen Stack
- CPU lädt mit Hilfe der Interrupt Descriptor Table (IDT) den Interrupt Handler
  - Sichert Register des zuvor ausgeführten Prozesses und die Daten die zuvor von der CPU auf einen Stack gelegt wurden
  - Richtet Stack ein
  - Führt das Interrupt Handling aus (z.B. Lesen von Daten in einen Puffer)
  - Bereinigt Stack/Register
- Aufruf des Schedulers
  - Selektion des nächsten Prozesses
- Ausführen des selektierten Prozesses
  - Laden des Instruction Pointers, Registers,...

Clock Interrupts

- HW Uhr stellt periodische (I/O) Interrupts zur Verfügung
  - Preemptive Scheduler verwenden diese (auch), um über Context Switches zu entscheiden
- Non-preemptive Scheduling
  - Context Switch nur, wenn ein Prozess blockiert oder die CPU „freiwillig hergibt“
- Preemptive Scheduling
  - Prozesse bekommen eine maximale Ausführungszeit (Zeitscheibe, „Quantum“) zugeordnet
  - Wird diese überschritten, wird der Prozess unterbrochen, auch wenn dieser noch rechenbereit ist, und ein anderer Prozess gestartet
  - Standard bei interaktiven Systemen

Schedulingalgorithmen

- Unterschiedliche OS implementieren unterschiedliche Algorithmen
  - Und Speichern auch unterschiedliche Daten im PCB aufgrund welcher eine Entscheidung zur Prozessselektion getroffen wird
- Hängt von dem Einsatzgebiet und der Zielsetzung ab
- Ein Batchsystem verfolgt andere Ziele als ein Echtzeitsystem oder ein interaktives System
  - Daher ist auch der Schedulingalgorithmus zwischen diesen Systemen anders
  - Ein Batchsystem kann z.B. auch mit non-preemptivem Scheduling auskommen
- Wir sehen uns im Folgenden einige Beispiele von Schedulingalgorithmen an

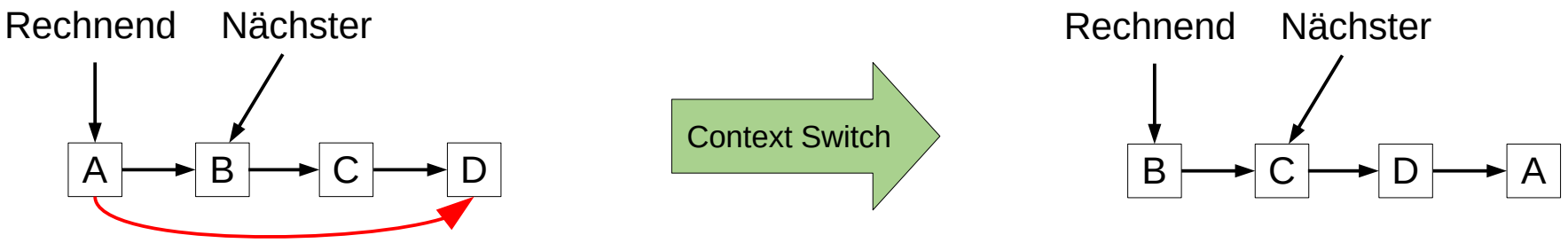
First Come, First Served (FIFO)

- Verwendung z.B. bei Batchsystemen
  - Non-preemptive
  - Neue Prozesse werden in eine Warteschlange (Queue) eingereiht
  - Prozesse werden der Reihe nach entnommen und abgearbeitet
  - Nachteil: Ziel der Balance wird nicht gewährleistet bei I/O-bound Prozessen

Round Robin

- Jeder Prozess bekommt das gleiche Quantum
  - Läuft er noch nach Ablauf des Quantums, wird er unterbrochen und ein anderer Prozess wird fortgesetzt (preemptive)
  - Blockiert der Prozess früher, wird ebenfalls gewechselt
- Daraus resultiert: Jeder Prozess ist gleich wichtig
  - Vorteil: Fair
  - Nachteil: Vielleicht nicht ganz richtig
- Weiterer Nachteil – Wahl des Quantums:
  - Kleines Quantum → Verschwendung von CPU Zeit
  - Großes Quantum → Beeinträchtigung der Interaktivität

Round Robin



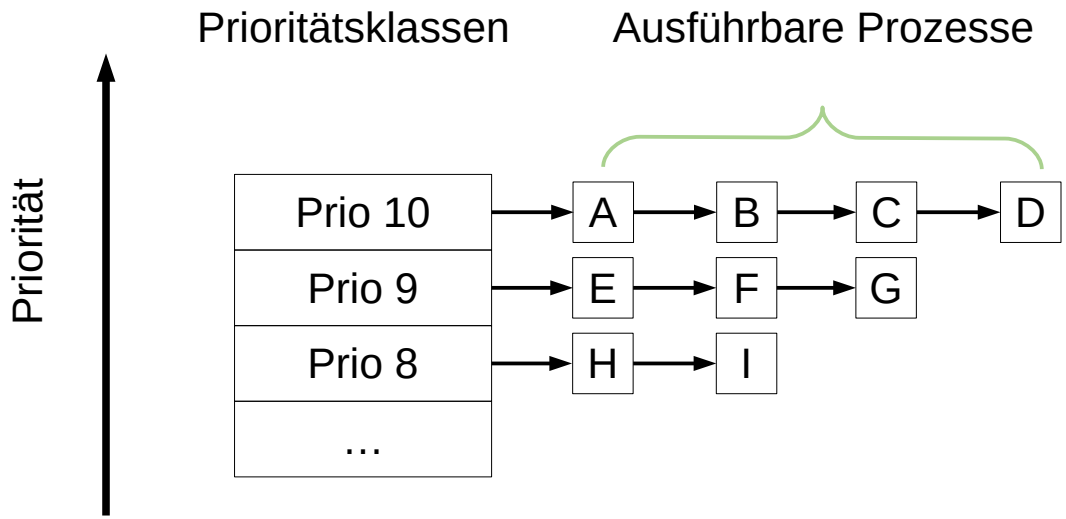
- Scheduler verwaltet eine Liste von Prozessen
- Nach der Ausführung wird der Prozess an das Ende der Liste gereiht und der nächste Prozess in der Liste ausgeführt
- Einfach zu implementieren
  - Wurde von Linux in frühen Versionen (0.x – 1.x) verwendet

Prioritäts-Scheduling

- Prozesse erhalten eine Priorität
  - Höhere Priorität bedeutet, dass diese Prozesse öfter ausgeführt werden und damit auch mehr Rechenzeit bekommen
- Statisch
  - Priorität wird festgelegt und bleibt unverändert
  - z.B. Webbrowser vs. periodischer E-Mail Check
  - z.B. Multiuser Systeme: General vs. Soldat
- Dynamisch
  - Priorität wird vom OS dynamisch angepasst
  - Setzt voraus, dass das OS z.B. verbrauchte Quantumzeit aufzeichnet
    - I/O-bound Prozesse bekommen eine höhere Priorität → Werden schneller/häufiger ausgewählt weil sie die CPU nur kurz belegen → Ziel Balance

Prioritäts-Scheduling - Beispiel

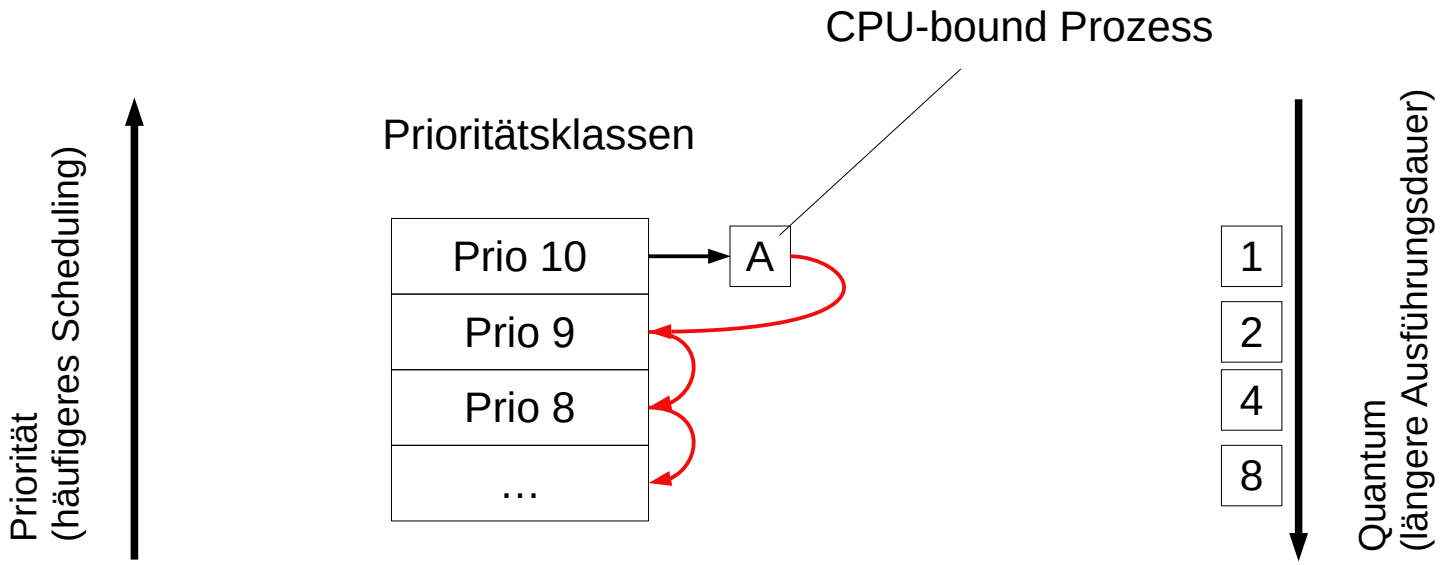
- f: Zeit (anteilig) des letzten Quantums, die für die Berechnung verwendet wurde
  - z.B. Quantum: 100; I/O-bound: 10/100 = 0,1; CPU-bound: 100/100 = 1
- 1/f: Priorität
- CPU-bound: Kleine Priorität, z.B. 1/1 = 1
- I/O-bound: Hohe Priorität, z.B. 1/0,1 = 10
- Gruppierung in Prioritätsklassen
  - Round Robin innerhalb dieser Klassen solange die Prozesse der Klasse rechenbereit sind
  - Ignorieren der anderen Klassen, Wechsel in die niederpriore Klasse erst, wenn keine Prozesse mehr rechenbereit sind
  - Nachteil: Verhungern von niederprioren Prozessen → Dynamische Anpassung der Priorität notwendig



Multilevel Feedback Queues (MLFQ)

- Bis jetzt haben wir festgestellt, dass I/O-bound Prozesse eher eine höhere Priorität bekommen sollen
- Erweiterung nun bei CPU-bound Prozessen
  - Besser diese für längere Zeit rechnen zu lassen, als oft für kurze Zeit
  - Minimierung der Context Switches
- Daher: Prozessen in unterschiedlichen Prioritätsklassen wird eine unterschiedliche Anzahl an Quanten zugeteilt
  - Höchste Prio Klasse: 1 Quantum
  - Nächste Prio Klasse: 2 Quanten
  - Nächste Prio Klasse: 4 Quanten
  - …
- Je nachdem wie viel ein Prozess rechnet oder auf I/O wartet, wird dieser zwischen den Klassen verschoben und
  - damit die Priorität geändert
  - die Menge an CPU Zeit, die der Prozess erhält, dynamisch angepasst

MLFQ - Beispiel



- Beispiel: Prozess A benötigt 20 Quanten und kein I/O
  - Startet in Prioritätsklasse 10 und wird unterbrochen → kann nicht fertig rechnen
  - A wird daher nach unten gereiht (CPU-bound)
  - A wird das nächste Mal in Prioritätsklasse 9 gestartet
  - A bekommt daher weniger häufig die CPU
  - A bekommt dafür die CPU für eine längere Zeit
- Bsp: Was würde passieren wenn A plötzlich beginnt I/O Operationen durchzuführen?
- Bsp: Vergleich Round Robin: 5 vs. 20 Context Switches

