FSST

Betriebssysteme

Für die 4. und 5. Klasse

Version 0.5
Dezember 2023

Basierend auf einem Skript der Hochschule München - FG Technische Informatik von R. Thomas
Erweitert von Markus Signitzer

Version 0.5
Dezember 2023

HTL Anichstraße

Inhaltsverzeichnis

1 Versions-Log

1.1 Version 02

Mehr Info zu File-Systemen siehe Kapitel 2.5.8 und Kapitel 2.5.9 auf Seite 26 und Seite 27.

1.2 Version 03

Tip und Rechtschreibfehler beseitigt und mehr Linux-Befehle zu den einzelnen Kapiteln eingefügt.

1.3 Version 04

Legacy Python2 Code Beispiele auf Python3 umgeschrieben.
Mehr Beispiele für memory allocation in C.

1.4 Version 05

Kurze Zusammenfassung zu Virtualisierung sowie ein Kapitel zu Container und Docker hinzugefügt.

2 Betriebssysteme

2.1 Einführung

2.1.1 Wesen und Aufgaben eines Betriebssystems

Es gibt mehrere Definitionen eines Betriebssystems, nach DIN 44300 oder nach Aussagen von verschiedenen IT-Koryphäen. Aus Sicht eines Users deckt die folgende Definition es wohl am besten ab:

> Ein **Betriebssystem** bildet ein **Interface** zwischen dem **Benutzer** und der **Hardware** eines Rechensystems, das **die Komplexität der Hardware** vor dem Benutzer **verbirgt**.
> Es stellt die **Dienste der Hardware** in einer abstrakten Form – als Dienste einer virtuellen Maschine – zur Verfügung.
> Der Benutzer wird dadurch in die Lage versetzt, ein Rechensystem **ohne genaue Hardwarekenntnisse sinvoll und flexibel zu nutzen**.

2.1.1.1 Hauptaufgaben eines Betriebssystems:

- **Abstraktion**: Verbergen der Komplexität der Hardware vor dem Benutzer
- Bereitstellen einer Benutzerschnittstelle (Shell, Kommandointerpreter, Graphical User Interface (GUI))
- Bereitstellen einer Programmierschnittstelle (BS-Funktionen, Application Programming Interface, API), Programmier muss die Hardware nicht genau kennen!
- Prozess(or)verwaltung, Ressourcenaufteilung
- Arbeitsspeicherverwaltung
- externe Datenverwaltung (Verwaltung der Hintergrundspeicher, wie Platte, Band usw)
- Geräteverwaltung (Verwaltung der I/O-Geräte, wie Terminal, Drucker, Scanner, Modems usw)
- Benutzerverwaltung
- Erweiterten Features (in modernen Desktops): Bereitstellen von Dienstprogrammen (Editor, Übersetzer usw)

2.1.2 Klassifizierung von Betriebssystems

Wie kann man **Betriebssystem** nun **einteilen**? Es gibt mehrere **Klassifizierungsmöglichkeiten**: z.B. nach der Anzahl der gleichzeitig laufenden Programme oder nach der Anzahl gleichzeitig am Rechner arbeitender Benutzer. Für diesen Kurs reicht es aber sie nach dem Einsatzbereich zu Klassifizieren.

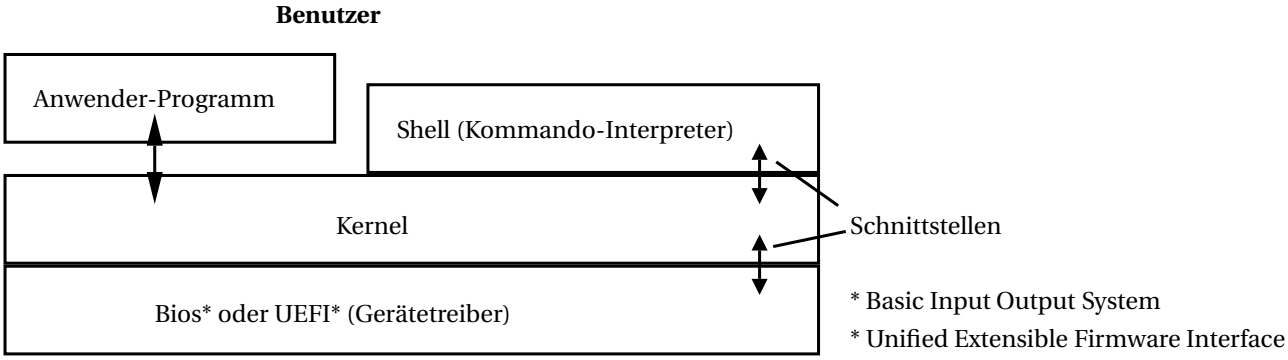
2.1.2.1 Klassifizierung nach dem Einsatzbereich - nicht vollständig!

- Mainframe- oder Super Computer Betriebssystem (mainframe - viele einfache Dinge gleichzeitig z.B. Flugbuchungen) - Super Computer - TerraFlops ohne Ende ;) - meist für Simulationen
- Server-Betriebssystem (server os), Web-,Mail-,FTP-,Datenbank-Server oder "Number Crunching"High Performance Computing - oft Cluster
- PC-Betriebssystem (personal computer os), Standardaufgaben, kennt jeder
- Betriebssystem für Industrie- und Medizinanwendungen meist Echtzeit-Betriebssysteme (Steuerungen, Regelungen, Herzschrittmacher)
- PDA-Betriebssystem (personal digital assistant os, handheld os, Android)
- Betriebssystem für eingebettete Systeme (embedded systems os) Videospiel-Konsolen, Fernseher, Geldautomaten, Router)
- Betriebssystem für Chipkarten (smart card os)

2.1.3 Architektur von Betriebssystems

2.1.3.1 Schichtenmodell:

Zur **logischen Strukturierung** ist ein Betriebssystem heute üblicherweise in **hierarchische Schichten** (bzw Schalen) unterteilt. Die **Anzahl der Schichten** (Schalen) hängt vom jeweiligen Betriebssystem ab. Sie beträgt aber i.a. **wenigstens drei BIOS, Kernel, Shell**



Jede Schicht bildet eine abstrakte (virtuelle) Maschine, die (bei konsequenter Realisierung des Schichtenmodells) nur mit ihren beiden direkt benachbarten Schichten über wohldefinierte Schnittstellen kommunizieren kann.
Sie kann Funktionen der nächstniedrigeren Schicht aufrufen und stellt ihrerseits ihre Funktionen der nächsthöheren Schicht zur Verfügung. Die **Gesamtheit** der von einer Schicht an ihrer oberen Schnittstelle angebotenen Funktionen werden auch als **Dienste** dieser Schicht bezeichnet.
Die Gesamtheit der **Vorschriften** (syntaktische, logische und physiche Festlegungen), die bei der Nutzung der Dienste einer Schicht einzuhalten sind, nennt man das **Protokoll** der Schnittstelle.

- Die **unterste Schicht** setzt direkt auf der Rechner-Hardware auf. Sie beinhaltet alle hardwareabhängigen Teile des Betriebssystems. Häufig wird diese Schicht BIOS (Basic Input Output System) genannt. Die Funktionalität des BIOS (Gerätetreiber) kann aber auch auf mehrere Schichten aufgeteilt sein. Die unterste (hardwarenächste) Schicht wird häufig als HAL – Hardware Abstraction Layer bezeichnet. Alle weiteren Schichten sind hardwareunabhängig.
- Der Kernel enthält die Betriebssystem-Funktionen höherer Ebene. Diese stehen über die Programmierschnittstelle (API – Application Programming Interface) den Anwendungsprogrammen (und der Shell) zur Verfügung. Auch hier ist eine Aufteilung auf mehrere Schichten möglich.
- Die oberste Schicht (Shell, Benutzeroberfläche, Kommandointerpreter) kommuniziert mit dem Benutzer. Sie stellt die Benutzerschnittstelle (Anwenderschnittstelle) zur Verfügung.

2.1.3.2 Kernel-Klassifizierung:

- Ein **Mikrokernel** ist ein Betriebssystemkern der **nur grundlegende Funktionen erfüllt** – in der Regel sind dies Speicher- und Prozessverwaltung, sowie Grundfunktionen zur Synchronisation und Kommunikation. **Alle weiteren Funktionen** werden als eigene Prozesse oder als Programmbibliothek **im Benutzer-Modus** implementiert.
  Vorteile: Bestandteile des OS können beliebig ausgetauscht werden; klare Schnittstellen; Absturz einer einzelnen Komponente führt nicht gleich zu Kernel-Panik; Treiber im Benutzer-Modus;
  Nachteile: Geschwindigkeit - langsamer durch die vielen Nutzer-Prozesse; Synchronisation zwischen den vielen Prozessen schwierig; Hardwarezugriff für normale Prozesse untersagt - benötigt **Kernel-Modus**;
  Beispiel OS: Symbian OS (Nokia), Minix, AmigaOS (Commodore)

- Ein **monolithischer Kernel** hat nicht nur Funktionen zu Speicher- und Prozessverwaltung und zur Kommunikation zwischen den Prozessen, sondern auch Treiber für die Hardwarekomponenten und möglicherweise weitere Funktionen direkt eingebaut.
  Vorteile: Schneller als ein Micro-Kernel; Zuverlässiger weil wichtige Funktionen wie z.B. Speicherverwaltung nicht von Userprogrammen abhängt;
  Nachteile: Nicht so flexibel wie ein Micro-Kernel - bei Änderungen ganzen Kernel neu kompilieren;
  Beispiel OS: Unix V, MS-Dos, FreeBSD - wird so eigentlich nicht mehr verwendet - hybrid Kernel!

- Ein **Hybridkernel** ist ein Kompromiss zwischen einem Microkernel und einem monolithischen Kernel, bei dem aus Geschwindigkeitsgründen einige Teile von monolithischen Kerneln in den Kern integriert und deswegen kein reiner Mikrokernel mehr ist, aber noch nicht genügend Funktionen besitzt um als monolithischer Kernel zu gelten. Er nicht so fehleranfällig wie ein monolithischer Kernel, da zum Beispiel nicht alle Treiber im privilegierten Modus laufen und somit bei einem Absturz nicht das ganze System zum Absturz bringen können. Andererseits sind nicht so viele Kontextwechsel nebst Kommunikation nötig wie bei einem Mikrokernel, was die Geschwindigkeit des Kernels erhöht.
  Beispiel Linux: Bei Bedarf wird zusätzliche Funktionalität über **Kernel-Module** dynamisch nachgeladen.

2.1.3.3 User-Mode vs Kernel-Mode

Im **Kernel-Modus** hat ein Programm zugriff auf sämtliche Systemeressourcen! Jede CPU Instruction kann ausgeführt werden und auf jede Speicheradresse kann zugegriffen werden! Kernel-Modus ist meist nur für die low-level Funktionen des Betriebssystems reserviert.
- Randnotiz: Spectre und Meltdown basieren auf spekulativen (branch prediction) Code Ausführungen bei denen auf Speicherbereiche zugegriffen wird wie im Kernel-Mode, obwohl der ursprüngliche Prozess im User-Mode läuft.
Der **User-Mode** ist viel eingeschränkter. Er kann auf Hardware nur über die System API zugreifen. Normalerweise laufen alle Prozesse eines Users im Usermode.

2.1.3.4 Linux Commands:

depmod - handle dependency descriptions for loadable kernel modules.
insmod - install loadable kernel module.
lsmod - list loaded modules.
modinfo - display information about a kernel module.
modprobe - high level handling of loadable modules.
rmmod - unload loadable modules.

Vorteil von Kernel-Modulen: Sie können dynamisch dem Kernel hinzugefügt werden - ohne restart! Vergleich zu Windows-Treiber-Installation - häufig ein reboot notwendig.