Implementierungen

- OS verwenden oft Varianten/Mischformen von MLFQ
  - Mac OS X, (Net|Free)BSD + Prioritäten
- Windows
  - 3.1x: Non-preemptive Scheduler
  - 95: Preemptive Scheduler
  - >= Vista: MLFQ + Magic (Closed Source)
    - Windows Scheduler arbeitet mit Threads
    - Threads besitzen zusätzlich eine statische Priorität (kann von BenutzerIn oder im Code gesetzt werden)
    - Priorität kann aber auch vom OS angepasst werden
    - Ausnahme: Threads der Real Time Prioritätsklasse

Implementierungen - Linux

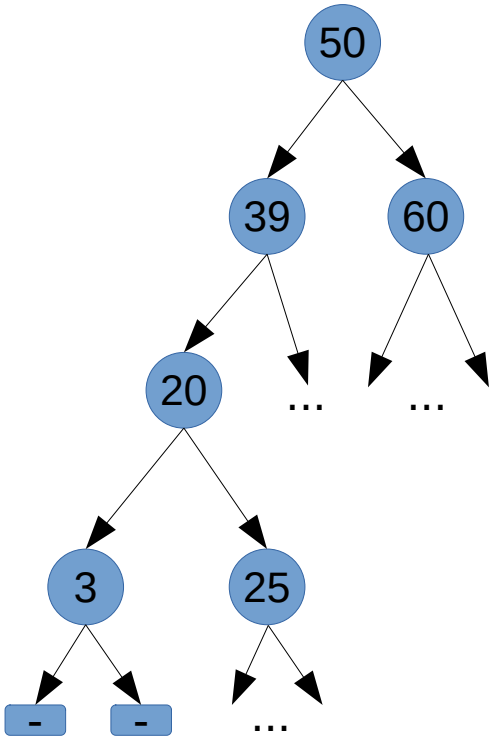
- Linux 0.x – 1.x: Round Robin
- Linux 2.2: Erweiterungen Scheduling Klassen und SMP Support
- Linux 2.4: O(n) Scheduler
  - Iteration über alle Prozesse der Run Queue → O(n)
  - Auswahl des nächsten Prozesses aufgrund einer Heuristik
  - Mitnahme von nicht verbrauchtem Quantum (wenn Prozess blockiert)
- Linux 2.6: O(1) Scheduler
  - Auswahl des nächsten Prozesse aus der Run Queue in O(1)
  - Allerdings viele Heuristiken zur Feststellung ob ein Prozess CPU-bound oder I/O-bound ist → komplex
- > Linux 2.6.23: Completely Fair Scheduler (CFS)
  - Verhalten kann durch Scheduling Policies verändert werden (neben dem standard, interaktiven Scheduling gibt es auch noch FIFO, Round Robin und Deadline)
  - Deadline Policy ist für Echtzeitsysteme ausgelegt
- Alternative: Brain Fuck Scheduler (BFS) / MuQSS (Con Kolivas)

Completely Fair Scheduler

- *CFS basically models an 'ideal, precise multitasking CPU' on real hardware. CFS's design is quite radical: it does not use runqueues, it uses a time-ordered rbtree to build a 'timeline' of future task execution“* - Ingo Molnár
- „Fair“ weil jeder Prozess Anspruch auf gleich viel Prozessorzeit hat
  - Jeder Prozess bekommt also den selben Anteil an CPU Zeit unter Berücksichtigung der anderen Prozesse (Maximum Execution Time)
  - Maximum Execution Time für einen Prozess wächst mit der Dauer, die er wartend verbringt („Sleeper Fairness“)
  - Die Zeit vergeht allerdings für low-prio Prozesse schneller
  - Da sich die Anzahl an Prozessen laufend ändert, gibt es kein fixes Quantum für einen Prozess
- Für jeden Prozess wird die Zeit, die er ausgeführt wird, aufgezeichnet (`vruntime`)
- Es gibt keine spezielle Behandlung oder Heuristik zur Feststellung der Interaktivität eines Prozess (→ einfacher als O(1) Scheduler)

Completely Fair Scheduler

- Die Datenstruktur (Run Queue) für die Prozesse ist ein Rot-Schwarz Baum, der nach der `vruntime` sortiert ist
  - Knoten mit der kleinsten `vruntime` befinden sich im Baum links unten
- Es wird immer der linkeste Knoten (Prozess) entnommen und als nächstes ausgeführt
  - Der Prozess wird für Maximum Execution Time ausgeführt
  - Die `vruntime` des Prozesses wird aktualisiert und der Prozess nach der Ausführung wieder im Baum nach der `vruntime` sortiert eingefügt
  - Während ein Prozess ausgeführt wird, wächst die Maximum Execution Time der anderen Prozesse
  - Damit gibt es einen neuen linken Knoten welcher als nächstes entnommen und ausgeführt wird
- CFS wurde über die Jahre weiterentwickelt
  - Auto-group von Prozessen, die logisch zusammengehören, um Fairness zu steigern
  - Verbesserungen im Bereich Multicore Performance



A Decade of Wasted Cores[1]

- Eine Run Queue pro Prozessor/Core
  - Bei einer einzigen globalen Run Queue wäre Locking notwendig → Performance
- CPU Affinity für Prozesse
  - Prozesse werden einer CPU/Core zugeordnet
    - Um Cache Misses zu reduzieren
    - Das Verschieben eines Prozesses von einer Run Queue in eine andere ist eine teure Operation
- Load Balancing zwischen CPUs/Cores
  - Ziel ist, alle CPUs/Cores möglichst gleichmäßig auszunutzen
  - Daher wird periodisch ein Load Balancing Algorithmus ausgeführt, der eine Umverteilung der Prozesse, falls benötigt, durchführt

[1] https://people.ece.ubc.ca/sasha/papers/eurosys16-final29.pdf

Zusammenfassung

- Prozesse
- Threads
- Prozesszustände
- Prozessverwaltung
- Scheduling
- Schedulingalgorithmen

Betriebssysteme

Speicherverwaltung

Susanne Schaller, MMSc
Andreas Scheibenpflug, MSc

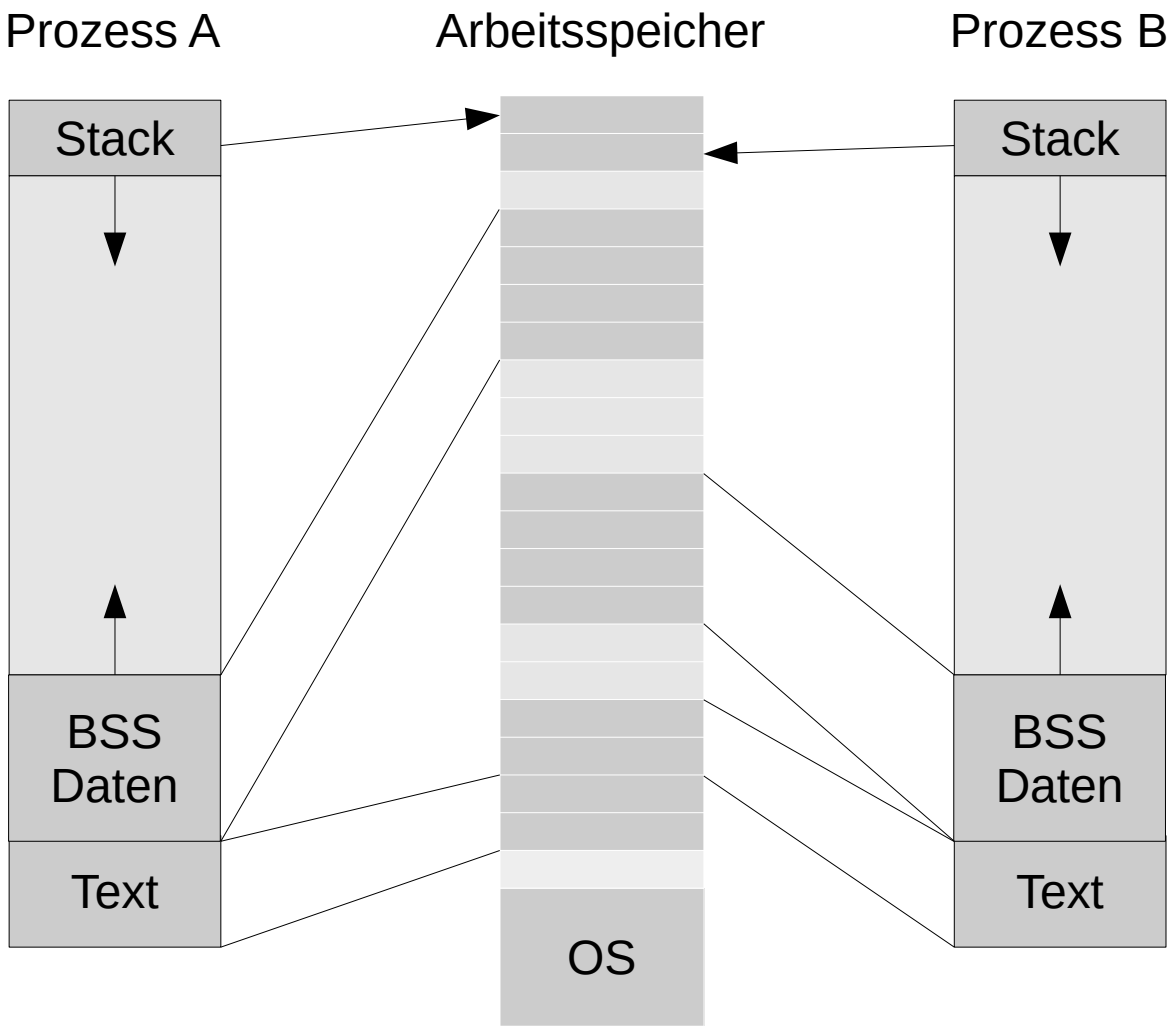
SE Bachelor (BB/VZ), 2. Semester

Speicherverwaltung

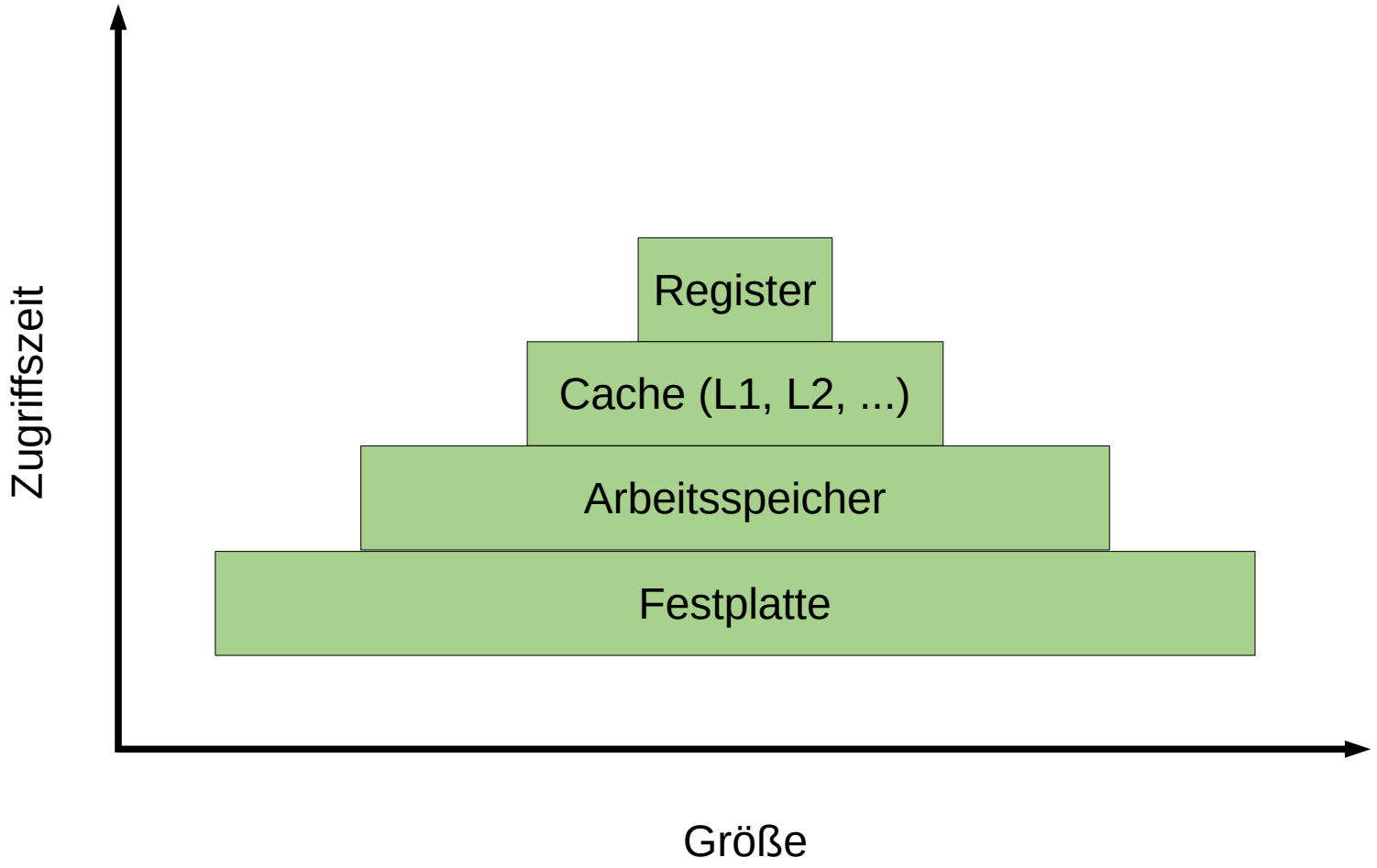
- Prozesse benötigen Arbeitsspeicher für
  - Code (Instruktionen)
  - Daten (Variablen, Heap, Stack)
- OS verwaltet den Hauptspeicher (RAM)
  - Zuteilung von Speicher zu Prozessen
  - Allokieren / Freigeben von Speicher
  - Auslagern (Paging) auf die Festplatte
  - Optimale Ausnutzung des Speichers

Speicherlayout (Linux)

- Text Segment
  - Instruktionen/Code
- Daten
  - Initialisierte Daten, z.B. statische Variablen, globale Variablen
- BSS
  - „Block Started by Symbol“
  - Nicht initialisierte Variablen
  - Heap
- Stack
  - Lokale Variablen, Rücksprungadressen, Argumente
  - Stack Pointer zeigt auf die Startadresse dieses Bereichs



Speicherhierarchien



Abstraktion

- Ein OS stellt auch mit der Speicherverwaltung eine Abstraktionsschicht für Programme im User Space zur Verfügung
- Virtual Memory ist dabei die gebräuchlichste Technik
- Wir sehen uns zur Einleitung aber auch noch weitere Möglichkeiten an
  - Keine Abstraktion
  - Adressräume

Keine Abstraktion

- Fand Anwendung bei z.B. Mainframe Computer vor 1960 oder PCs vor 1980
- Findet noch Anwendung bei manchen Eingebetteten Systemen
- Jede Operation in Bezug auf Arbeitsspeicher, die eine Anwendung durchführt, wird direkt 1:1 auf dem physischen Speicher ausgeführt
  - z.B. JMP 28 springt an die tatsächliche Speicheradresse 28
- Konsequenz: Es kann nur ein Prozess im Speicher gehalten werden da
  - Speicherzugriffsverletzungen (Segfault) nicht festgestellt bzw. verhindert werden können
  - Adressierung bei mehreren Prozessen im Speicher nicht mehr funktionieren würde
- Mögliche Lösungsansätze
  - Anpassung von Adressen relativ zur Startspeicheradresse
  - Auslagern des Speicherabbilds von inaktiven Prozessen auf die Festplatte

Keine Abstraktion - Beispiel

Prozesse	Speicheradressen
Prozess 2	
JMP 8	1016
	1012
	1008
	1004
	1000
	...
	...
Prozess 1	
	20
JMP 8	16
	12
	8
	4
	0

Adressräume

- Jeder Prozess hat seinen eigenen Adressraum
  - d.h. jedem Prozess wird eine Menge an Adressen zugewiesen, die dieser verwenden darf
- Dynamic Relocation
  - Base Register ist die Startadresse des Programms
  - Limit Register ist die Programmlänge
  - Greift das Programm auf Speicheradressen zu, wird die Adresse mit dem Base Register addiert und überprüft, ob die Endadresse des Programms nicht überschritten wird
  - Nachteil: Zusätzliche Operationen (Addition + Vergleich) bei jedem Speicherzugriff

Dynamic Relocation - Beispiel

Prozesse	Speicheradressen
Prozess 2	
JMP 8	1016
	1012
	1008
	1004
	1000
	...
	...
Prozess 1	
	20
JMP 8	16
	12
	8
	4
	0

1000 + 8 &&
1008 <= 1016
→ JMP 1008

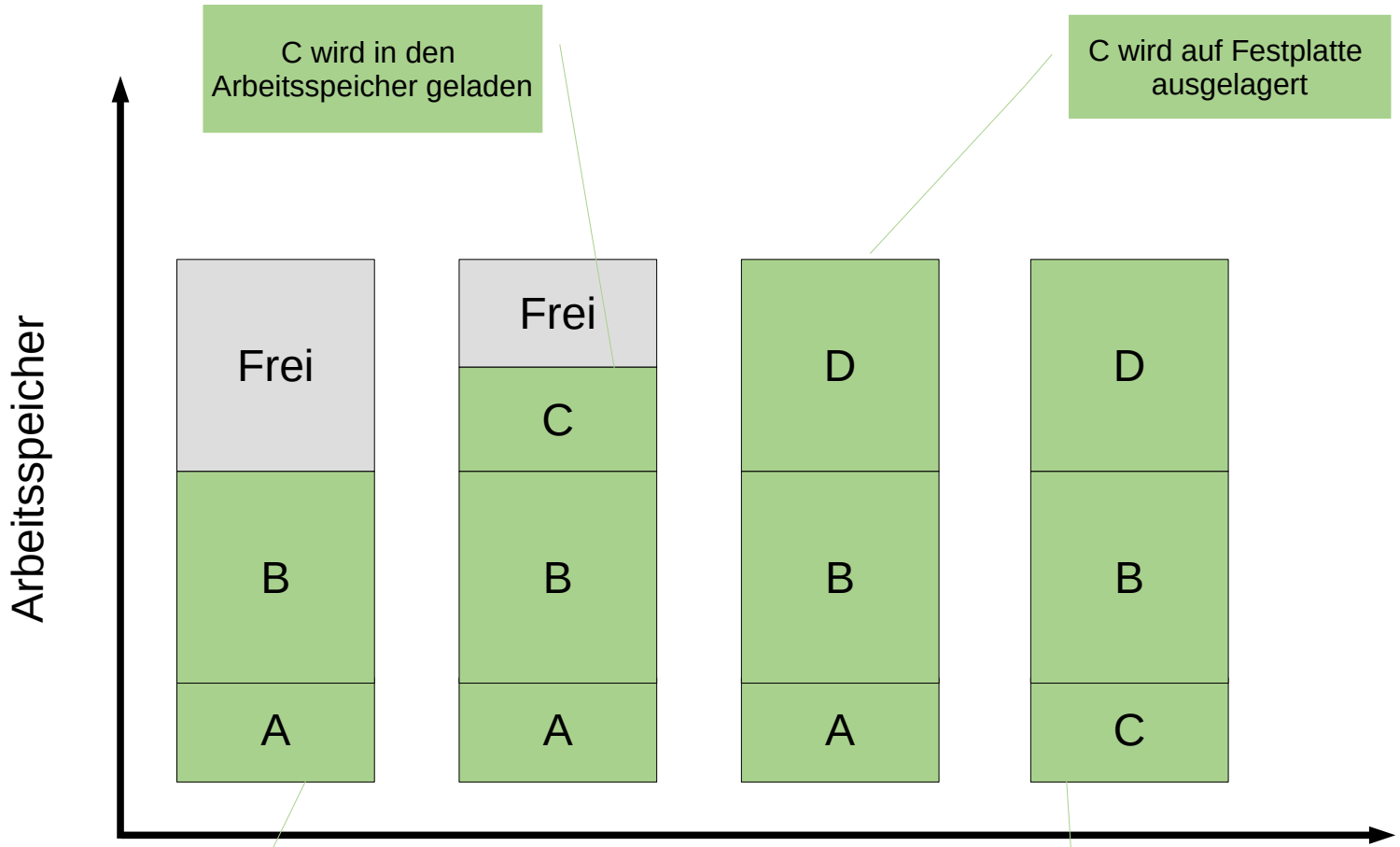
Limit Register

Base Register

Swapping

- Es ist unter Umständen nicht möglich, alle Prozesse im Arbeitsspeicher zu halten
  - Wenn viele Prozesse gleichzeitig laufen und/oder Prozesse große Mengen an Arbeitsspeicher benötigen
- Swapping bedeutet,
  - den Speicher des kompletten Prozesses auf die Festplatte auszulagern, wenn dieser nicht läuft
  - den Speicher des kompletten Prozesses wieder von der Festplatte in den Arbeitsspeicher zu laden, wenn der Prozess fortgesetzt wird
- Damit wird Arbeitsspeicher für andere laufende Prozesse frei

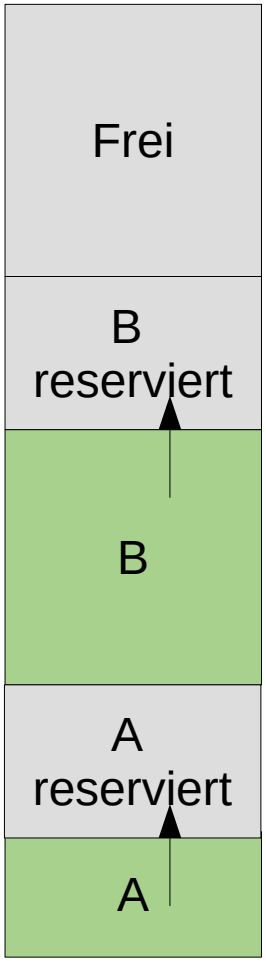
Swapping - Beispiel



malloc anyone?