2.2 Grundlegende Aufgaben eines Betriebssystems

2.2.1 Prozessverwaltung

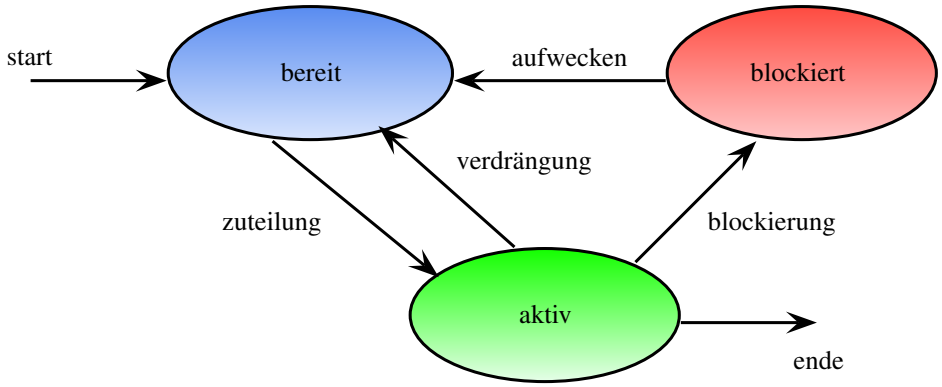
2.2.1.1 Prozesse

Definitionen: Unter einem **Prozess (Task)** versteht man den Ablauf eines **sequentiellen Programms** in einem Rechnersystem. Der Prozess läuft in einer **Prozessumgebung** (Ablaufumgebung), welche durch (Verwaltungs-)Informationen und zugeteilten Ressourcen definiert ist.
Multiprocessing (bei modernen Betriebsysteme) bezeichnet die Möglichkeit der „gleichzeitigen“ (mehrere Prozessorkerne) oder „quasi-gleichzeitigen“ (mehr Prozesse als Kerne) Ausführung von Prozessen in einem Betriebssystem - wird auch Nebenläufigkeit genannt. Der **Process-Scheduler** teilt den wartenden Prozessen CPU-Rechenzeit zu (siehe Prozessverwaltung unten).
Prozesse können während ihres Ablaufs neue (Child-) Prozesse erzeugen.
Nebenläufigkeiten innerhalb eines Prozesses werden **Threads** genannt. Jeder Prozess hat mindestens einen Thread (Main-Thread) welcher den Programmcode ausführt.
Unterschied Thread - Prozess: Threads teilen sich Ressourcen (z.B. Speicher) innerhalb ihrer Prozessumgebung. Prozesse können zwar kommunizieren aber haben keinen direkten Zugriff auf gegenseitige Ressourcen

2.2.1.2 Prozesszustände

Während seiner Abarbeitung kann ein Prozess unterschiedliche Zustände einnehmen. Die drei elementaren Zustände sind:

- **aktiv (running)**: Der Prozess belegt gerade CPU-Zeit (es wird gerechnet)
- **bereit (ready)**: Der Prozess ist lauffähig, er verfügt über alle benötigten Betriebsmittel (Prozessumgebung hergestellt) CPU-Zeit. Er kann jederzeit aktiv werden.
- **wartend/blockiert (waiting)**: Der Prozess wartet auf ein bestimmtes Ereignis (z.B: User Input), hat keine CPU-Zeit und wird erst wieder bereit wenn das Ereignis eingetreten ist.



2.2.1.3 Linux Commands:

top - simple process monitoring tool - standard in all distros
htop - better version of top - needs to be installed: sudo apt install htop
bpytop - even more flashy version of top; written in python; install via pip3

2.2.2 Prozess-Scheduler

Um die Steuerbefehle wie Zuteilung und Verdrängung kümmert sich der **Prozess-Scheduler** (nur in Multitasking-Systemen verhanden). Es gibt verschiedene Scheduling-Algorithmen um die zum Teil wiedersprüchlichen Ziele umzusetzen. Diese hängen zum Teil von der Betriebsart (Stapel, Dialog, Echtzeit) des Systems ab. Einige Ziele/Wünsche:

- Gerechtigkeit (Fairness) : Jeder Prozess erhält einen gerechten CPU-Anteil
- Effizienz : Optimale Auslastung der CPU und der übrigen Systemkomponenten (möglichst zu 100 Prozent) zu häufiger Prozesswechsel verbraucht zuviel Verwaltungs-CPU-Zeit und führt daher zur Ineffizienz

- Antwortzeit : Minimale Antwortzeit in interaktiven Systemen bzw Einhalten der Antwortzeit in Echtzeitsystemen
- Verweilzeit : Möglichst geringe Verweilzeit für Hintergrund-Prozesse
- Durchsatz : Abarbeitung möglichst vieler Aufträge pro Zeiteinheit
- Terminerfüllung : Bereitstellung bestimmter Ergebnisse zu festgelegten Terminen.

Der vom Scheduler realisierte Algorithmus sollte u.a. auch das Prozessverhalten bezüglich der Häufigkeit von I/O-Anfragen berücksichtigen. CPUs sind sehr schnell aber I/O kann (z.B. Festplattenzugriff) relativ lange dauern, das heißt das Scheduling von E/A-lastigen Prozessen wird immer wichtiger. Am besten wäre wenn einem rechenbereiten Prozess, der eine E/A durchführen möchte, sehr schnell die CPU zugeteilt würde. Nach kurzer CPU-Belegung wird der Prozess die E/A-Anforderung durchführen und die CPU wieder freigeben. Diese kann dann einem anderen Prozess zugeteilt werden. Dadurch werden sowohl die CPU als auch die Platte besser ausgelastet.
Problem : Wie soll der Scheduler erfahren, ob ein rechenbereiter Prozess E/A durchführen möchte? (geht nicht vorausschauend!)

2.2.2.1 Scheduling-Strategien

Grundsätzlich kann man in zwei Arten unterscheiden: - **kooperatives Multitasking (non preemptive)**: Der aktive Prozess gibt die CPU von sich aus frei (freiwillig oder weil er blockiert wird).Nur geringer Verwaltungsaufwand, Gefahr der Blockierung durch unkooperative Prozesse
- **verdrängendes Multitasking (preemptive)**: Dem aktiven Prozess wird die CPU durch den Scheduler entzogen (z.B. bei Ablauf der Zeitscheibe, Timer-Interrupt). Erhöhter Verwaltungsaufwand, keine Blockierung durch unkooperative (fehlerhafte) Prozesse. Wann kommt die Verdrängung? Entweder zeitgesteuert oder ereignisgesteuert.

- **Eingangsreihenfolge (first come first served, FCFS)**
  Zuteilung der CPU in der Start-Reihenfolge der Prozesse. Alle rechenbereiten Prozesse bilden eine einzige Warteschlange. Jeder neu gestartete Prozess wird an das Ende der Warteschlange gehängt. Bei einem Prozesswechsel wegen Beendigung oder Blockierung des aktiven Prozesses wird dem am Anfang der Warteschlange stehenden Prozess die CPU zugeteilt. Ein nach Blockierung wieder rechenbereit werdender Prozess kommt ebenfalls an das Ende der Warteschlange. Der jeweils aktive Prozess kann die CPU bis zu seiner Beendigung oder bis er blockiert wird belegen. Es findet keine Verdrängung des aktiven Prozesses statt.
  Gute Systemauslastung (geringe Verluste durch Prozesswechsel), schlechtes Antwortzeitverhalten (kurz laufende Prozesse können von langlaufenden stark verzögert werden), einfache Implementierung

- **Kürzester Auftrag zuerst (shortest job first, SJF)** Zuteilung der CPU nach der (bekannten oder geschätzten) Rechenzeit der Prozesse. Kürzer laufende Prozesse werden vor länger laufenden ausgeführt. Kürzeste mittlere Verweilzeit, gut für Batch-Systeme geeignet.
  Modifikation : **Shortest Remaining Time Next (SRTN)**: Auch für preemptives Multitasking geeignet. Bei jeder Scheduling-Entscheidung wählt der Scheduler den Prozess aus, dessen verbleibende Rechenzeit am kürzesten ist. Hier muß die Rechenzeit eines Prozesses im voraus bekannt sein.

- **Zeitscheibenverfahren (round robin, RR)**
  Die CPU wird zyklisch nacheinander allen bereiten Prozessen für einen bestimmten Zeitabschnitt (Zeitscheibe, time slice) zugeordnet (in der Reihenfolge der Warteschlange). Nach Ablauf seiner Zeitscheibe wird der aktive Prozess verdrängt und an das Ende der Warteschlange gehängt, alle Prozesse haben die gleiche Priorität. Gleichmäßige Aufteilung der Betriebsmittel auf alle Prozesse, kurze Antwortzeiten bei kurzen Zeitscheiben, aber erhöhte Verluste durch Prozesswechsel - Kompromiss wie lange ist ein time slice?

- **Prioritätssteuerung**
  Jedem bereiten Prozess wird eine Priorität zugeordnet. Vergabe der CPU in absteigender Priorität. Ein Prozess niedrigerer Priorität kann die CPU erst erhalten, wenn alle Prozesse höherer Priorität abgearbeitet sind. Ein bereit werdender Prozess höherer Priorität verdrängt einen aktiven Prozess niedrigerer Priorität. Alle Prozesse gleicher Priorität werden häufig in jeweils einer eigenen Warteschlange geführt (Prioritätsklassen). Realisierung mehrerer unterschiedlicher Verfahren, z. Tl miteinander u/o mit anderen Strategien kombiniert, z.B.:
  - Reine Prioritätssteuerung: Prozesse gleicher Priorität werden nach dem FCFS-Prinzip abgearbeitet (z.B. in Echtzeit-BS)
  - Prioritätssteuerung mit unterlagertem Zeitscheibenverfahren: Prozesse gleicher Priorität werden nach dem Zeitscheibenverfahren abgearbeitet - Dynamische Prioritätsänderung: Allmähliche Erhöhung der Priorität der auf die CPU wartenden Prozesse bzw Erniedrigung der Priorität des gerade ausgeführten Prozesses nach jedem Timer-Tick

- Mehrstufiges Herabsetzen (multilevel feedback, MLF): Festlegung einer maximalen Rechenzeit für jede Prioritätsstufe. Hat ein Prozess die max. Rechenzeit seiner Priorität verbraucht, bekommt er die nächstniedrigere Priorität.

- **Lotterie-Scheduling**
  Bei jeder Scheduling-Entscheidung wird der als nächstes laufende Prozess zufällig ausgewählt. Eine Prozess-Priorität kann dadurch berücksichtigt werden, dass höherpriore Prozesse eine ihrer Priorität entsprechende höhere Auswahl-Chance bekommen.

```
Windows-Info:
https://docs.microsoft.com/en-us/windows/desktop/procthread/processes-and-threads

Linux-Info: Seit Kernel 2.6.23: Completely Fair Scheduler:
https://de.wikipedia.org/wiki/Completely_Fair_Scheduler
```

2.2.3 Prozess-Synchronisation

Kooperierende nebenläufige Prozesse müssen wegen der zwischen ihnen vorhandenen Abhängigkeiten miteinander synchronisiert (koordiniert) werden. Prinzipiell lassen sich zwei Klassen von Abhängigkeiten unterscheiden:
- Die Prozesse konkurieren um die Nutzung gemeinsamer (exklusiv nutzbarer) Betriebsmittel (race condition!). Beispiel : Zwei Prozesse greifen verändernd zu gemeinsamen Daten zu. Der Zugriff zu den gemeinsamen Daten muß koordiniert werden, um eine Inkonsistenz der Daten zu vermeiden. Dies wird auch äußere Abhänigkeit genannt.
- Die Prozesse sind voneinander datenabhängig. Beispiel: Ein Prozess erzeugt Daten, die von einem anderen Prozess weiter bearbeitet werden sollen. Es muß eine bestimmte Abarbeitungsreihenfolge entsprechender Verarbeitungsschritte eingehalten werden. Zustands- oder Ereignissynchronisation (z.B. Produzenten–Konsumenten–Synchronisation). Dies wird auch innere Abhängigkeit genannt.