- Bisher haben wir einem Programm eine fixe Größe im Arbeitsspeicher zugewiesen
- Es muss aber auch dynamische Speicherallokierung beachtet werden
  - Hat einen Einfluss auf die Aufteilung des Arbeitsspeichers
  - Anders gesagt: Das Datensegment eines Prozesses kann zur Laufzeit wachsen und schrumpfen
- Wenn also ein Prozess über seine Grenzen hinauswachsen würde, muss er im Speicher umgesiedelt werden
  - Schlecht für die Performance
  - Daher ist es eine bessere Idee, Lücken zwischen den Speicherbereichen von Prozessen zu schaffen
  - Beim Swappen wird aber nur der tatsächlich belegte Teil des Speichers auf die Festplatte geschrieben

Swapping mit Lücken



- Wenn reservierter Speicher ausgeht:
  - Relocation in einen Speicherbereich der groß genug ist
  - Swapping und Pausieren, bis dass ein größerer Speicherbereich frei ist
  - Beenden (Out Of Memory Exception)

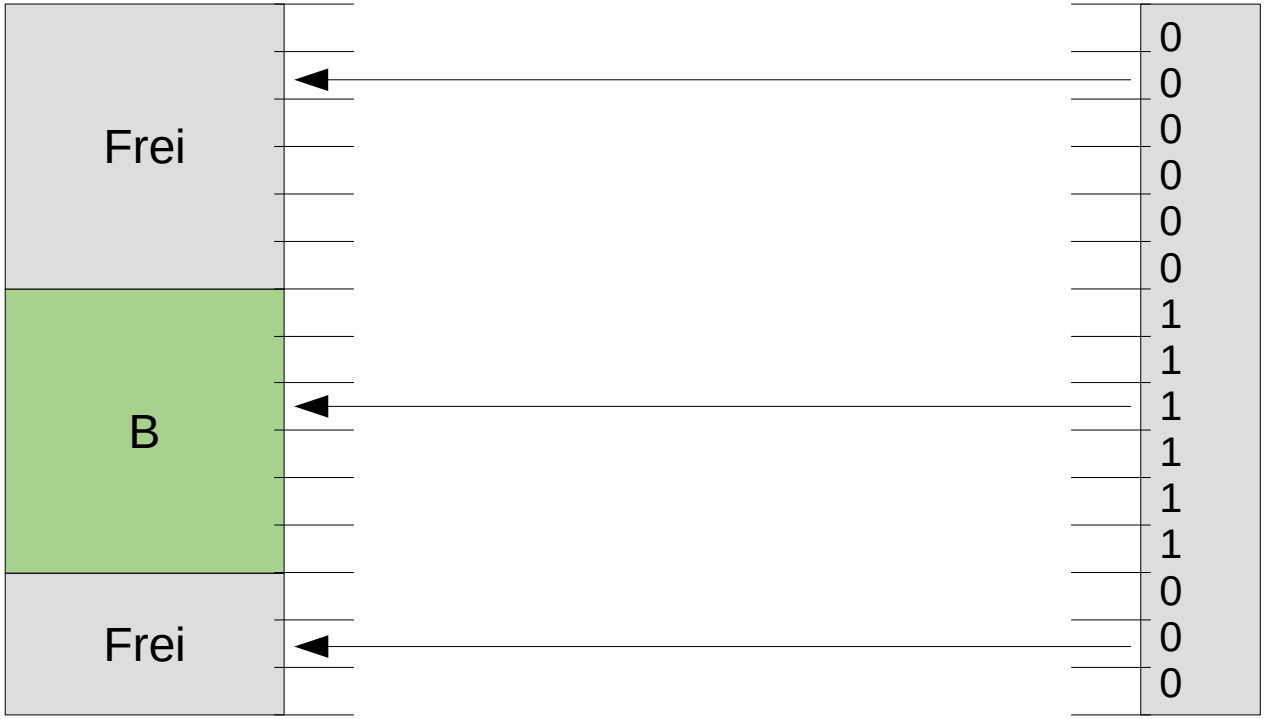
Keeping Track of Memory

- Wie wir im Swapping Beispiel gesehen haben, trifft das OS Entscheidungen, an welche Stellen im Arbeitsspeicher Programme geladen werden
- Voraussetzung dafür ist, dass das OS weiß,
  - wo wie viel Speicher frei ist
  - wie viel Speicher als Buffer (Lücke) freigehalten werden sollte
- Wir sehen uns im Folgenden zwei Strategien zum Verwalten der Speicherbelegung an
  - Bitmaps
  - Verkettete Listen

Bitmaps

- Jedes Bit in einer Bitmap ist einer Allocation Unit zugeordnet
  - Allocation Unit: Adressierbare Speichereinheit
  - 0 bedeutet Allocation Unit ist frei
  - 1 bedeutet Allocation Unit ist belegt
- Größe einer Allocation Unit hat einen Einfluss auf die Größe der Bitmap
  - Umso kleiner die Allocation Unit, umso größer wird die Bitmap
  - Bsp: AU 4 Bytes, d.h. 1 Bit pro 4 Bytes, bei 2 GB RAM würde also die Bitmap 64 MB groß sein

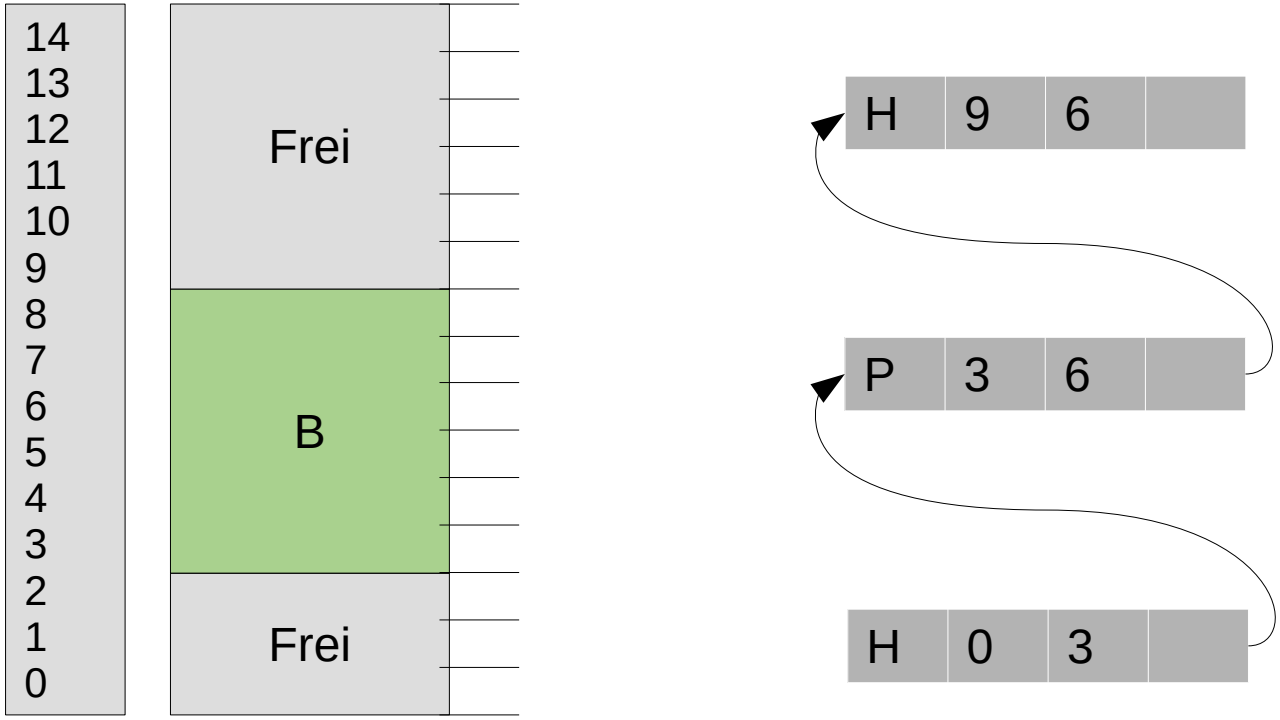
Bitmaps - Beispiel



Verkettete Listen

- Anstelle einer Bitmap wird eine verkettete Liste mit 2 Knotentypen (Process, Hole) verwendet
- Knoten speichern
  - Typ
    - P – Process: Speicherbelegung eines Prozesses
    - H – Hole: Block aus freiem Speicher
  - Startadresse
  - Länge
- Länge der Liste und der Speicherbedarf der Liste hängt damit von der Anzahl an Prozessen und freien Speicherblöcken ab
  - Im Gegensatz zur Bitmap, die eine konstante Größe (abhängig von der Größe des Arbeitsspeichers) hat

Verkettete Listen - Beispiel



Verkettete Listen - Varianten

- Wird ein neuer Prozess in den Speicher geladen, muss entschieden werden, welcher freie Platz verwendet werden soll
  - First Fit: Durchlaufen der Liste und Verwenden des ersten passenden Platzes
  - Best Fit: Durchlauf der Liste und Verwenden des kleinsten freien Platzes
  - Separate Listen für Prozesse und Holes als Optimierung
- Jede Variante hat Auswirkung auf Performance und die Menge an verschwendetem Arbeitsspeicher
  - Best Fit tendiert z.B. überraschender Weise dazu, kleine Speicherlöcher zu produzieren
  - Best Fit ist langsamer beim Speicherzuweisen als First Fit
  - First Fit tendiert zu größeren Speicherlücken

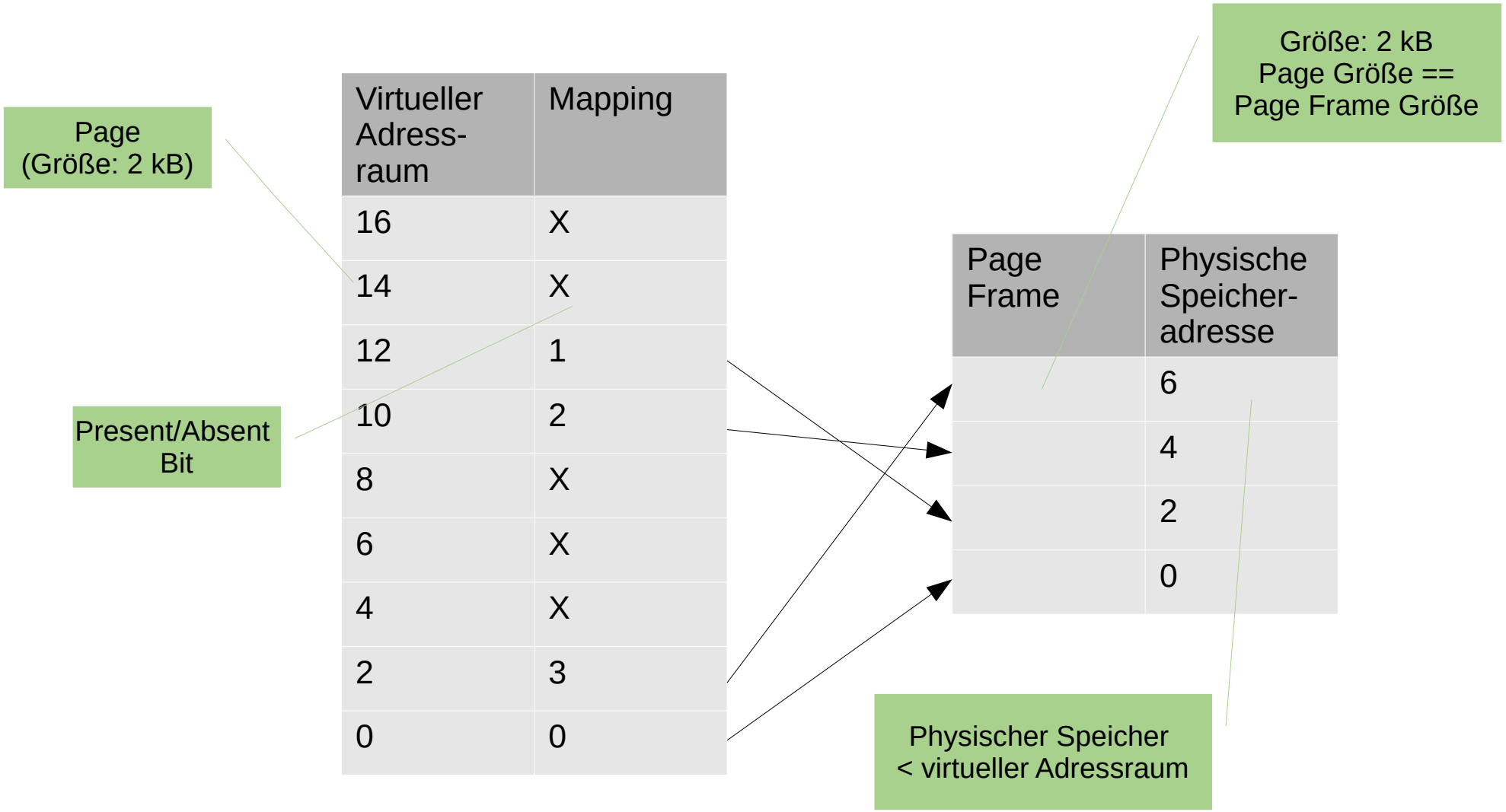
Virtual Memory

- Jeder Prozess hat seinen eigenen Adressraum
- Der Adressraum ist unterteilt in Seiten (Pages)
  - Eine Page ist eine aufsteigende Folge von Adressen
  - Pages werden auf den physischen Speicher gemappt
- Während ein Prozess ausgeführt wird, müssen sich nicht alle Pages des Prozesses im Hauptspeicher befinden
- Hardware führt das Adressmapping durch, wenn sich eine referenzierte Speicheradresse schon im physischen Speicher befindet (siehe Base/Limit Register)
- Befindet sich die referenzierte Speicheradresse nicht im Hauptspeicher, wird das OS benachrichtigt (Page Fault, wird vom Prozessor (MMU) ausgelöst) und lädt die Page, die die Adresse enthält
- Vorteil: Es kann eine größere Anzahl an Prozessen gleichzeitig im Speicher gehalten werden
  - Ist z.B. das Laden einer Page für einen Prozess notwendig, kann in der Zwischenzeit ein anderer, wartender Prozess ausgeführt werden