2.2.3.1 Semaphore und Locks

Fur die Synchronisation von Prozessen werden **Locks** (kennen wir aus den Programmiersprachen - siehe Deadlocks weiter unten) und Semaphore eingesetzt.
Ein **Semaphor** (Sperrvariable, Steuervariable) signalisiert einen Zustand (Belegungszustand, Eintritt eines Ereignisses) und gibt in Abhängigkeit von diesem Zustand den weiteren Prozessablauf frei oder versetzt den betreffenden Prozess in den Wartezustand.
Semaphore fur den **gegenseitigen Ausschluss (mutual exclusion oder MUTEX)** sind dem jeweiligen exklusiv nutzbaren Betriebsmittel zugeordnet und verwalten eine Warteliste fur dieses Betriebsmittel. Sie sind allen Prozessen zugänglich.
Semaphore fur die **Ereignissynchronisation** zweier voneinander datenabhängiger Prozesse sind diesen Prozessen direkt zugeordnet. Sie dienen zur Übergabe einer Meldung über das Ereignis zwischen den Prozessen.
Für ein Python-Beispiel zum Thema locks siehe Kaptiel 2.3.5 auf Seite 16.

2.2.4 Kommunikation zwischen Prozessen - IPC

In Multitasking-System werden Mechanismen für den Informationsaustausch zwischen Prozessen bereitgestellt - interprocess communication, IPC
U.a. beruht auch die Prozess-Synchronisation auf einer Kommunikation der beteiligten Prozesse (z.Tl. indirekt über das Betriebssystem)
Mögliche Interprozesskommunikation:

- Kommunikation über **gemeinsame Speicherbereiche**
  Prozesse können dort bestimmte Variable, Pufferbereiche usw anlegen und gemeinsam nutzen

- Kommunikation über **gemeinsame Dateien**
  Prozesse schreiben in Dateien, die von anderen Prozessen gelesen werden können

- Kommunikation über **Pipes**
  Eine Pipe (oder Pipeline) ist ein unidirektionaler Kommunikationskanal zwischen zwei Prozessen: Ein Prozess schreibt Daten in den Kanal (Anfügen am Ende),der andere Prozess liest die Daten in derselben Reihenfolge wieder aus (Entnahme am Anfang).
  Realisiert werden Pipes entweder im Arbeitsspeicher oder als spezielle Dateien. Von den beteiligten Prozessen werden sie i.a. wie Dateien behandelt. Die Lebensdauer einer Pipe ist i.a. an die Lebensdauer der beteiligten Prozesse gebunden.

- Kommunikation über **Signale**
  Signale sind asynchron auftretende Ereignisse, die eine Unterbrechung bewirken Typischerweise werden sie zur Kommunikation zwischen Betriebssystem und Prozess eingesetzt
- Kommunikation über **Nachrichten** (message passing)
  Nachrichten werden vom BS verwaltet. Es sind zahlreiche Ausprägungen und Varianten dieses Mechanismus implementiert.
- Kommunikation über **Sockets**
  Sockets ermöglichen eine Kommunikation über Rechnernetze - kennen wir vom Programmieren!
- Kommunikation über **Prozedurfernaufrufe** (remote procedure call, RPC)
  Ein Prozess ruft eine in einem anderen Prozess angesiedelte Prozedur – also über seine Adressgrenzen hinweg – auf. Besonders für Client-Server-Beziehungen geeignet. Ist z.B. die Basis für JAVA-RMI.

2.2.4.1 Linux Commands:

Es gibt ein paar nützliche Tricks um Prozesse in den Hintergrund zu schieben. Beispiel: SSH Verbindung auf Server, man editiert gerade ein File via nano und möchte kurz in einem anderen Verzeichnis was nachsehen. Statt nano zu beenden und später wieder zu starten, kann man den Prozess einfach in den Hintergrund legen (nano verschwindet und man ist auf der CMD)

ctrl - c: beendet den laufenden Prozess
ctrl - z: schiebt den laufenden Prozess in den Hintergrund - es wird eine job id (n) ausgegeben, der Prozess ist blockiert!
fg %n: den Prozess mit der Job-Nummer n wieder in den Vordergrund bringen und aktivieren
bg %n: den Prozess mit der Job-Nummer n im Hintergrund lassen aber aktivieren! Achtung falls der Prozess z.B. auf stdout schreibt oder user input braucht.
jobs -l: Liste aller Jobs anzeigen
command &: Ein Command/Prozess gleich im Hintergund ausführen

2.2.5 Deadlocks - Verklemmung von Prozessen

Unter einem **Deadlock** (Verklemmung) versteht man die **gegenseitige Blockade** von wenigsten zwei Prozessen. Jeder der beteiligten Prozesse wartet auf ein Ereignis (Zustandsänderung, Bedingung), das nur ein anderer der beteiligten Prozesse auslösen kann und das – da dieser Prozess auch wartet – nie eintreten kann.

2.2.5.1 Typische Ursache für ein Deadlock:

Zwei Prozesse warten jeweils auf die **Zuteilung** eines nur **exklusiv nutzbaren Betriebsmittels**, das aber gerade von dem **jeweils anderen Prozess belegt ist** - beide Prozesse bleiben ständig im Zustand "**wartend**".
Beispiel:
Prozess A sammelt Daten, aktualisiert mit diesen Einträge in einer Datei und protokolliert das auf dem Drucker
Prozess B gibt die gesamte Datei auf dem Drucker aus.
Beide Betriebsmittel (Datei und Drucker) sind jeweils mit gegenseitigem Ausschluss (Semaphor oder Lock) geschützt.
Prozess A belegt die Datei und aktualisiert einen Eintrag
Prozess B belegt den Drucker und gibt eine Überschrift aus
Prozess B fordert die Datei zum Ausdrucken an - da Datei von A belegt ist, wartet B
Prozess A fordert den Drucker zur Protokollierung an - da Drucker von B belegt ist, wartet A
- gegenseitige Blockierung !

2.2.5.2 Bedingungen für das Auftreten von Deadlocks:

- **Exklusive Nutzung**: Die angeforderten Betriebsmittel sind nur exklusiv nutzbar (gegenseitiger Ausschluss).
- **Wartebedingung**: Alle beteiligten Prozesse warten auf die Zuteilung von Betriebsmitteln, die von den anderen Prozessen bereits belegt sind, ohne die von ihnen selbst belegten Betriebsmittel freizugeben.
- **Nichtentziehbarkeit**: Die von einem Prozess belegten Betriebsmittel können ihm nicht zwangsweise entzogen werden.
- **Zyklisches Warten**: n ($n \geq 2$) Prozesse warten in einer geschlossenen Kette auf ein Betriebsmittel, das durch den jeweils nächsten Prozess in der Kette gehalten wird.

2.2.5.3 Stragegien zu Behandlung von Deadlocks:

- **Ignorieren (ignore)** ("Vogel-Strauß-Algorithmus"):
  Dieser Strategie liegt die Annahme zu Grunde, dass Verklemmungen relativ selten sind und sich daher der Aufwand für andere Behandlungsstrategien nicht lohnt. Zumindest in Echtzeitsystemen (Prozesssteuerungen) nicht tragbar.

- **Erkennen und Beseitigen** (detect and recover):
  Deadlocks werden prinzipiell zugelassen, das System versucht, diese zu erkennen und dann etwas dagegen zu tun. Es gibt mehrere Algorithmen, mit denen das Vorliegen von Verklemmungen festgestellt und die daran beteiligten Prozesse identifiziert werden können. Beseitigung einer erkannten Verklemmung z.B. durch Zurückversetzen eines Prozesses, der ein benötigtes Betriebsmittel besitzt, in einen früheren Zustand (Rollback), in dem er dieses Betriebsmittel noch nicht belegt hat, und anschließende Zuteilung dieses Betriebsmittels an einen anderen Prozess. Eine andere – einfachere aber härtere – Möglichkeit hierfür: Gewaltsamen Beendigung eines oder mehrerer der beteiligten Prozesse.
  Einfachere Methode : Zyklische Überprüfung auf Prozesse, die bereits eine längere Zeit blockiert sind und gewaltsame Beendigung derartiger Prozesse. Nachteil: Daten können verloren gehen

- **Vermeiden** (avoid):
  Bei dieser Strategie wird dafür gesorgt, daß nie alle für eine Verklemmung notwendigen Bedingungen erfüllt sind. Beispiele:
  - Vorsorge, daß ein exklusives Betriebsmittel überhaupt nur von einem einzigen festgelegten Prozess benutzt werden kann (z.B. Druckerspooler)
  - Verhinderung der Wartebedingung dadurch, daß jeder Prozess bei der Neuanforderung eines Betriebsmittels zunächst alle bisher belegten Betriebsmittel freigeben muss. Nur bei erfolgreicher Neuanforderung bekommt er auch alle bisherigen Betriebsmittel wieder zurück.

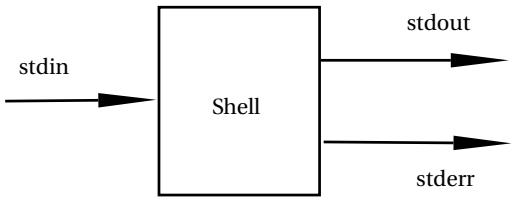
- **Verhindern** (prevent) durch sorgfältige Betriebsmittelvergabe und darauf abgestimmtes Scheduling:
  Das BS darf ein Betriebsmittel nur zuteilen, wenn es *ungefährlich* ist. Hierfür muß ihm der maximale Betriebsmittelbedarf aller Prozesse bekannt sein. Das BS kann dann bei jeder Betriebsmittelanforderung überprüfen, ob der Restvorrat freier Betriebsmittel auch noch nach erfolgter Zuteilung groß genug ist, um bei wenigstens einem Prozess den Maximalbedarf befriedigen zu können. Nur wenn das der Fall ist, erfolgt auch tatsächlich die Zuteilung, sonst wird sie verweigert.

2.3 Python, C und Betriebssysteminteraktion

Python interagiert mit dem Betriebssystem primär über die zwei modules *sys* und *os*.

2.3.1 Standard Data Streams in Linux

Folgendes Beispiel erlaubt das Einlesen und Ausgeben von der Shell: Soweit nichts Neues, nur mit Exception-Handling. Warum? Wird klar wenn wir nicht selbst die Daten eingeben sonder zum Beispiel den stdin für unser python Skript so umleiten, dass der Input aus einem File kommt.




```
import sys

while True:
    #ausgabe zu stdout
    print("yet another iteration ...")
    try:
        number = input("Zahl eingeben oder Ctrl-D um abzubrechen ->")
    except EOFError:
        print("\nciao")
        break
    else:
        number = int(number)
        if number == 0:
            print("0 has no inverse", file = sys.stderr)
        else:
            print("inverse of {} is {:.3f}".format(number, 1.0/number))
```

Das Programm einfach einmal normal, dann mit umgeleiteten Input und auch mit umgeleiteten stdout ausprobieren:

```
$ python streams.py < numbers.txt      #eingabe der zahlen aus dem txt file
$ python streams.py < numbers.txt > output.txt  #nun auch stdout umgeleitet
```

Im Vergleich dazu das ganze nochmals in C:

```
#include <stdio.h>

int main()
{
    int number;
    for(;;){
        printf("Enter a number or Ctrl-D to exit\n");
        int result = scanf("%d",&number);
        if (result == EOF){
            printf("\n ciao \n");
            break;
        }else{
            if (number == 0){
                fprintf(stderr,"0 geht nicht!\n");
            }else{
                printf("Zahl %d hat den Kehrwert %f\n",number,1.0/number);
            }
        }
    }
    return 0;
}
```

2.3.2 Prozesse und Forks

Begriff **Fork**: Ein vom Betriebssystem bereitgestellter Systemaufruf, der den bestehenden Prozess aufspaltet - und dabei eine exakte Kopie des Prozesses erzeugt - und dann beide Prozesse gewissermaßen parallel laufen lässt. Mit dem *if pid == 0* kann man im child prozess eventuell anderen code ausführen als im parent-process.
Beispiel Skript:
Der Loop läuft nur zweimal - wie viele Prozesse werden erzeugt?

```
import os
import time

for i in range(2):
    print("I'm PID={} and about to be a mum!".format(os.getpid()))
    time.sleep(5)
    pid = os.fork()
    if pid == 0:
        print("I'm PID={}, a newborn that".format(os.getpid()),
            "knows to write to the terminal!")
    else:
        print("I'm PID={} the mum of {},".format(os.getpid(),pid),
            "and it knows to use the terminal!")
        os.waitpid(pid,0)
```

Und nochmals in C:

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

int main(void)
{
    int dummy;
    for(int i = 0; i < 3; i++){
        printf("Ich PID=%d bin der Parent-Process\n",getpid());
        //scanf eingabe nur zum durchsteppen
        scanf("%d",&dummy);
        int pid = fork();
        if (pid == 0){
            //bin im child process
            printf("Ich bin NEU und meine PID=%d\n",getpid());
            printf("Mein Parent hat die PID=%d\n",getppid());
        }else{
            //bin im parent process
            printf("Ich PID=%d bin der Parent-Process\n",getpid());
        }
    }
    return 0;
}
```

2.3.2.1 Externe Programme mit exec*() aufrufen

Bis jetzt haben wir mit `fork()` nur Code ausgeführt, welcher schon in unseren Skripten definiert war. Es ist aber auch möglich externe Programme aufzurufen - dies funktioniert mit den exec*() Funktionen (sind sowohl in Python als auch C gleich).
Hier ein Beispiel C-Programm, welches forked und im Child-Process die Linux Funktion `ls -la /` aufruft (also den Inhalt des Root-Directories auflistet):
Achtung: Wir verwenden auch die `wait()` Funktion, um einen Prozess auf den anderen warten zu lassen. In diesem Beispiel wartet der Parent-Process auf das Ende des Child-Process bevor er mit seiner Code Ausführung fortfährt.

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


int main() {
    char *arg_list[] = {"ls", "-l", "/", NULL};
    pid_t child_pid = fork();
    if (child_pid != 0){
        printf("This is the Parent ... will wait\n");
        wait(NULL);
        printf ("Main program exiting...\n");
        return 0;
    }else{
        printf("This is the Child-Process!\n");
        // execute external program
        execvp("ls", arg_list);
        fprintf(stderr, "An error occurred in execvp\n");
        abort();
    }
}
```

Das Gleiche in Python3:

```
import os

arg_list = ["ls", "-l", "/"]
child_pid = os.fork()
if (child_pid != 0):
    print("This is the Parent ... will wait")
    os.wait()
    print("Main program exiting...")
else:
    print("This is the Child-Process!")
    #execute external program
    try:
        os.execvp("ls", arg_list);
    except OSError:
        print("An Error occured in execvp")
```

Mehr Informationen zu den verschiedenen `exec*` Funktionen und ihren Argumenten findet man unter folgenden links:
Für Python3: https://docs.python.org/3/library/os.html#process-management
Für C: http://openbook.rheinwerk-verlag.de/linux_unix_programmierung/Kap07-009.htm#RxxKap07009040002001F028100

2.3.3 Python Multi-Threading vs Multi-Processing

Für **Python** gilt:

Die Aufgabe von **Multi-Threading** besteht darin, die Reaktion von Anwendungen zu ermöglichen z.B. User-Interaktion mit einer GUI während im Hintergrund auf eine Datenbank zugegriffen wird. Da in cPython das Global Interpreter Lock (GIL) läuft, hat man keine echte Nebenläufigkeit. Vorteile: gleicher Adressbereich, einfacher Datenaustausch; Nachteile: Wenn ein Thread abstürzt kann das ganze Programm abstürzen, Geschwindigkeit ist nicht hoch.
Multi-Processing ist im Vergleich dazu viel schneller, aber die Kommunikation zwischen den Prozessen ist aufwendiger, weil sie in isolierten Speicherbereichen laufen. Dies ist aber wieder ein Vorteil für die Absturz-Sicherheit.
Fazit: Multi-Processing für echtes Rechnen auf CPU-Cores und Multi-Threading für responsive Apps!
p.s. um herauszufinden welcher Python-Interpreter installiert ist genügen zwei Zeilen:
import platform
print(platform.python_implementation())

Beispiel für **Multi-Threading:**

```
import threading
import time

def cpuBurn(threadNumber):
    for i in range(10000):
        for j in range(10000):
            i*j
    print("Finished: Thread_",threadNumber)


if __name__ == "__main__":
    startTime = time.time()
    myThreads = []
    for i in range(4):
        t = threading.Thread(target=
            cpuBurn, args=(i,))
        myThreads.append(t)
    for x in myThreads:
        x.start()
    for x in myThreads:
        x.join()
    endTime = time.time()
    print("runtime = ",endTime -
        startTime,"sec.")
```

Beispiel für **Multi-Processing:**

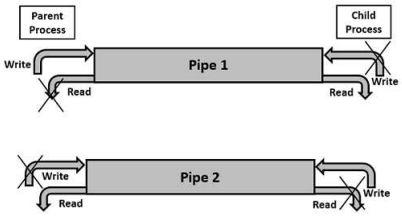
```
import multiprocessing
import time

def cpuBurn(threadNumber):
    for i in range(10000):
        for j in range(10000):
            i*j
    print("Finished: Thread_",threadNumber)


if __name__ == "__main__":
    startTime = time.time()
    myProcesses = []
    for i in range(4):
        p = multiprocessing.Process(target=
            cpuBurn, args=(i,))
        myProcesses.append(p)
    for x in myProcesses:
        x.start()
    for x in myProcesses:
        x.join()
    endTime = time.time()
    print("runtime = ",endTime -
        startTime,"sec.")
```

In C ist das **Multi-Threading** schon schnell und nützt mehrere CPU Kerne voll aus. Ein Beispielcode bei dem die Zeitnehmung allerdings nur für UNIX in dieser Form funktioniert.
ACHTUNG: Beim Compelieren die Option `-l pthread` mit angeben. Es muss dem Linker manuell gesagt werden die pthread Library einzubinden (warum ist mir ???)

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>
#include <sys/time.h>

//this function is the workload for our threads
//it is in itself quite useless :)
void *cpuBurn(void *id)
{
    int x;
    for(int i = 0; i<200000; i++){
        for(int j = 0; j<10000; j++){
            x=j*i;
        }
    }
    long myid = (long)id;
    printf("Finished Thread: %ld \n", myid);
    return NULL;
}

int main()
{
    //variables needed for time measurment
    long start, end;
    struct timeval timecheck;
    //get the start time (only on unix)
    gettimeofday(&timecheck, NULL);
    start = (long)timecheck.tv_sec * 1000 + (long)timecheck.tv_usec / 1000;

    pthread_t threads[4];
    // Let us create four threads
    for (long i = 0; i < 4; i++){
        pthread_create(&threads[i], NULL, cpuBurn, (void *)i);
    }
    // join: the main thread will wait for the threads to finish
    for (long i = 0; i < 4; i++){
        pthread_join(threads[i], NULL );
    }

    //get the time again
    gettimeofday(&timecheck, NULL);
    end = (long)timecheck.tv_sec * 1000 + (long)timecheck.tv_usec / 1000;

    printf("Runtime was %ld milliseconds\n", (end - start));
    return 0;
}
```

2.3.4 Inter-Process-Communication (IPC) mittels Pipes

Als **Pipe** bezeichnet man einen gepufferten uni- oder bidirektionalen Datenstrom zwischen zwei Prozessen nach dem FIFO (First In - First Out)- Prinzip. Das bedeutet, dass die Ausgabe eines Prozesses als Eingabe für einen weiteren verwendet wird. Pipes wurden 1973 in Unix eingeführt.
Generell unterscheidet man zwei Sorten von Pipes: anonyme (anonymous pipes) und benamte (named pipes)
Anonyme Pipes existieren nur innerhalb von Prozessen und werden typischerweise in Verbindung mit forks benutzt.
Achtung: Wenn man eine pipe erstellt bekommt man zwei file descriptors - einen für lesen und einen für schreiben. Da beim `fork()`aber alle Variablen des Parent-Prozesses im Child-Prozess geklont werden, haben beide Prozesse die Schreib-/Lese-Descriptoren. Da pipes unidirectional sind, muss darauf geachtet werden nur in eine Richtung zu schreiben! Wenn man bidirektionale Kommunikation möchte muss man mit zwei pipes arbeiten.



Beispiel mit einer Pipe für unidirektionale Kommunikation:

```
import os, sys

print("The child will write text to a pipe and ")
print( "the parent will read the text written by child...")

# bi-directional pipe gives us two
# file descriptors r, w for reading and writing
r, w = os.pipe()

processid = os.fork()
if processid:
   # This is the parent process
   # Closes file descriptor w, will only read
   os.close(w)
   r = os.fdopen(r)
   print("Parent reading")
   # will wait until it receives something
   str = r.read()
   print("text =", str)
   sys.exit(0)
else:
   # This is the child process
   # close file descriptor r, will only write
   os.close(r)
   w = os.fdopen(w, 'w')
   print("Child writing")
   w.write("Text written by child...")
   w.close()
   print("Child closing")
   sys.exit(0)
```

Das Programm ergibt folgenden output:

```
The child will write text to a pipe and
the parent will read the text written by child...
Parent reading
Child writing
Child closing
text = Text written by child...
```

Nun das Gleiche nochmals in C:

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

int main()
{
        int fd[2]; // Used to store the two ends of the pipe
        //create the pipe
        pipe(fd);

        pid_t p;
        p = fork();

        // Parent process
        if (p == 0){
                char str[100];
                close(fd[1]); // Close writing end of the pipe
                printf("Parent reading/waiting ...\n");
                // Wait for child to send a string
                wait(NULL);
                // Read string from child, print it and close
                // reading end.
                read(fd[0], str, 100);
                printf("Child send: %s\n", str);
                close(fd[0]);
        }else{
                // child process
                // Close reading end
                close(fd[0]);
                char message[100];
                printf("Child: Enter message to parent:\n");
                fgets(message, 100, stdin);
                // Write message to parent and close writing end
                write(fd[1], message, strlen(message)+1);
                close(fd[1]);
                exit(0);
        }
        return 0;
}
```

Ergibt folgenden Input/Output:

```
Parent reading/waiting ...
Child: Enter message to parent:
Hello my dear Parent!
Child send: Hello my dear Parent!
```

Unter Linux ebenso wie unter Unix ist es möglich Pipes anzulegen, die als Dateien eingerichtet sind. Diese Pipes werden als **benamte Pipes** (englisch: named Pipes) oder manchmal auch als Fifos (First In First Out) bezeichnet.
Ein Prozess liest und schreibt von einer solchen Pipe wie bei einer normalen Datei. Oft schreiben mehrere Prozesse in eine Pipe und nur ein Prozess liest daraus z.B. bei Zugriff auf geteilte Ressourcen wie den Drucker-Spooler.