Virtual Memory

- Programme referenzieren virtuelle Adressen (vom Compiler generiert) in einem virtuellen Adressraum
- Die MMU (Memory Management Unit, Teil des Prozessors)
  - Mappt die virtuelle Adresse auf die physische Adresse
  - Stellt fest, ob die Page, die eine virtuelle Adresse enthält, sich im Hauptspeicher befindet (ob die Page einen korrespondierenden Page Frame hat)
  - Löst einen Page Fault aus, wenn sich die Adresse nicht im Hauptspeicher befindet
- Paging ist
  - das Laden von benötigten Pages von der Festplatte in den Arbeitsspeicher
  - das Auslagern von nicht benötigten Pages aus dem Arbeitsspeicher auf die Festplatte
  - Aufgabe des OS ist es, dass
    - benötigte Pages nachgeladen (eingelagert) werden
    - Page Frames auf die Festplatte ausgelagert werden, um Arbeitsspeicher freizugeben

Virtual Memory - Beispiel



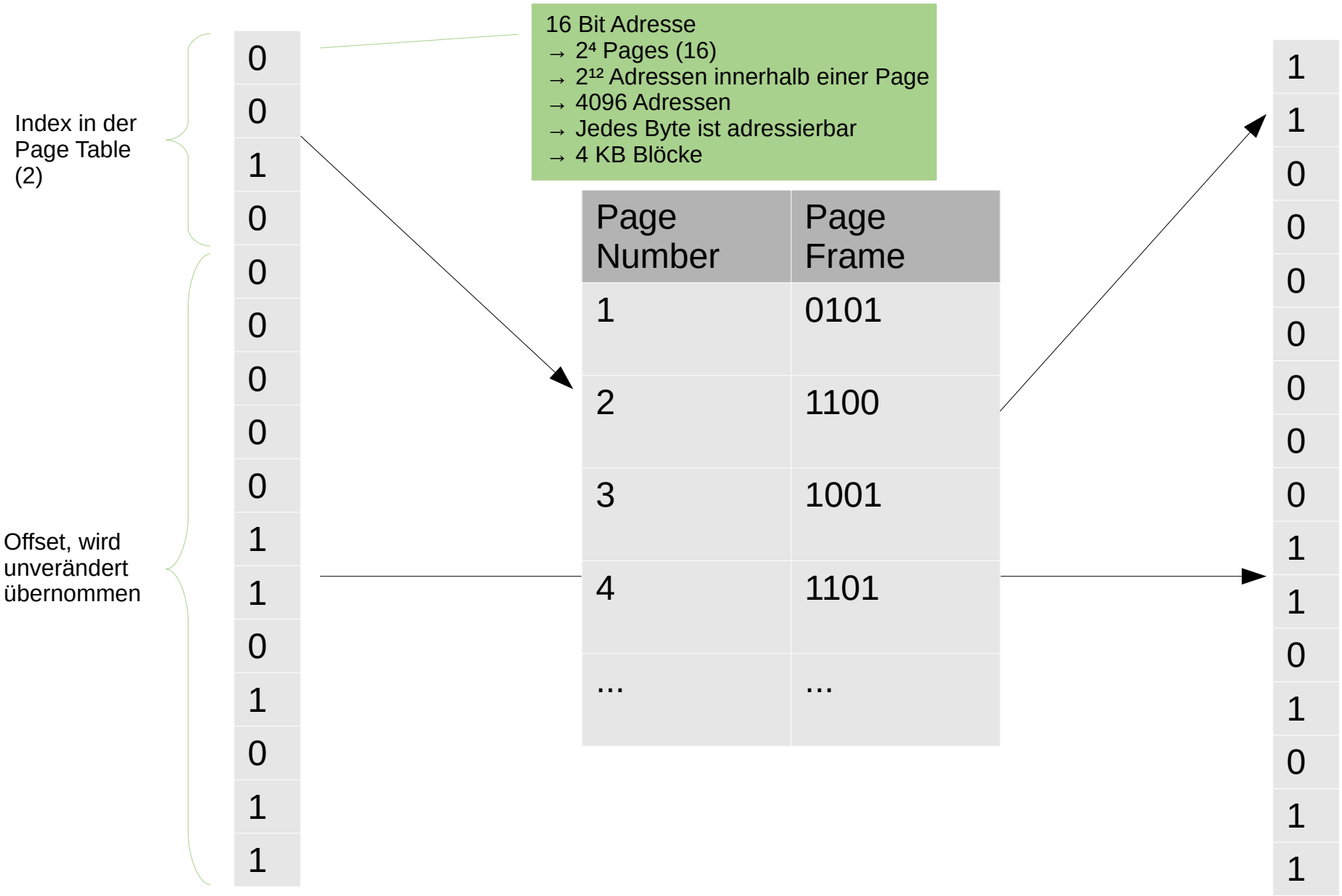
Virtueller Adress-raum	Mapping
16	X
14	X
12	1
10	2
8	X
6	X
4	X
2	3
0	0

Page Frame	Physische Speicher-adresse
	6
	4
	2
	0

Page Table

- Page Table enthält
  - das Mapping zwischen Page und Page Frame (virtuellen und physischen Adresse)
  - Zusatzinformationen (z.B. Present/Absent Bit,...)
- Adresse besteht aus 2 Teilen
  - Page Number: Wird in der Page Table als Index für den Eintrag der Page Frame Nummer verwendet
  - Offset: Für die Adressierung innerhalb des Frames, wird unverändert übernommen

MMU – Vereinfachtes Beispiel



Page Table Einträge

- Tatsächlich enthält ein Eintrag in der Page Table mehr Information
  - Page Frame Number
  - Present/Absent Bit:
    - 0: Page ist nicht gemappt → Page Fault
    - 1: Page ist gemappt und es kann zugegriffen werden
  - Protection: Lese- und oder Schreibzugriff erlaubt
  - Modified: Wurde in die Page geschrieben, wird das Modified Bit gesetzt. Wurde die Page verändert, muss sie beim Freigeben auf die Platte zurückgeschrieben werden. Andernfalls ist keine Aktion des OS nötig
  - Referenced: Wird gesetzt, wenn eine Adresse von einer Aktion referenziert wird. Damit weiß das OS, dass diese Page ein schlechter Kandidat zum Freigeben ist, da sie noch, oder bald, verwendet wird
- Aber keine Information zu z.B. Speicherort auf der Festplatte, da dies Aufgabe des OS ist

Performance Aspekte

- Auflösen von virtuellen (auf physische) Adressen muss schnell sein
  - Wird bei jedem Speicherzugriff benötigt
  - Sollte daher HW-unterstützt werden, da schnell
- Virtueller Adressraum ist groß
  - Die Page Tables können sehr groß werden
    - Mit 32 Bit können ~1 Million Pages (2^{20}) adressiert werden
    - Mit 64 Bit können sehr sehr viele Pages adressiert werden
  - Nicht wirtschaftlich, HW Register für komplette Page Tables anzubieten
- Lösung ist ein Kompromiss: Translation Lookaside Buffer

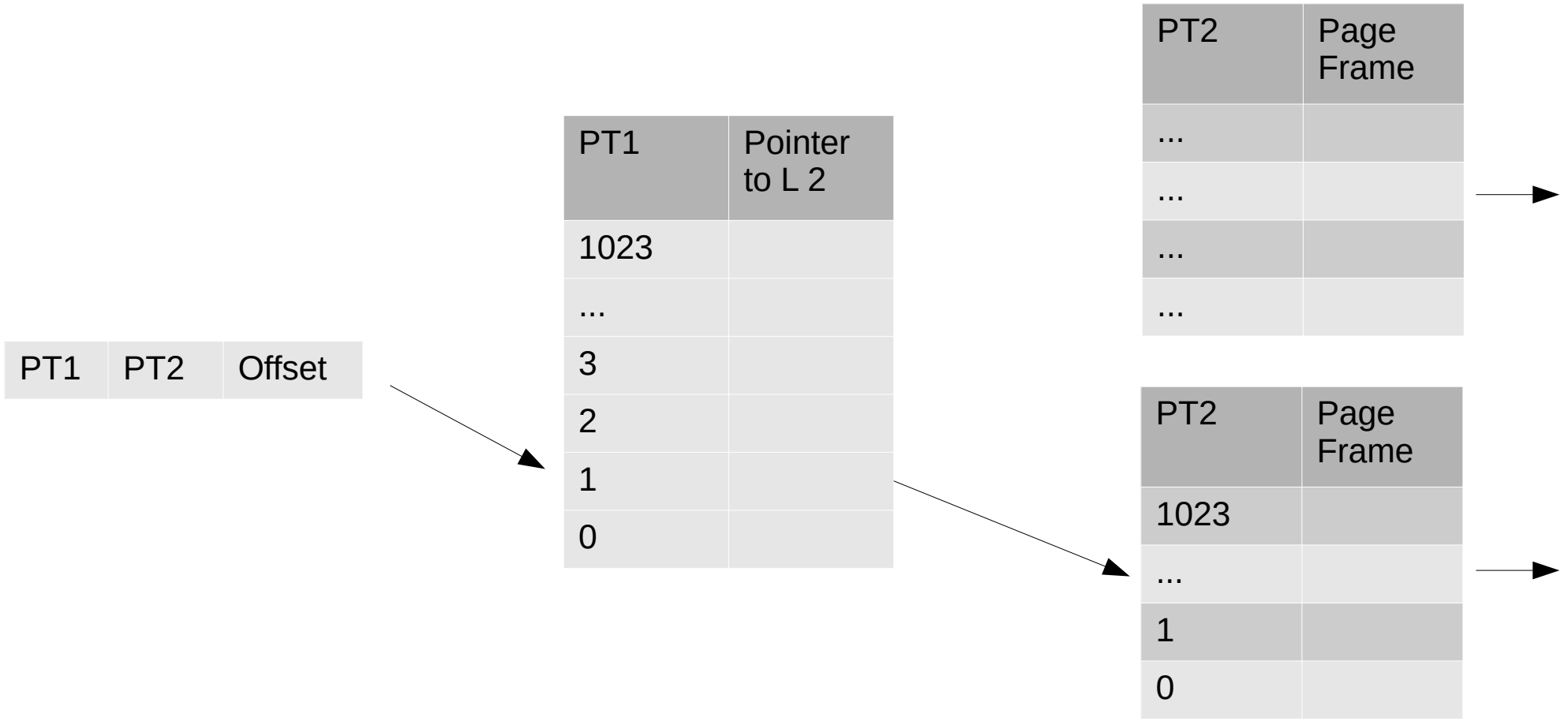
Translation Lookaside Buffer

- Technik zur Beschleunigung von Page Lookups
- Page Table ist im Arbeitsspeicher, TLB ist Teil der MMU und in HW implementiert
- Idee: Die meisten Programme greifen nur auf eine kleine Anzahl an Pages oft zu
- TLB ist Teil der MMU und ist eine kleine Mapping Tabelle, die den Umweg über die Page Table und damit den Arbeitsspeicher vermeidet
- Wird beim Anfordern einer Page das Mapping nicht im TLB gefunden, wird der Eintrag aus der Page Table geladen und der TLB aktualisiert