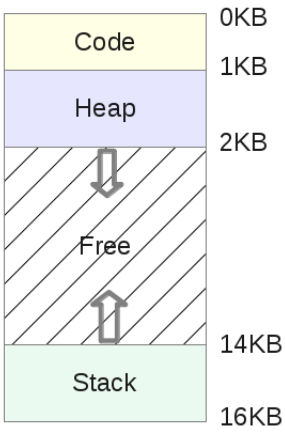
2.3.5 Mit Locks Race-Conditions vermeiden

Im folgenden Beispiel wird ein Lock verwendet, um Race-Conditions zu vermeiden. Der Code ist recht einfach: Zehn Threads sollen über eine eigene Methode eine shared integer Variable je tausend mal um eins erhöhen. Wenn jeder Thread nacheinander auf die Variable zugreifen würde, sollte sich nach Programmablauf den Wert 10000 haben. Ohne Lock hat die Variable aber ganz willkürliche Werte, weil es zu Race-Conditions kommt.
Bitte den folgenden Code einmal mit den `lock.acquire()` und `lock.release()` Funktionen ausführen und einmal ohne.

```
import threading
import time

# Shared variable among threads
shared_variable = 0
# Number of threads to be created
num_threads = 10
# Number of increments per thread
increments_per_thread = 1000
#create a lock object
lock = threading.Lock()

def work_for_threads():
    for _ in range(increments_per_thread):
        #acquire the lock befor calling the function
        lock.acquire()
        increment_shared_variable()
        #release the lock after the function call
        lock.release()

# Function to increment the shared variable
def increment_shared_variable():
    global shared_variable
    x = shared_variable
    time.sleep(0.0001)
    x = x + 1
    shared_variable = x

if __name__ == "__main__":
    threads = []

    for _ in range(num_threads):
        thread = threading.Thread(target=work_for_threads)
        threads.append(thread)
        thread.start()

    # Wait for all threads to complete
    for thread in threads:
        thread.join()

    print(f"Expected value: {num_threads * increments_per_thread}")
    print(f"Actual value: {shared_variable}")
```

2.4 Speicherverwaltung

Der Arbeitsspeicher ist ein Betriebsmittel, das von jedem Prozess benötigt wird. Die Arbeitsspeicherverwaltung ist daher eine zentrale Aufgabe jedes Betriebssystems.
Moderen Betriebssysteme verwenden für laufende Prozesse meist folgendes **code-stack-heap memory layout**: Beispiel für einen laufenden Prozess dem 16kB Speicher zugewiesen wurden:



- **Code**: Hier liegen die CPU-Instruktionen des kompilierten Programms - wird vom compiler erzeugt und hat meist eine fixe Größe. In unserem Beispiel 1kB.
- **Heap**: Freier Speicherbereich eines Programms. Hier liegen z.B. erzeugte Variablen und Objekte.
- **Stack**: Hier liegen Funktionsaufrufe und die Variablen innerhalb der aufgerufenen Funktionen - Achtung: kann bei rekursiven Funktionsaufrufen sehr groß werden!
- **Freier Bereich** zwischen Stack und Heap wird während der Laufzeit des Programms dynamisch dem Stack und/oder Heap zugewiesen.

2.4.0.1 Aufgaben der Arbeitsspeicherverwaltung (Memory Manager)

- Allokation von Speicherbereichen für die Prozesse (für Programmcode und Daten, beim Laden und aufgrund späterer Anforderungen)
- Freigabe nicht mehr benötigten Speichers
- Führung einer Übersicht über freie und belegte Teile des Arbeitsspeichers
- Gegebenenfalls Durchführung einer Speicherbereinigung (garbage collection)
- Organisation der Auslagerung von Speicherbereichen auf einen Hintergrundspeicher, wenn nicht alle geladenen (laufenden) Prozesse vollständig in den Arbeitsspeicher passen
- Realisierung von Speicherschutzmaßnahmen

2.4.1 Der tatsächliche Umfang der Speicherverwaltung ...

... hängt oft von der Art des eingesetzten Speichersystems (Speicher ohne Auslagerung, Speicher mit Auslagerung wie Swapping oder Virtueller Speicher, Speicher mit Bank-Umschaltung) und der Klasse und dem Umfang des Betriebssystems (Singletasking-BS, Multitasking-BS, Echtzeit-BS usw) ab.
In **modernen Systemen** wird die Speicherverwaltung meist durch geeignete **Hardwarekomponenten** (Speicherverwaltungseinheit, **memory management unit, MMU**) **unterstützt**. In derartigen Systemen weist die Speicherverwaltung i.a. eine **Schichtenstruktur** auf:
- untere Schicht: Verwaltung des realen Arbeitsspeichers
- obere Schicht: Verwaltung eines virtuellen Speichers

2.4.2 Verwaltung des realen Arbeitsspeichers

- Aufteilung des Speichers in eine variable – sich ständig ändernde – Anzahl von Bereichen variabler Größe (siehe Segmentation Kapitel: 2.4.5.1 auf Seite 19) ODER Aufteilung des Arbeitsspeichers in eine feste Anzahl von Bereichen (Partitionen, partitions) fester Größe (die aber nicht notwendigerweise für alle Bereiche gleich sein muß) - siehe paging Kapitel: 2.4.5.2 auf Seite 20.
- Da dynamisch eine wechselnde Anzahl von Bereichen unterschiedlicher Größe allokiert und wieder freigegebenwird, wird im Laufe der Zeit der Speicher in belegte und freie Bereiche **fragmentiert** (Speicherzerstückelung). Dies kann dann dazu führen, daß eine Anforderung nach einem Speicher bestimmter Größe nicht erfüllt werden kann, weil der gewünschte freie Speicher nicht geschlossen zur Verfügung steht, obwohl insgesamt genügend Speicher frei ist.
  Abhilfe schafft eine **Speicherbereinigung** (Freispeichersammlung, garbage collection, bzw. Speicherverdichtung, compactification). Diese ist sowohl aufwendig als auch Zeit beanspruchend. Für die Durchführung einer

Speicherbereinigung existieren verschiedene Konzepte. Üblich ist u.a. ein im Hintergrund – mit niedriger Priorität – laufender Systemprozess (bei Freispeichersammlung) oder ein Warten bis erstmals eine Speicheranforderung nicht befriedigt werden kann (bei Speicherverdichtung).

- Die Speicherverwaltung benötigt eine **Übersicht** über den **Belegungszustand** (frei oder belegt) deszu verwaltenden Speichers. Üblich sind die folgenden Konzepte:
  - **Belegungs-Bitmap** (allocation bit map): Sie wird angewendet, wenn der Speicher nur in ganzzahligen Vielfachen einer bestimmten Einheitsgröße (z.B. 2k) verwaltet wird. Jedem Einheitsgrößenbereich wird ein Bit zugeordnet, dessen Wert anzeigt, ob der Bereich frei (0) oder belegt (1) ist.
  - **Speicherbelegungstabelle**: Sinnvoll anwendbar, wenn die einzelnen Speicherbereiche eine beliebige Größe haben können. Für jeden Bereich ist ein Eintrag in der Tabelle enthalten, der u.a. die Startadresse und die Länge des Bereichs, sowie den Belegungszustand enthält.
  - **Speicherbelegungsliste**: Die Informationen über die einzelnen Speicherbereiche (Startadresse, Länge, Belegungszustand) sind in einer linearen Liste zusammengefasst.

2.4.3 Virtuelle Speichertechnik

Virtuelle Speichertechnik liegt vor, wenn der Speicheradressraum, den die Befehle eines Programms referieren (**logischer oder virtueller Adressraum**) , getrennt ist vom Adress- raum des realen Arbeitsspeichers, in dem sich das Programm bei seiner Abarbeitung befindet (**physikalischer Adressraum**).

2.4.3.1 Vorteile

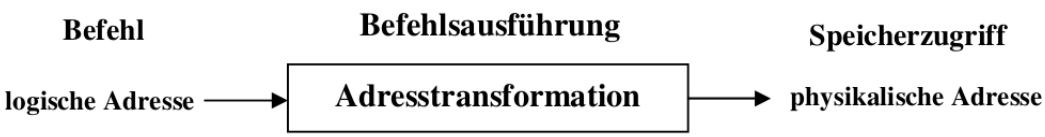
- Die Trennung des logischen Adressraums vom physikalischen Adressraum ermöglicht es, Programme **unabhängig** vom tatsächlich vorhandenen Arbeitsspeicher zu schreiben!
- Der Programmierer kann für die Adressierung in allen Programmen den **einheitlichen** (und sehr großen) virtuellen Adressraum verwenden. Die führt zu einer wesentlich einfachere Programmierung und wesentlich größere Flexibilität.
- Außerdem ermöglicht es **"gleichzeitige Abarbeitung"** (Multitasking, Timesharing) mehrere Programme, deren Gesamtlänge die reale Arbeitsspeichergröße übersteigen würde.

2.4.3.2 Funktion

Der **logische** Adressraum ist meist **viel größer** als der **physikalische Adressraum**. Dies wird erreicht, indem es auf einem Hintergrundspeicher (Plattenspeicher oder Block-Device) abgebildet wird. Zur **Ausführung** muß ein Programm in den **echten Arbeitsspeicher geladen** werden. Wegen der sequentiellen Abarbeitung wird allerdings (in einem gewissen Zeitabschnitt) nie der ge- samte Programmcode und nie der gesamte Datenbereich des Programms benötigt. Es reicht daher aus, nur die jeweils aktuell benötigten Programmcode- und Datenbereichs-Teile (= ”working set”) im Arbeitsspeicher zu halten.

2.4.4 Dynamic Address Translation (DAT) und Memory Management Unit (MMU)

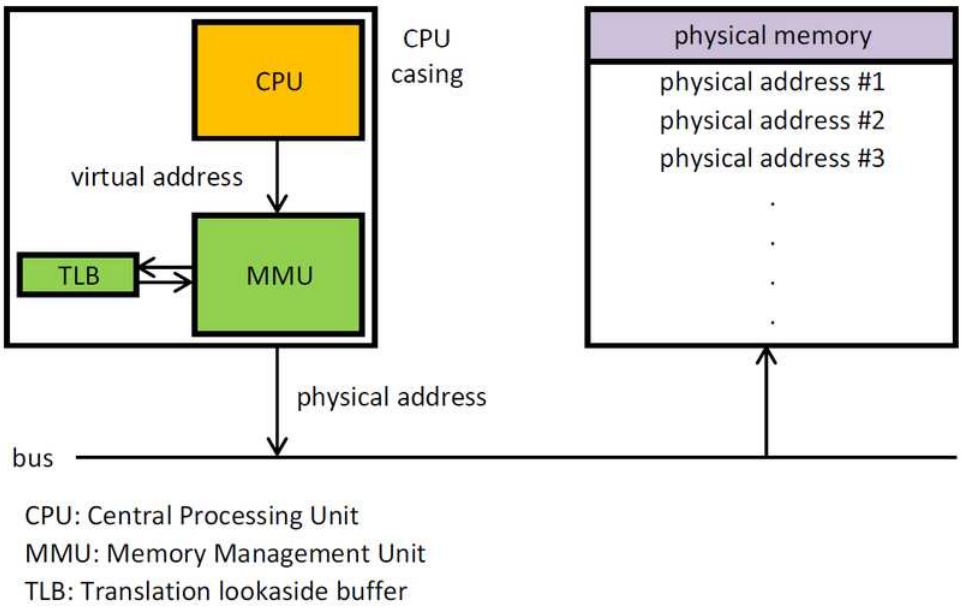
Zur Abarbeitung der einzelnen Befehle eines Programms ist eine Transformation der logischen Adressen in die korrespondierenden physikalischen Adressen erforderlich. Diese Transformation findet aber erst während der Befehlsausführung statt und wird Dynamische Adressumsetzung (DAT) genannt.



Wird bei der Abarbeitung eines Programms eine Adresse in einem noch nicht im Arbeitsspeicher befindlichen Code- oder Datenabschnitt referiert, so muss der entsprechende Abschnitt bei Bedarf (on demand) nachgeladen (und gegebenenfalls ein anderer Abschnitt ausgelagert) werden.
Die virtuelle Speichertechnik ist für den Anwender (und den Anwendungsprogrammierer) vollkommen transparent. Sowohl die Adressumsetzung als auch das Nachladen von Code- und Datenabschnitten erfolgen automatisch und brauchen bei der Programmerstellung nicht berücksichtigt zu werden.

2.4.5 Realisierung der virtuellen Speichertechnik

Die virtuelle Speichertechnik wir durch eine Kombination von Hard- und Software realisiert: - Die Adressumsetzung (DAT) muß bei jedem Speicherzugriff durchgeführt werden und muß daher **sehr schnell geschehen**. Sie **erfolgt** deshalb meist **mittels spezieller Hardware**-Einrichtungen (Speicherverwaltungseinheit, **memory management unit, MMU**). Diese überprüft auch die Zulässigkeit eines Speicherzugriffs und erlaubt eine Realisierung von Speicherschutzmechanismen.
- Das Nachladen von Programmabschnitten wird dagegen durch spezielle Betriebssystem- Komponenten bewirkt. Grundvoraussetzung hierfür ist die Fähigkeit eines Prozessors, einen laufenden Befehl zu unterbrechen (bei erkannter Nachladenotwendigkeit) und nach Behebung der Unterbrechungsursache (erfolgtem Nachladen) diesen erneut aufzusetzen (und dann vollständig auszuführen).