Multilevel Page Tables

- Technik für große virtuelle Adressräume
- Es müssen damit nicht immer alle Page Tables im Speicher gehalten werden
  - z.B. jene Page Tables für Bufferblöcke eines Programms, die für wachsenden Speicher reserviert sind, müssen nicht im Speicher gehalten werden
- Beispiel 32 Bit Adressen
  - Ersten 10 Bit für die Level 1 Page Table
    - Referenziert 4 MB Blöcke (→ 4 GB adressierbarer Speicher in 32 Bit Architekturen)
  - Nächsten 10 Bit für die Level 2 Page Table
  - Verbleibenden 12 Bit für den Offset
- z.B.
  - x86 (32 Bit): 2^{10} x 2^{10} x 2^{12} → 4294967296 Bytes → 4 GB
  - x68_64, 4 Levels à 512 Byte → 2^9 x 2^9 x 2^9 x 2^9 x 2^{12} → 2^{48} → 256 TB

Multilevel Page Tables



Inverted Page Tables

- „Inverted“, weil die Tabelle ein Mapping von Frame auf Page ist
  - Eine Zeile der Tabelle entspricht einem Frame des Arbeitsspeichers und der zugeordneten Page
- Damit ist die Größe der Tabelle konstant
- Problem: Suche nach virtuellen Adressen ist langsam
- Lösung
  - Verwendung des TLB
  - Hashing der virtuellen Adressen um Suchen zu beschleunigen
- z.B. Itanium, UltraSPARC, PowerPC

Ersetzungsstrategien

- Wird eine Page benötigt, die sich nicht im Arbeitsspeicher befindet
  - Muss dass OS diese von der Festplatte laden
  - Eventuell eine Page aus dem Speicher entfernen
  - Die entfernte Page eventuell auf die Festplatte schreiben, wenn diese verändert wurde
    - Oder keine Aktion durchführen, wenn der Eintrag auf der Festplatte aktuell ist weil die Page nicht verändert wurde
- Der Page Replacement Algorithmus führt die Auswahl einer geeigneten Page zum Ersetzen mit der nachzuladenden Page durch
  - Im Optimalfall wird jene Page, die zeitlich möglichst weit entfernt verwendet wird, ersetzt
  - Das Problem ist, dass diese Information nicht verfügbar ist

Not recently used (NRU)

- Voraussetzung: Zu jeder Page in der Page Table wird folgende Information gespeichert
  - Referenced (R Bit): Wird gesetzt, wenn eine Page gelesen oder verändert wird
  - Modified (M Bit): Wird gesetzt, wenn eine Page verändert wird
- Wird im Normalfall von der Hardware aktuell gehalten (auch Verwaltung durch OS möglich)
- Zu bestimmten Zeitpunkten (z.B. Timer Interrupt) wird das R Bit zurückgesetzt
  - Länger nicht verwendete Pages werden damit als nicht referenziert markiert

Not recently used (NRU)

- Daraus ergeben sich folgende Klassen
  - Klasse 0: nicht referenziert, nicht modifiziert
  - Klasse 1: nicht referenziert, modifiziert
  - Klasse 2: referenziert, nicht modifiziert
  - Klasse 3: referenziert, modifiziert
- NRU entfernt zufällig Seiten, beginnend mit der niedrigsten, nicht leeren Klasse
  - Besser nicht referenzierte Seiten zu entfernen, auch wenn diese modifiziert sind
- Vorteil: Einfach zu implementieren, für viele Fälle ausreichend

FIFO / Second Chance

- Pages werden in eine verkettete Liste eingefügt
  - Bei einem Seitenfehler wird der älteste Eintrag einfach entfernt
  - Nachteil: Eintrag könnte noch benötigt werden → untauglich in der Praxis
- Second Chance Algorithmus
  - Es wird zusätzlich das R Bit verwendet, um zu überprüfen, ob die Seite noch referenziert ist
  - Ist das der Fall, wird die Page übersprungen, das R Bit gelöscht und die Page an das Ende der Liste gestellt
  - Danach wird der nächsten Eintrag in der FIFO Liste überprüft
  - Sind alle R Bits gesetzt, wird aus Second Chance FIFO und der älteste Eintrag entfernt
  - Vorteil: Vernünftiger Algorithmus, praktische Implementierungen vermeiden Umhängen von Knoten in der Liste aus Performance Gründen (Clock Algorithmus)

Least Recently Used (LRU)

- Idee: Eine Seite, die in letzter Zeit häufig benutzt wurde, wird wahrscheinlich auch in Zukunft häufig benutzt
- Benötigt wird eine Liste aus allen Pages, geordnet nach Benutzungshäufigkeit
- Problem dabei: Aufwändig zu implementieren bzw. langsam
  - Liste müsste bei jedem Speicherzugriff aktualisiert werden
  - Benötigt für jede Page einen Zähler (Counter) und dieser muss auch bei jedem Zugriff aktualisiert werden
- Es würde also zumindest HW Unterstützung benötigt werden, um dies effizient umsetzen zu können → selbst dann nicht praxistauglich
- Daher Annäherung über Algorithmus Not Frequently Used (NFU)

Not Frequently Used (NFU)

- Für jede Page gibt es einen (Software) Zähler
  - Ist zu Beginn 0
  - Bei einem Timer Interrupt werden alle Seiten durchlaufen und bei jenen, die das R Bit gesetzt haben, der Zähler inkrementiert
  - Die Seite mit dem niedrigsten Zähler wird zur Ersetzung ausgewählt
- Nebeneffekt: Wurde auf eine Seite oft zugegriffen, wird diese unter Umständen sehr lange im Speicher gehalten obwohl sie nicht mehr benötigt wird → zeitliche Komponente fehlt
- Erweiterung
  - Zähler werden 1 Bit nach rechts geschoben
  - Inkrementieren erfolgt durch Addieren des R Bits zum höchstwertigen (linken) Bit des Zählers
  - Wird auch als „Aging“ bezeichnet
- Dadurch „vergessen“ Zähler ihre Werte nach einer bestimmten Zeit

Aging Beispiel

Zeit	R-Bit Page 0	R-Bit Page 1	Page 0	Page 1
0	1	0	10000000	00000100
1	1	0	11000000	00000010
2	1	0	11100000	00000001
3	1	0	11110000	00000000
4	0	1	01111000	10000000
5	0	1	00111100	11000000
6	0	0	00011110	01100000

Working Set Page Replacement Algorithmus

- Die Menge an Pages, die ein Prozess zu einem bestimmten Zeipunkt verwendet, wird als Working Set bezeichnet
  - Ist eine Teilmenge aller Pages, die ein Prozess über seine Lebensdauer verwendet
  - Prozesse verwenden meist nur einen kleinen Teil der theoretisch benötigten Pages
  - Working Set ändert sich über die Lebensdauer des Prozesses
  - Sind alle theoretisch benötigten Pages im Speicher, gibt es keine Seitenfehler
  - Trashing: Muss zwischen Instruktion eingelagert werden, weil Pages nicht im Speicher vorhanden sind, dauert das Einlagern länger als das tatsächliche Ausführen der Instruktion → schlecht für Performance
- Idee
  - OS merkt sich das Working Set eines Prozesses
  - Stellt sicher, dass das Working Set eingelagert ist, bevor ein Prozess durch den Scheduler fortgesetzt wird

Working Set Page Replacement Algorithmus

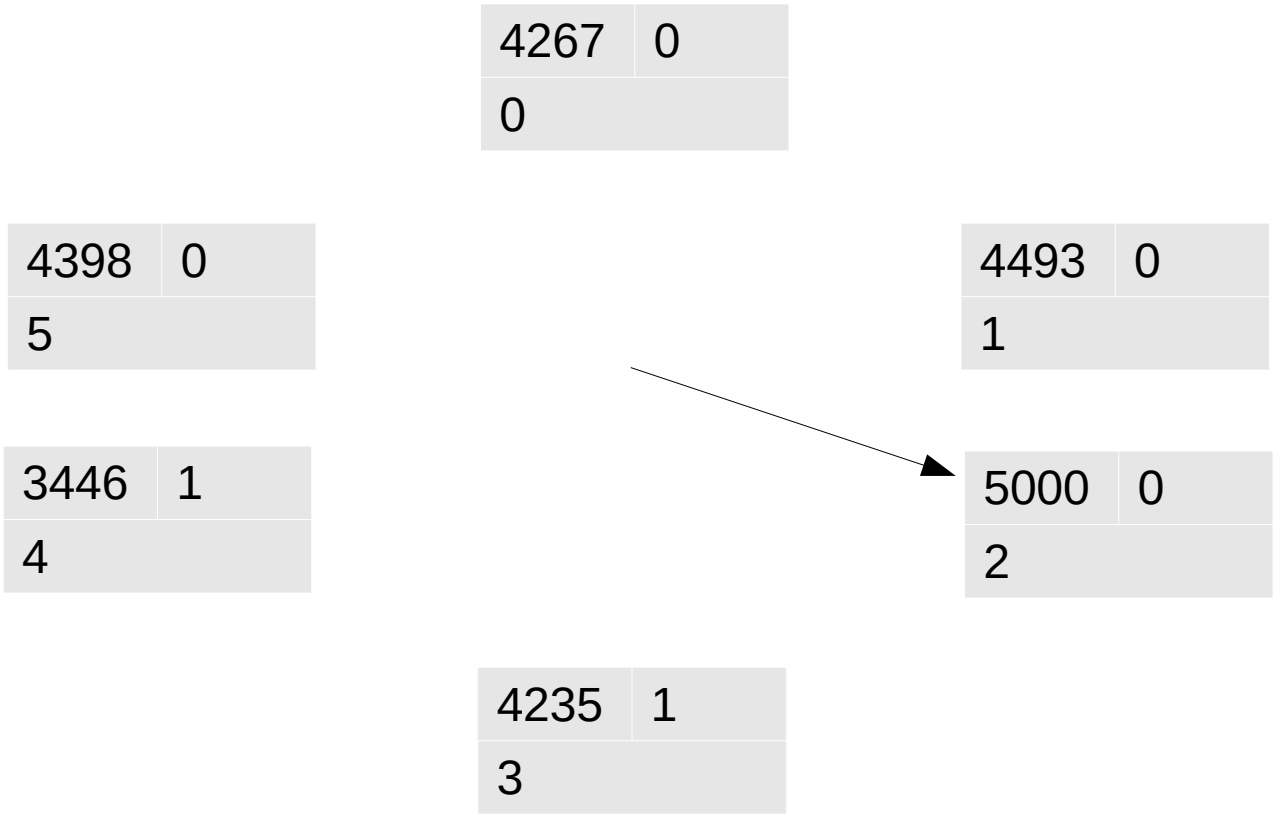
- Working Set: Menge an Pages, die in den letzten k Speicherzugriffen benutzt wurden
  - Protokollieren dieser Information würde Performance negativ beeinflussen
- Daher Vereinfachung: Menge an Pages, die in den letzten t Zeiteinheiten benutzt wurden
- Beim Ersetzen von Pages werden immer jene ersetzt, die nicht zum Working Set zählen
- Es wird zusätzlich zu den R/M Bits der Zeitpunkt des letzten Zugriffs (Alter) gespeichert
- Wir nehmen auch wieder an, dass das R Bit periodisch zurückgesetzt wird
- Seitenauslagerung wenn
  - R == 0 ist und
  - Alter > t
- Wenn R == 0 und Alter <= t wird Alter mit höchstem Alter ersetzt
- Tabelle wird immer komplett durchlaufen und das Alter aller Einträge aktualisiert

WSClock

- Problem des Working Set Page Replacement Algorithmus ist das Durchlaufen und Aktualisieren der kompletten Page Table bei einem Seitenfehler
- Daher: Kombination des Clock Algorithmus mit dem Working Set Page Replacement Algorithmus
- Datenstruktur ist eine ringförmige Liste (→ „Clock“)
  - Pages werden diesem Ring, wenn benötigt, hinzugefügt
  - Zusätzlich enthalten die Einträge den Zeitstempel der letzten Verwendung (und R / M Bit)
  - Es gibt einen Zeiger, der immer auf die älteste Page zeigt
    - Bei ihr wird bei einem Page Fault mit der Suche nach zu ersetzenden Pages begonnen
    - Ist R == 0 && M == 0 && Alter > t wird
      - die Page entfernt
      - die neue Page an diese Stelle geladen
      - Der Zeiger weiterbewegt
    - Ist R == 0 && M == 1 wird
      - Aus Performance Gründen das Schreiben der Page eingeplant
      - Der Zeiger aber weiterbewegt und die Suche fortgesetzt
    - Ist R == 1, wird R auf 0 gesetzt und der Zeiger weiterbewegt

WSClock - Beispiel

- Aktuelle Zeit: 5100
- t: 200
- Zeiger steht auf Page 2
  - R Bit ok, Alter nicht ok
- Zeiger wird auf Page 3 gestellt
  - R Bit nicht ok, Alter ok
  - Reset R Bit
- Zeiger wird auf Page 4 gestellt
  - Gleich zu Page 3
- Zeiger wird auf Page 5 gestellt
  - R Bit ok, Alter ok
  - Page wird ersetzt, vorausgesetzt M == 0

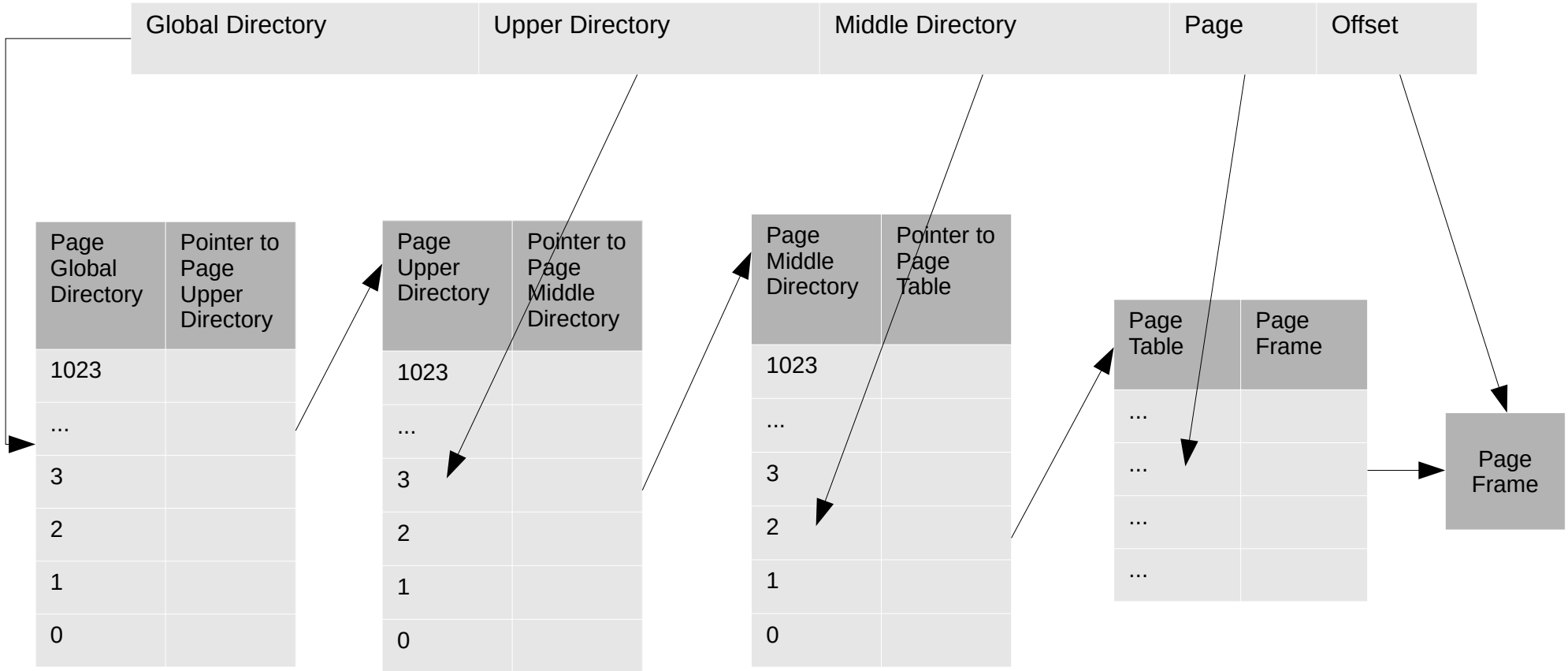


Implementierung - Linux

- Linux verwaltet Speicher mit einem Array bestehenden aus Page Deskriptoren (32 Byte → benötigt < 1% des Speichers)
  - Mapping Page → Page Frame
  - Enthält Zeiger zum Adressraum
  - Zeiger zu anderen Deskriptoren → Verlinkte Liste für z.B. freie Blöcke
- Speicher ist in Zonen unterteilt
  - Werden in einer Liste verwaltet (DMA, Normal, HighMem)
  - Enthält Informationen
    - Über Belegung, freie Bereiche
    - Active/Inactive Pages
    - Watermarks für Page Replacement Algorithmus
- Zusätzlich wird der Speicher in Nodes unterteilt
  - Für NUMA (Non-Uniform Memory Access) Systeme
  - Bei mehreren CPUs kann der Zugriff auf einen bestimmten Speicher je nach CPU unterschiedlich lange dauern
    - Muss beim Allokieren von Speicher beachtet werden
    - Es sollte nach Möglichkeit der schnellste Speicher der CPU, an die ein Prozess (CPU Affinity) gebunden ist, ausgewählt werden

Linux – Multilevel Page Tables

- Linux verwendet Multilevel Page Tabels mit 4 Ebenen (Levels)
- Update auf 5 Level für 64 Bit Systeme möglich



Linux - Allocators

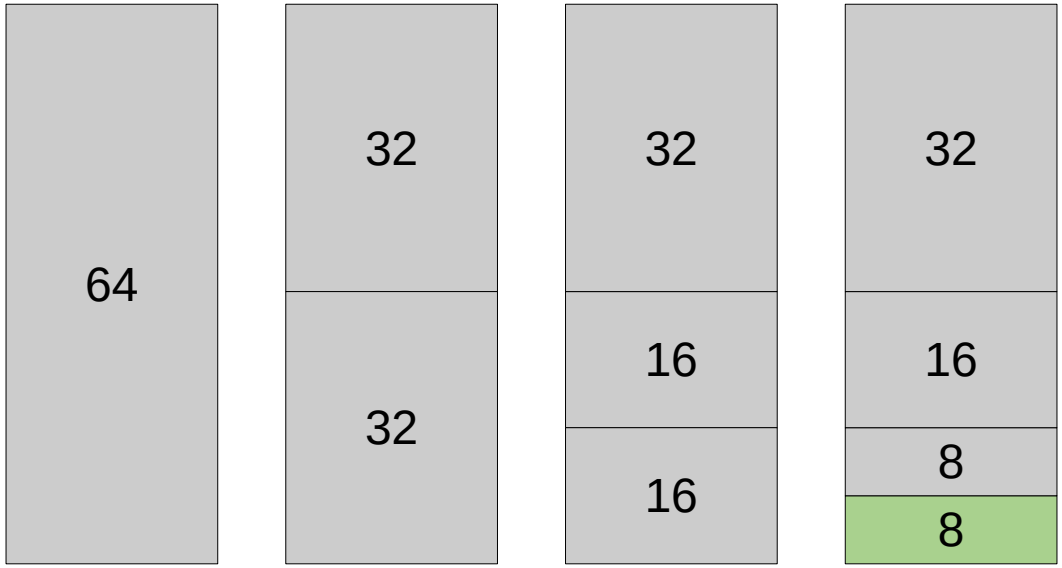
- Bis jetzt haben wir komplett ignoriert, welche Strategie ein OS bei der Zuteilung (Allokierung) von Arbeitsspeicher zu Prozessen verfolgen kann
  - Neu gestartete Prozesse müssen in den Speicher geladen werden (→ `fork()`)
  - Prozesse benötigen über ihre Lebensdauer unterschiedlich große Blöcke an Speicher (→ `malloc()`)
  - Prozesse geben unterschiedliche große Blöcke an Speicher über ihre Lebensdauer frei (→ `free()`)
- Linux verwendet einen leicht modifizierten Buddy Algorithmus in Kombination mit Slab Allocation

Exkurs: Allokierung und Fragmentierung

- Nehmen wir an, wir stellen Speicher in 4 KB Blöcken zur Verfügung
  - Was wenn wir nur 4 Byte allokieren?
    - Verschwendung → Interne Fragmentierung
  - Was wenn wir 2 MB allokieren
    - Dann benötigen wir 512 4 KB Blöcke
    - Nehmen wir an, dass es sich um das Working Set eines Prozesses handelt und der TLB 64 Adressen speichern kann → Performance Problem → TLB Trashing
  - Über die Zeit wird Speicher freigegeben, dadurch entstehen Lücken und eine Durchmischung aus belegtem und freiem Speicher
    - z.B. sind noch viele kleine, nicht durchgängige Blöcke an Speicher frei, können diese aber bei einer Anforderung nach einem großen Block nicht mehr verwendet werden, d.h. theoretisch wäre noch Speicher verfügbar, dieser ist aber nicht mehr verwendbar → Externe Fragmentierung
    - D.h. es müsste der Speicher neu angeordnet werden und die freien Blöcke zu einem Bereich zusammengefasst werden (Compaction)
- Es sollte also möglich sein, Blöcke mit variabler Größe zu allokieren
  - x86 CPUs und Linux unterstützen zusätzlich variable Page Größen

Linux - Buddy Algorithms

- Speicherblock, bestehend aus Pages, wird immer weiter unterteilt, bis die benötigte Blockgröße erreicht ist
- Bei Speicherfreigaben werden Blöcke wieder vereint
- Erweiterung Linux: Zusätzliche Listen mit unterschiedlichen Größen (1-Page Liste, 2-Page Liste, 4-Page Liste,…)
- Beispiel:
  - 64 Page Block
  - 8-Page Block wird benötigt
  - Was passiert wenn ein 16-Page Block benötigt wird?
  - Was passiert wenn danach der 8-Page Block freigegeben wird?
  - Was passiert wenn ein 33-Page Block benötigt wird?



Linux - Slab Allocation

- Buddy Algorithmus vermeidet externe Fragmentierung (leere Speicherlücken zwischen allokiertem Speicher), leidet jedoch unter interner Fragmentierung
- Zusätzlicher Slab Allocation Algorithmus reserviert mit dem Buddy Algorithmus Blöcke, welche für Objekt Caches verwendet werden
  - 1 Slab kann mehrere Objekte des selben Typs speichern, z.B. File Deskriptoren, Prozess Deskriptoren,...
  - Wird z.B. eine Datei geöffnet, wird aus einem Slab ein bereits allokierter (aber nicht benutzter) File Deskriptor verwendet
- Slab Allocation reduziert daher die Anzahl an Allokierungs- und Freigabe-Operationen für bestimmte Typen von Objekten

Linux – Ersetzungsstrategie

- Linux verwendet kein Preloading von Pages oder Working Sets
- Linux führt Demand Paging durch → es werden nur jene Pages geladen, die auch benötigt werden
- Linux verwaltet einen Pool an freien Pages und hält diesen auf einem vernünftigen Level
- Ein Daemon (kswapd) untersucht periodisch die Watermarks von Zonen und überprüft, ob noch genug freier Speicher in den Zonen vorhanden ist
  - Ist das nicht der Fall, wird der Page Frame Reclaiming Algorithmus ausgeführt
- Eigener Daemon (pdflush) schreibt
  - periodisch alte Pages mit gesetztem w Bit auf die Festplatte
  - Oder schreibt Pages auf die Festplatte, wenn der Arbeitsspeicher knapp wird (Watermark erreicht)

Linux - Page Frame Reclaiming Algorithm

- Jede Page besitzt 2 Bits
  - Active
  - Referenced
- 2 Listen (Ringe)
  - Active Pages
  - Inactive Pages
- Pages werden zwischen den Listen aufgrund ihrer gesetzten Bits verschoben
  - Unterschiedliche Arten von allokiertem Speicher werden unterschiedlich behandelt
  - Es wird eine Mischung aus LRU und Second Chance Clock Algorithmus verwendet
    - Referenced Bit wird bei jedem Durchlauf zurückgesetzt
- kswapd gibt zuerst inactive und unreferenced Pages frei
  - Erst wenn diese Möglichkeit ausgeschöpft ist, versucht er „schwierigere“ Pages freizugeben (z.B. Shared Pages zwischen Prozessen,...)

Zusammenfassung

- Memory Abstraktionen
- Virtual Memory
- Page Tables
- MMU & TLB
- Ersetzungsstrategien
- Linux & Allokierung

Betriebssysteme

OS Architekturen

Susanne Schaller, MMSc
Andreas Scheibenpflug, MSc

SE Bachelor (BB/VZ), 2. Semester

Architekturen

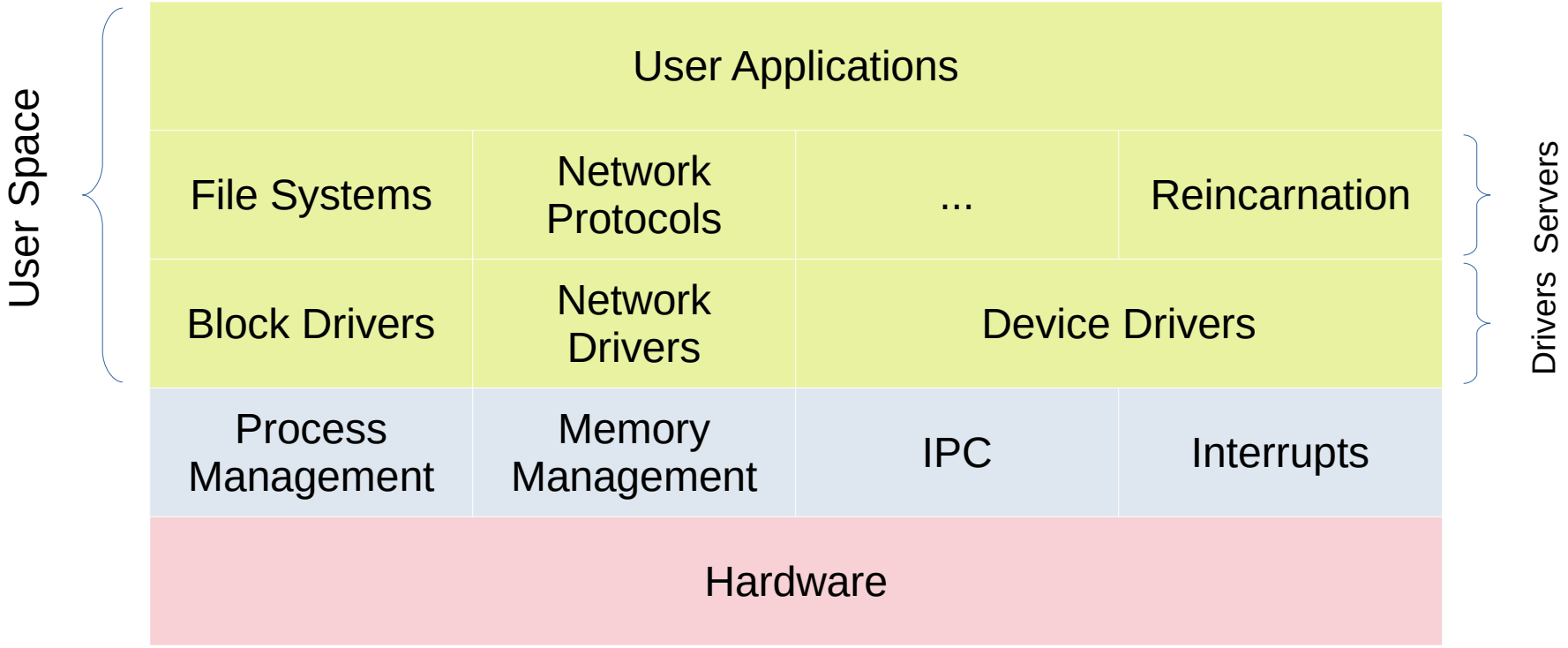
- Bis jetzt haben wir Betriebssysteme als eine große Komponente im Kernel Space (Ring 0) betrachtet
  - Es gibt auch andere Möglichkeiten
- Architekturen
  - Monolith
    - Alle OS Komponenten im Kernel Space
  - Microkernel
    - Nur die minimalen, notwendigsten Komponenten im Kernel Space
    - Alle anderen Komponenten befinden sich im User Space als Service/Server
  - Hybride
    - Mischung aus Monolith und Microkernel
    - Manche Komponenten laufen im User Space
    - Großteil der Komponenten befindet sich im Kernel Space

Monolith - Linux

<table>
<tr><td colspan="5">User Space</td></tr>
<tr><td colspan="5">System Calls / Device Files</td></tr>
<tr><td colspan="3">Virtual File System (VFS)</td><td>Memory Management</td><td>Process Management</td></tr>
<tr><td rowspan="2">Device Drivers</td><td rowspan="3">Sockets</td><td>File Systems (ext4,...)</td><td rowspan="3">Virtual Memory</td><td>Signals / IPC</td></tr>
<tr><td rowspan="2">Network Protocols (TCP/IP,...)</td><td rowspan="2">Process Handling</td></tr>
<tr><td rowspan="2">Interrupts</td><td>I/O Schedulers</td><td rowspan="2">Page Replacement</td><td rowspan="2">Scheduler</td></tr>
<tr><td>Network Device Drivers</td><td>Block Device Drivers</td></tr>
<tr><td colspan="5">Hardware</td></tr>
</table>

- Vorteil
  - Performance
- Nachteil
  - Bugs können Stabilität des ganzen OS gefährden

Microkernel



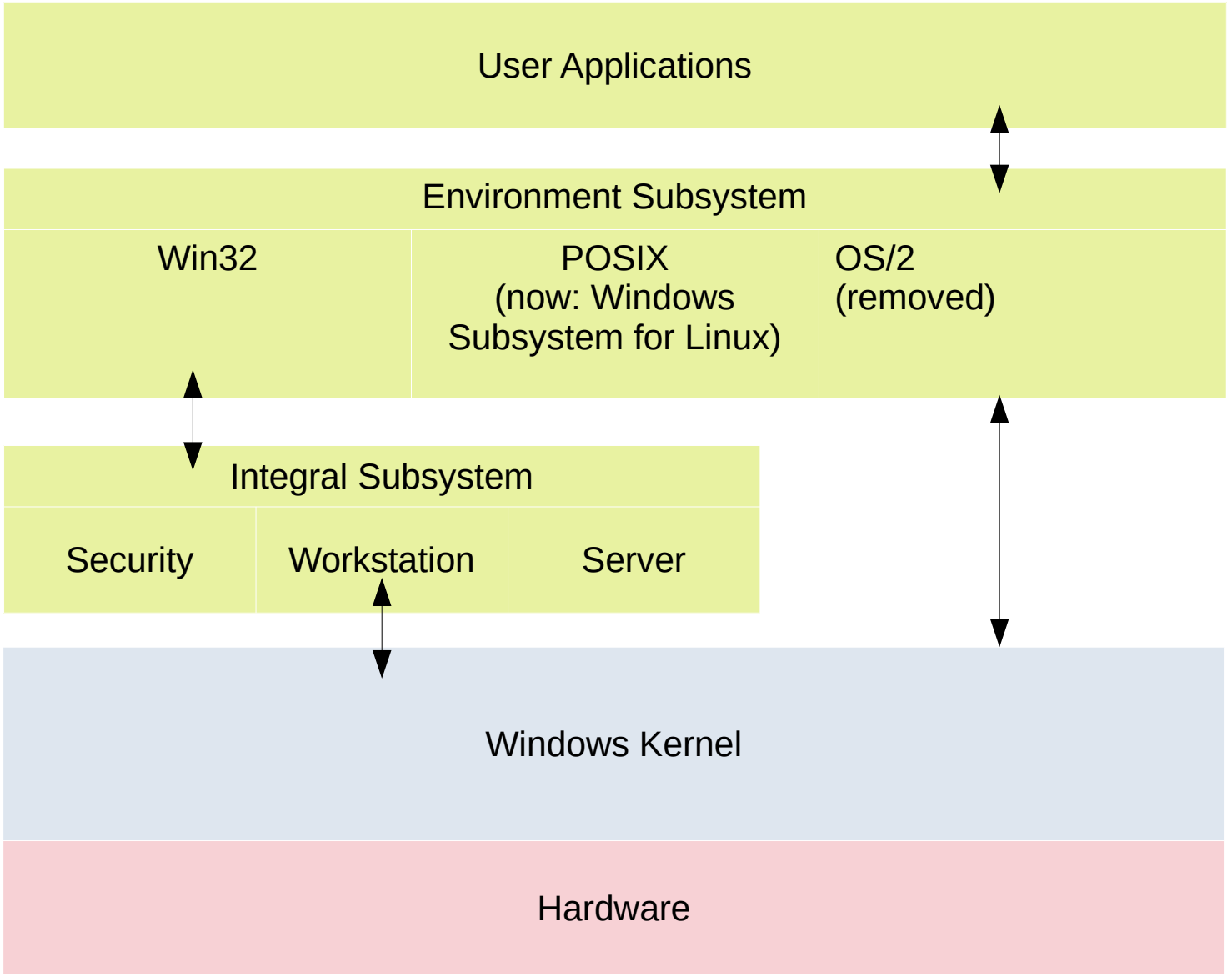
Microkernel

- Im Kernel Space wird nur die notwendigste Funktionalität implementiert
  - Geräte Treiber werden im User Space ausgeführt
  - Andere OS Funktionalität wird durch eigene Prozesse („Server“) im User Space abgebildet
  - Anwendungsprogramme nutzen die Funktionalität der Server
  - Kommunikation meist mit IPC/Message Passing
- Bekanntester Vertreter: Mach
  - Forschungsprojekt
  - Probleme mit Performance aufgrund des IPC Overheads zwischen Servern und OS
  - Mitarbeiter aus dem Projekt arbeiteten später bei Next/Apple und Microsoft

Hybrid

- Mischung aus Monolith und Microkernel Architektur
  - Teile der OS Komponenten werden in den User Space ausgelagert
  - Teilweise werden Gerätetreiber im User Space unterstützt
  - Großteil der Komponenten ist im Kernel Space implementiert
  - Damit werden mögliche Performance Probleme von Microkernel Architekturen abgeschwächt/vermieden
- Bekannte Vertreter
  - Windows >= NT
  - macOS

Windows Architektur



macOS Architektur

- Mit der Rückkehr von Steve Jobs zu Apple und dem Kauf von NeXT Computer wurden viele Technologien und Konzepte für macOS übernommen
  - NeXTSTEP
  - Objective-C
  - OO Anwendungs API (Kits)
  - Gerätetreiber Framework
- macOS basiert auf Mach
  - Implementiert allerdings viele Funktionalitäten im Kernel Space
  - Leiht Network Stack, Sockets, IPC, Filesysteme, div. Bibliotheken von (Free|Net|Open)BSD
  - Stellt mit Driver Kit ein Framework zur Treiberentwicklung im User Space zur Verfügung
- Das Kern OS wird als Darwin bezeichnet
  - Mach + *BSD + I/O Kit + Dateisysteme
- MacOS ist Darwin mit
  - Grafischen Benutzeroberfläche (Aqua)
  - Diverse Bibliotheken/Kits/Services (z.B. Quarz, Metal, Core Audio, Core Image, Foundation, Cocoa,...)

macOS Architektur