Es gibt zwei prinzipielle Realisierungsmethoden: Segmentierung und Seitenaddressierung (Paging):

2.4.5.1 Realisierungsmethode - Segmentation

Unterteilung des logischen Adressraums **in Abschnitte variabler Größe** entsprechend den logischen Einheiten eines Programms (Hauptprogramm, Unterprogramme, Datenbereiche); d.h. Unterteilung nach logischen Gesichtspunkten. Diese Abschnitte heißen Segmente. Sie stellen die kleinste Austauscheinheit dar. Für jedes System ist eine minimale u. maximale Segmentgröße festgelegt (z.B. zwischen 256 Bytes und 64 kBytes).
Vorteil:
Die Unterteilung erlaubt die Implementierung von Schutzmechanismen. Je nach Betriebssystem und zu- grundeliegender Hardware können einem Segment verschiedene Attribute zugewiesen werden. So können beispielsweise Programm-, Daten- und Stack-Segmente festgelegt werden. Die Speicherverwaltung sorgt dann unter anderem dafür, dass aus dem Programmsegment nur Befehle, aber keine Daten gelesen werden, oder dass umgekehrt Daten im Datensegment nicht als Befehle interpretiert werden. Oft ist es auch möglich, Segmenten Privilegierungsebenen zuzuweisen (Vergleich Linux: Kernel-Mode und User-Mode), sodass auf die entsprechenden Segmente nur von Programmen bestimmter Privilegierungsebenen zugegriffen werden kann. Man kann so zum Beispiel Betriebssystemdaten und -befehle vor Zugriff durch andere Programme schützen. Häufig kann auch die Zugriffsart (zum Beispiel nur lesen, nur schreiben, kein Zugriff) eingeschränkt werden.
Nachteil:
Memory Fragmentation: Speicherzersplitterung - Zwischen den im Arbeitsspeicher befindlichen Segmenten können nicht belegte Lücken entstehen (wenn ein Segment gegen ein kleineres Segment ausgetauscht wird).
Es kann Fälle geben, bei denen der für ein Segment benötigte Speicher nicht geschlossen zur Verfügung steht, obwohl insgesamt genügend freier Speicher vorhanden ist. Dann wird eine Neuordnung der Speicherbelegung durch das Betriebssystem notwendig (Zusammenschieben der Segmente), welches einen umständlichen und zeitaufwendigen Austauschalgorithmus erfordert.

Vielleicht hat jemand von euch schon mal die Fehlermeldung **segmentation fault** bekommen?!? Häufig bei frühere Windows-Versionen im BSoD (Blue Scree of Death). Häufige Auslöser waren Geräte- und andere Hardwaretreiber die in einem privilegierten Modus (Kernel mode) ausgeführt werden, wobei sie direkten Zugriff auf Systemspeicherbereiche und Hardwareschnittstellen haben. Schreibt ein fehlerhafter Gerätetreiber Daten in einen Speicherbereich,

der von anderen Systemteilen (auch anderen Gerätetreibern) benutzt wird, so wird die Systemintegrität verletzt. Wenn das System in solchen Fällen weiterliefe, bestünde die Gefahr der irreversiblen Zerstörung von Daten (zum Beispiel auf der Festplatte). Daher wird das System sofort angehalten und ist nicht mehr bedienbar, was wiederum ebenfalls nachteilige Wirkung auf die Datenintegrität haben kann, insbesondere gehen nicht gespeicherte Daten des Benutzers verloren.

2.4.5.2 Realisierungsmethode - Paging

Die **Seitenadressierung** oder **paging** ist eine Erweiterung der Segmentation Methode welche zur Zeit in allen modernen BS verwendet wird.
Grundliegende Idee:
Unterteilung des logischen und des physikalischen Adressraums in **Abschnitte gleicher Länge**. Diese Abschnitte werden Seiten/page genannt. Es besteht keinerlei Bezug zur **logischen Struktur des Programms!**

- Typische Seitengröße : 512 Bytes ... 4 kBytes. In Linux über den Befehl: `getconf PAGESIZE` ermittelbar
- Eine Seite stellt nun die (einheitliche) Austauscheinheit dar: es entstehen keine unförmigen Löcher mehr da alle Seiten die gleiche Größe haben! Jede Seite problemlos durch jede andere Seite ersetzt werden.
- Für die Auswahl derjenigen Seite, die bei einem Seitenwechsel entfernt werden soll, existieren mehrere Algorithmen (Seitenwechsel-Algorithmen), am gebräuchlichsten ist ”least recently used”(LRU)

Vorteile gegenüber Segmentierung:

- keine Speicherzersplitterung
- schnellerer Nachladevorgang (kleinere Abschnitte)
- relativ einfacher Austauschalgorithmus

2.4.6 Code und Linux Beispiele zum Thema Speicherverwaltung

Kleiner C-Testcode um die virtuellen Speicheradressen von Code, Stack und Heap auszulesen:

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

int main(int argc, char **argv)
{
    int v = 3;
    printf("Code is at %p \n", (void *)main);
    printf("Stack is at %p \n", (void *)&v);
    printf("Heap is at %p \n", malloc(8));
    scanf("input ...");
    return 0;
}
```

Code compilieren und im Hintergrund ausführen (mit dem & am Ende vom Command). Dann im Terminal die angezeigt PID merken und das Programm pmap aufrufen. Syntax `pmap PID -XX`

Super Link um ein praktische Übungen zum Thema Virtual Memory auszuprobieren:
https://blog.holbertonschool.com/hack-the-virtual-memory-c-strings-proc/

2.5 Dateisysteme

2.5.1 Was ist ein Dateisystem?

Betriebssysteme stellen ein – oder auch mehrere – Dateisystem(e) zur Verfügung.
Ein Dateisystem legt fest:

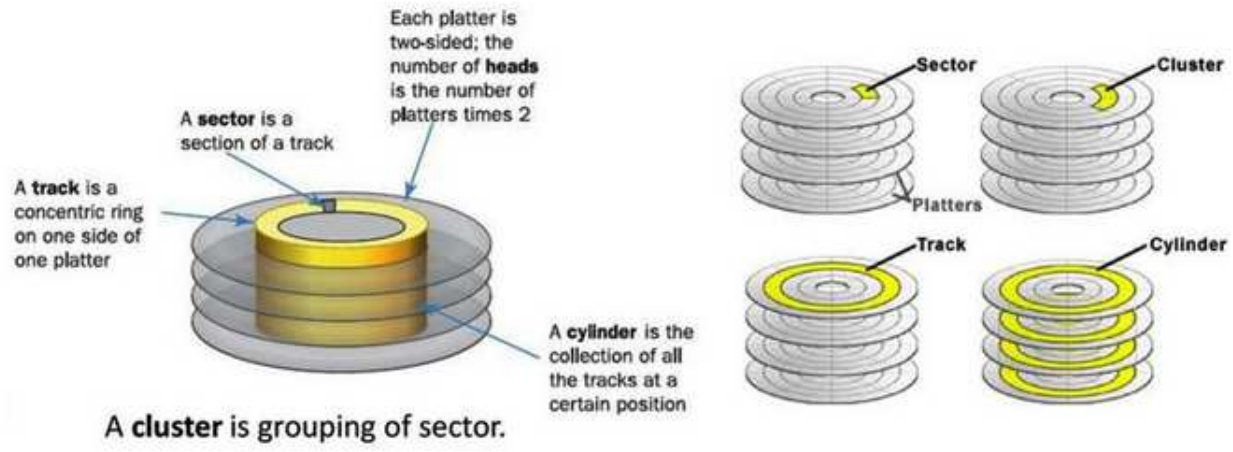
- die Organisationsform und die Bedeutung und Verwendung der auf dem Datenträger vorhandenen Verwaltungsstrukturen sowie
- Randbedingungen für die Dateien (z.B. max. Größe) und ihre Verwendung (z.B. Aufbau u. Länge von Dateinamen)

- Benutzerbezogene Verwaltung von Zugriffsrechte - Funktionen zum Bearbeiten von Dateien (Erzeugen, Löschen, Lesen und Schreiben von Dateien)

In vielen heutigen Dateisystemen werden auch Geräte als – spezielle – Dateien betrachtet. (Linux: *in linux everything is a file*) Ein weiteres zusätzliches Feature kann die Verschlüsselung von Dateien sein, wobei dies nicht unbedingt Aufgabe des Filesystems sein muss, es kann auch über externe Tools verschlüsselt werden. NTFS und auch Ext4 können verschlüsseln.

2.5.2 Dateien

Daten werden extern (außerhalb vom RAM) üblicherweise in Form von Dateien gespeichert. Die Speicherung erfolgt auf **blockorientierten** Hintergrundspeichern (Linux: `lsblk` - command), heute typischerweise auf Magnetplatten bzw nichtflüchtige Halbleiterspeicher - solid state disks.
Dateien können – betriebssystemabhängig – unterschiedlich strukturiert sein. Im einfachsten - und am weitesten verbreiteten - Fall bestehen sie aus einer **linearen Folge** von Bytes (**sequentielle** Dateien). Man unterscheided noch zwischen **Binärdateien** und **Textdateien**:
Eine Textdatei enthält darstellbare Zeichen, welche eventuell durch Steuerzeichen wie Zeilen- und Seitenwechsel untergliedert sind. Sie sind mit einfachsten Text-Editoren lesbar.
Das Gegenstück zur Textdatei ist eine Binärdatei. Sie kann beliebige Bitmuster enthalten und ist üblicherweise nicht mit einfachen Text-Editoren lesbar.
Auf dem Datenträger sind die Dateien blockweise abgelegt. Ein **Block** (häufig Sektor genannt) stellt die Adressiereinheit auf dem Datenträger und gleichzeitig die Transfereinheit von/zum Datenträger dar. Aus diesem Grund unterteilen Betriebssysteme Dateien für den Zugriff ebenfalls in Blöcke (Sektoren).
(Linux: Blockgröße einer Partition herausfinden: `blockdev --getbsz /dev/sda1`) Physikalische Benamung von Festplatten:



2.5.3 Abbildung des Dateiinhaltes auf der Hardware

Unter der physischen Struktur eines Dateisystems (Dateiorganisation) versteht man das Schema der Abbildung des Dateiinhalts auf die durch die Datenträgergeometrie (Zylinder, Kopf, Sektor) definierten physikalischen Sektoren.
Diese Abbildung erfolgt in zwei Schritten:

- **Abbildung des Dateiinhalts auf logische Sektoren**: Dieser Schritt erfolgt durch den Betriebssystem kernel. Das BS betrachtet den Datenträger (unabhängig von der tatsächlichen physikalischen Struktur) als eine lineare Folge von logischen Sektoren.
- **Abbildung der logischen Sektoren auf physikalische Sektoren**: Dieser Schritt wird i.a. durch Gerätetreiber und/oder durch in die Datenträger integrierte Steuereinheiten realisiert.

2.5.3.1 Dateiabbildung auf logische Sektoren

Aufgaben des Betriebssystems im Rahmen der Dateiabbildung auf logische Sektoren:

- **Buchführung** über freie (sowie belegte und defekte) Sektoren
- Allokation von Sektoren bei Erzeugung bzw Erweiterung einer Datei (bzw eines Verzeichnisses)

- Zuordnung von Sektoren zu Dateien (und Verzeichnissen) i.a. ist es zulässig, Dateien beliebige Sektoren in beliebiger Reihenfolge zuzuordnen → **fragmentierte Dateien**
- Buchführung über die **Reihenfolge** der einer Datei (oder einem Verzeichnis) zugeordneten Sektoren
- **Freigabe** der von gelöschten Dateien belegten Sektoren
- Verwaltung der Verzeichnisse und übrigen dateispezifischen Informationsstrukturen

2.5.3.2 Wie funktioniert die Buchführung?

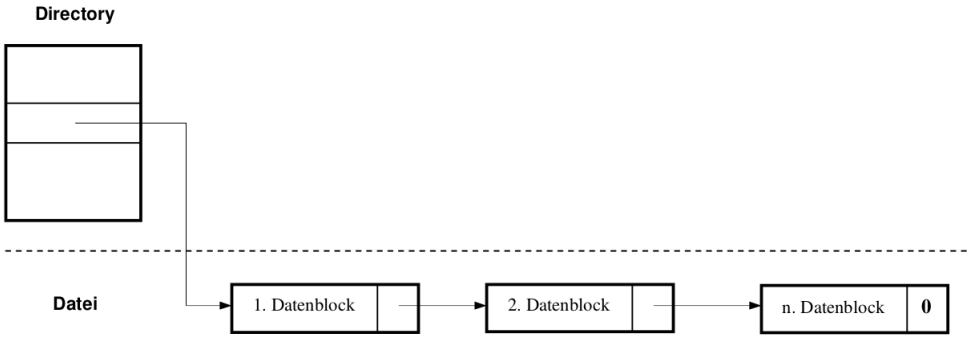
Zur Wahrnehmung der oben erwähnten Buchführungs-Aufgaben braucht es mindestens zwei Dinge:
Übersicht über freie und defekte Sektoren und eine Zuordnungs-Übersicht von Sektoren zu Dateien.
Aus Effizienzgründen können in diesen Übersichten mehrere aufeinanderfolgende Sektoren jeweils zusammengefasst sein (logischer Datenblock = Cluster). In der Realisierung dieser Strukturen unterscheiden sich die verschiedenen Dateisysteme.

2.5.3.3 Realisierungsmethoden für die Übersicht über freie und defekte Sektoren

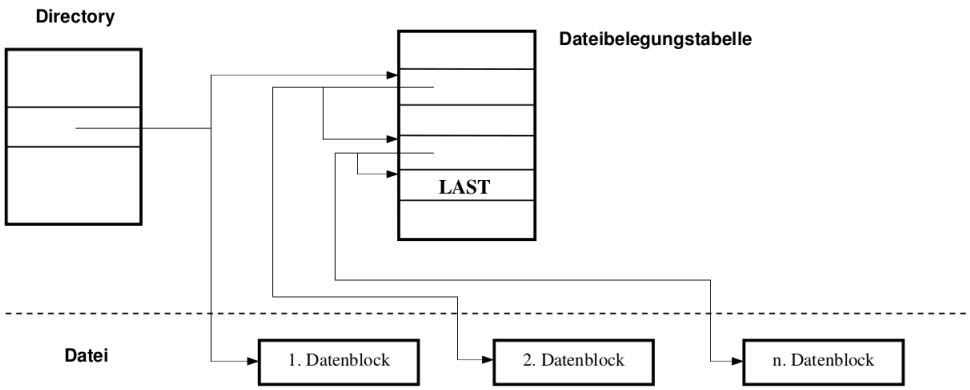
- Markierung des Belegungszustands für alle Sektoren in einer Bitmap (z.B. Bit=0 → Sektor belegt, Bit=1 → Sektor frei).
- Markierung freier und defekter Sektoren in einer Tabelle.
- Zusammenfassen aller freien Sektoren in einer verketteten linearen Liste.
- Defekte Sektoren werden häufig als belegt markiert und in einer Pseudodatei zusammengefasst.

2.5.3.4 Realisierungsmethoden für die Sektor-Datei-Zuordnungs-Übersicht

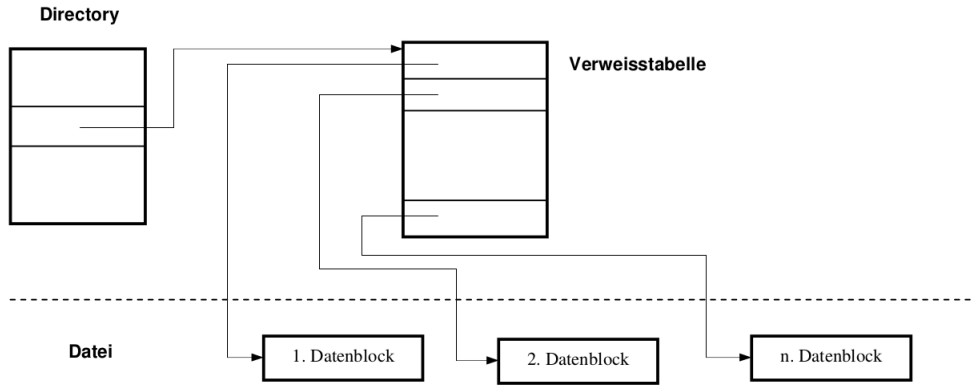
- **Verweiskette** innerhalb der Sektoren einer Datei (interne Verkettung).



- **Verweiskette** in einer **gesonderten** zentralen **Dateibelegungstabelle** (externe Verkettung). Die Dateibelegungstabelle enthält Verweisketten für alle Dateien (und Verzeichnisse). Gleichzeitig enthält sie die Übersicht über freie und defekte Sektoren. Diese sind jeweils durch spezielle Eintrags-Werte gekennzeichnet.

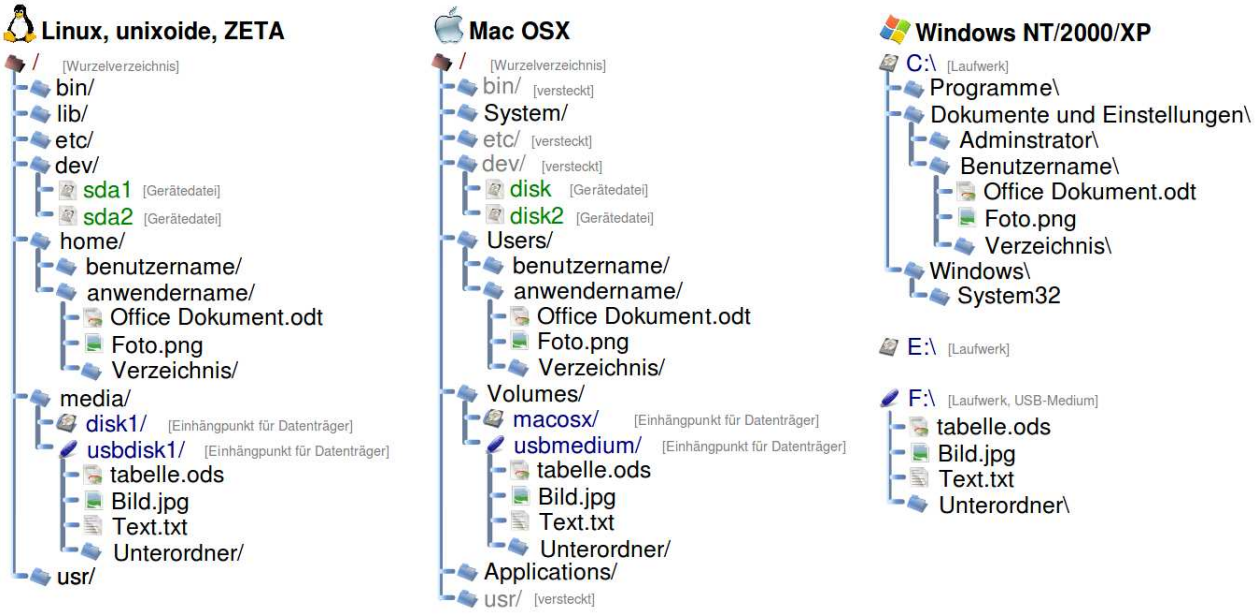


- **Verweistabelle:** Für jede Datei existiert eine eigene Verweistabelle. Die Reihenfolge der Tabellenreihenfolge entspricht der Reihenfolge der Sektoren in der Datei.



2.5.4 Dateiverzeichnisse (Directories)

Um eine vernünftige Zugriffsmöglichkeit auf die Dateien eines Dateisystems zu haben, werden diese in Inhaltsverzeichnisse (Dateiverzeichnisse, Kataloge, Ordner, directories) eingetragen.
Bei den heutigen Dateisystemen sind hierarchisch strukturierte Verzeichnisse (Verzeichnisbaum) üblich:



In einem Verzeichnis können nicht nur Dateien, sondern wiederum Verzeichnisse eingetragen sein. Der Verzeichnisbaum beginnt im Wurzelverzeichnis (root directory). Bei Verzeichnisbäumen werden Dateien nicht durch ihren Namen allein angesprochen, sondern durch einen Zugriffspfad, der die Verzeichnisse, über die man zu der Datei kommt, spezifiziert. Dies kann erfolgen durch:
- einen absoluten Pfad (Beginn im Wurzelverzeichnis) oder durch
- einen relativen Pfad (Beginn im aktuellen (Arbeits-) Verzeichnis)
(Commandline: `cd` -Befehl)

2.5.5 Dateispezifische Informationen

Im Rahmen des Dateisystems werden für jede Datei spezifische Informationen benötigt, die auf dem Datenträger abgelegt sein müssen, z.B.:
- Dateiname (u. gegebenenfalls Dateityp)
- Zugriffsrechte (Dateiattribute)
- Datum der Erstellung / der letzten Änderung / des letzten Zugriffs
- Eigentümer
- Größe
- Speicherort

Diese Informationen können im Verzeichniseintrag der Datei oder in anderen Verwaltungsstrukturen enthalten sein. In Linux werden diese Informationen in **Inodes (Index und Node(Konten))** gespeichert.

2.5.6 Inodes

Dateisysteme unixoider Betriebssysteme – wie Linux und macOS – verwenden sogenannte Inodes. Diese enthalten die Metadaten sowie Verweise darauf, wo Nutzdaten gespeichert sind. An einem speziellen Ort des Dateisystems, dem Superblock, wird wiederum die Größe, Anzahl und Lage der Inodes gespeichert. Die Inodes sind durchnummeriert und an einem Stück auf dem Datenträger gespeichert. Das Wurzelverzeichnis eines Dateisystems besitzt eine feste Inodenummer. Unterordner sind gewöhnliche Dateien, welche eine Liste der darin enthaltenen Dateien mit der Zuordnung der dazugehörenden Inodenummern als Nutzdaten enthalten.
Vereinfachte Struktur eines Unix-Dateisystems:

Boot-Block	Super-Block	Inode-Liste	Datenblöcke

Beispiel: Öffnen der Datei `/usr/myFile`

- Der Dateisystemtreiber liest den Superblock aus, dadurch erfährt er die Startposition der Inodes und deren Länge, somit kann nun jeder beliebige Inode gefunden und gelesen werden.
- Nun wird der Inode des Wurzelverzeichnisses / geöffnet. Da dies ein Ordner ist, befindet sich darin ein Verweis auf die Speicherstelle der Liste aller darin enthaltenen Dateien mitsamt ihren Inodenummern. Darin wird das Verzeichnis `usr` gesucht.
- Nun kann der Inode des usr-Verzeichnisses gelesen werden und analog zum letzten Schritt der Inode der Datei `myFile` gefunden werden.
- Da es sich bei der Datei `myFile` nicht um ein Verzeichnis, sondern um eine reguläre Datei handelt, enthält ihr Inode nun einen Verweis auf die Speicherstelle der gewünschten Daten.

2.5.6.1 Aubau eines Inodes

Jedem einzelnen von einem Schrägstrich / (slash) begrenzten Namen ist genau ein Inode zugeordnet. Dieser speichert (bei ext4 sind Inodes 256 Bytes groß) folgende Metainformationen zur Datei, aber nicht den eigentlichen Namen:

- die Art der Datei (reguläre Datei, Verzeichnis, Symbolischer Link, . . .), siehe unten
- die nummerische Kennung des Eigentümers (UID, user id) und der Gruppe (GID, group id)
- die Zugriffsrechte für den Eigentümer (user), die Gruppe (group) und alle anderen (others)
- verschiedene Zeitpunkte der Datei: Erstellung, Zugriff (access time, atime) und letzte Änderung (modification time, mtime)
- die Zeit der letzten Status-Änderung des Inodes (status, ctime)
- die Größe der Datei
- den Linkzähler (siehe unten)
- einen oder mehrere Verweise auf die Blöcke, in denen die eigentlichen Daten gespeichert sind

2.5.6.2 Verzeichnisse

Verzeichnisse sind Dateien, deren Dateiinhalt aus einer tabellarischen Liste der darin enthaltenen Dateien besteht. Die Tabelle enthält dabei eine Spalte mit den Dateinamen und eine Spalte mit den zugehörigen Inodenummern. Bei manchen Dateisystemen umfasst die Tabelle noch weitere Informationen, so speichert ext4 darin auch den Dateityp aller enthaltenen Dateien ab, so dass dieser beim Auflisten eines Verzeichnisinhalts nicht aus den Inodes aller Dateien ausgelesen werden muss. Für jedes Verzeichnis existieren immer die Einträge . und .. als Verweise auf das aktuelle bzw. übergeordnete Verzeichnis.

2.5.6.3 Linux Befehle zum Thema Inode

Die Inodenummer einer Datei lässt sich mittels des Befehls `ls -i Dateiname` anzeigen, bzw. mit `ls -i` wird der Inhalt des akutellen Unterverzeichnisses mit Inode Nummern angezeigt. Die in einem Inode gespeicherten Metadaten können mit dem Befehl `stat` angezeigt werden. Das Programm `df` (display free disk space, Anzeige der Festplattenbelegung) zeigt mit der Option `-i` die Anzahl der belegten und verfügbaren Inodes auf allen gemounteten Dateisystemen an. (Die Anzahl der möglichen Inodes und somit der möglichen Dateien ist bei manchen Dateisystemen beschränkt; wird die Maximalanzahl erreicht, lassen sich keine weiteren Dateien anlegen. Beim Erstellen von ext2/ext3/ext4- Dateisystemen lässt sich die Anzahl der Inodes einstellen. Bei Datenträgern mit sehr vielen kleinen Dateien muss man also beim Formatieren bereits darauf achten, die Anzahl der Inodes hoch genug zu wählen).

2.5.7 Dateiverweise (Links)

In modernen Dateisystemen gibt es keine feste Zuordnung zwischen Dateinamen und der eigentlichen Datei. Vielmehr wird beim Erstellen der Datei zunächst bloß eine Nummer als Referenz auf die Datei benutzt (je nach Betriebssystem **Inode- oder File-Record-Nummer** genannt) und in einem zweiten Schritt ein Verzeichniseintrag mit dem Dateinamen erzeugt, der auf diese Nummer verweist.
Links ermöglichen, dass eine (nur einmal vorhandene) Datei (oder ein Verzeichnis) über unterschiedliche Verzeichniseinträge, auch unter unterschiedlichen Namen, referiert werden kann. Es gibt zwei Arten dies umzusetzten:

2.5.7.1 Hard Links

Im eigentlichen Sinne bezeichnet harter Link eine Verknüpfung von Dateiname und Datei (letztere repräsentiert durch Inode- oder File-Record-Nummer). Interessant dabei ist – und das ist meistens gemeint, wenn man den Begriff harter Link benutzt –, dass mehrere harte Links auf dieselbe Datei verweisen können, also mehrere Verzeichniseinträge bzw. Dateinamen für ein und dieselbe Datei existieren können.
Hard Links sind nur innerhalb ein und desselben logischen Laufwerks möglich.
Linux Befehl zum Anlegen eines Hard-Links:
`ln /Pfad1/echteDatei /Pfad2/Hardlink`

2.5.7.2 Soft Links

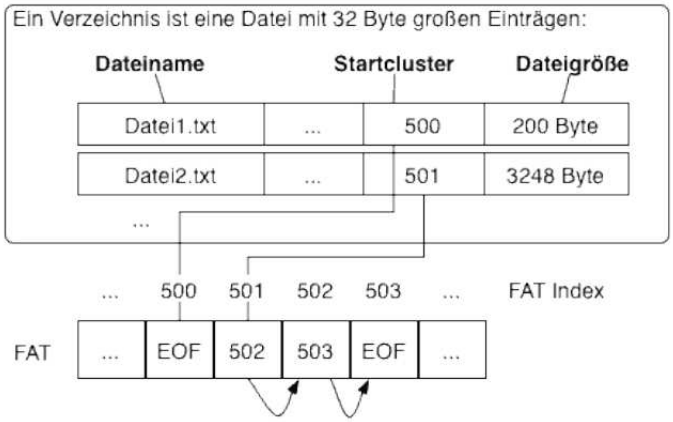
Eine symbolische Verknüpfung, auch symbolischer Link, Symlink oder Softlink genannt, ist eine Verknüpfung in einem Dateisystem (Dateien und Verzeichnisse), die auf eine andere Datei oder ein anderes Verzeichnis verweist. Es ist also lediglich eine Referenz auf die Zieldatei bzw. das Zielverzeichnis. Ein Löschen oder Verschieben der eigentlichen Datei führt üblicherweise dazu, dass die Referenz ins Leere weist. Anders als ein Hardlink ist diese Referenz nicht gleichwertig zum eigentlichen Dateisystem-Eintrag der referenzierten Datei.
Linux Befehl zum Anlegen eines Soft-Links:
`ln -s /Pfad1/echteDatei /Pfad2/Symlink`
Beispiel:

```
markus@markus-desktop:~/Downloads/test$ ls -li
insgesamt 4
79038783 -rw-r--r-- 1 markus markus 12 Okt 29 14:56 orginalFile
markus@markus-desktop:~/Downloads/test$ stat orginalFile
  Datei: orginalFile
  Größe: 12              Blöcke: 8          EA Block: 4096   Normale Datei
Gerät: 811h/2065d       Inode: 79038783    Verknüpfungen: 1
Zugriff: (0644/-rw-r--r--)  Uid: ( 1000/  markus)   Gid: ( 1000/  markus)
Zugriff    : 2018-10-29 15:17:47.565159433 +0100
Modifiziert: 2018-10-29 14:56:49.474790251 +0100
Geändert   : 2018-10-29 15:17:42.893254543 +0100
 Geburt    : -
markus@markus-desktop:~/Downloads/test$ ln orginalFile hardLinkToOrginalFile
markus@markus-desktop:~/Downloads/test$ ln -s orginalFile softLinkToOrginalFile
markus@markus-desktop:~/Downloads/test$ ls -li
insgesamt 8
79038783 -rw-r--r-- 2 markus markus 12 Okt 29 14:56 hardLinkToOrginalFile
79038783 -rw-r--r-- 2 markus markus 12 Okt 29 14:56 orginalFile
79038782 lrwxrwxrwx 1 markus markus 11 Okt 29 15:19 softLinkToOrginalFile -> orginalFile
markus@markus-desktop:~/Downloads/test$ stat hardLinkToOrginalFile
  Datei: hardLinkToOrginalFile
  Größe: 12              Blöcke: 8          EA Block: 4096   Normale Datei
Gerät: 811h/2065d       Inode: 79038783    Verknüpfungen: 2
Zugriff: (0644/-rw-r--r--)  Uid: ( 1000/  markus)   Gid: ( 1000/  markus)
Zugriff    : 2018-10-29 15:17:47.565159433 +0100
Modifiziert: 2018-10-29 14:56:49.474790251 +0100
Geändert   : 2018-10-29 15:18:54.467796913 +0100
 Geburt    : -
markus@markus-desktop:~/Downloads/test$ stat softLinkToOrginalFile
  Datei: softLinkToOrginalFile -> orginalFile
  Größe: 11              Blöcke: 0          EA Block: 4096   symbolische Verknüpfung
Gerät: 811h/2065d       Inode: 79038782    Verknüpfungen: 1
Zugriff: (0777/lrwxrwxrwx)  Uid: ( 1000/  markus)   Gid: ( 1000/  markus)
Zugriff    : 2018-10-29 15:19:16.215353816 +0100
Modifiziert: 2018-10-29 15:19:02.811626922 +0100
Geändert   : 2018-10-29 15:19:02.811626922 +0100
 Geburt    : -
markus@markus-desktop:~/Downloads/test$
```

2.5.8 Eigenschaften ausgewählter Dateisysteme

2.5.8.1 FAT32

Der Name dieses Dateisystems kommt von **File Allocation Table** (Deutsch: Datei Zuteilungs Tabelle). Einer Datei wird eine Menge von Clustern zugeteilt, die Zuteilung in einer **verketteten Listenstruktur** gespeichert - in der FAT. Der Name der Datei und ihr Startclusterverweis in einem Eintrag einer Verzeichnisdatei.



Fat32 addressiert nur mit 28 bit, kann deshalb nur 2^{28} = ca.268 Millionen Addressen (Dateien) verwalten.
Maximale Dateigröße ist 4 GByte, maximale Partitionsgröße ist je nach gewählter Cluster (max. Cluster = 32kB) bis zu 8 TByte.

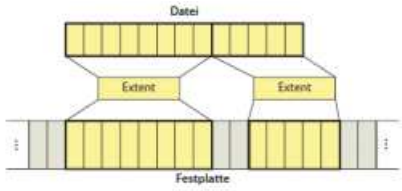
2.5.8.2 NTFS

New Technologie File System: Im Gegensatz zu Inode-basierten Dateisystemen, welche bei Unix zum Einsatz kommen (Konzept: alles ist eine Datei), werden bei NTFS alle Informationen zu Dateien in einer Datei (Konzept: **alles ist in einer Datei**), der **Master File Table, kurz MFT** gespeichert. In dieser Datei befinden sich die Einträge, welche Blöcke zu welcher Datei gehören, die Zugriffsberechtigungen und die Attribute. Zu den Eigenschaften (Attributen) einer Datei gehören unter NTFS Dateigröße, Datum der Dateierstellung, Datum der letzten Änderung, Freigabe, Dateityp und auch der eigentliche Dateiinhalt.
Sehr kleine Dateien und Verzeichnisse werden in der MFT direkt abgespeichert. Größere Dateien werden dann als Attribut in einem Datenlauf gespeichert.
Beim **Formatieren der Festplatte** wird für die **MFT ein fester Platz reserviert**, der nicht von anderen Dateien belegt werden kann. Wenn dieser Bereich mit Informationen komplett gefüllt ist, beginnt das Dateisystem freien Speicher vom Datenträger zu benutzen, wodurch es zu einer Fragmentierung der MFT kommen kann. Standardmäßig wird ein Bereich von 12,5 % der Partitionsgröße für die MFT reserviert.
Beim Speichern von Metadaten wird ein **Journal** geführt, was bedeutet, dass eine geplante Aktion zuerst in das Journal geschrieben wird. Erst dann wird der eigentliche Schreibzugriff auf die Daten ausgeführt, und abschließend wird das Journal aktualisiert. Wenn ein Schreibzugriff nicht vollständig beendet wird, zum Beispiel wegen eines Absturzes, braucht das Dateisystem nur die Änderungen im Journal zurückzunehmen und befindet sich anschließend wieder in einem konsistenten Zustand. Zusätzlich zum Journaling unterstütz NTFS auch noch Datenkompremierung und Verschlüsselung.
NTFS addressiert nur mit 32 bit, kann deshalb nur 2^{32} = ca.4,3 Millarden Addressen (Dateien) verwalten.
Maximale Dateigröße ist 16 TByte, maximale Partitionsgröße ist 256 TByte.

2.5.8.3 Ext4

Ext4 arbeitet mit 48-bit Blocknummern bei einer Standard-Blockgröße von nach wie vor 4 KByte. Das erlaubt eine Dateisystemgröße von bis zu 2^{48} Blöcken à 4 KByte, also einem Exabyte (1024 PByte). Die maximale Dateigröße ist 16 TB.
Die wichtigste interne Verbesserung in Ext4 ist die **Verwendung von Extents** anstelle der in Ext3 verwendeten indirekten Blockadressierung. Bei letzterer speichert ein Inode die Nummern von maximal zwölf 4-KByte-Blöcken. Ist eine Datei größer als 48 KByte, kommen zunächst indirekt adressierte Blöcke (bis 4 MByte), dann doppelt (bis 4 GByte) und schließlich dreifach indirekt adressierte Blöcke zum Einsatz, bei denen eine Blocknummer im Inode auf einen Block mit Blocknummern verweist, die auf Blöcke mit Blocknummern verweisen, die auf Blöcke mit den Blocknummern der Daten verweisen. Dieses klassische Adressierungsschema der Unix-Welt bewährt sich bei kleinen oder sehr stark fragmentierten Dateien, bringt aber bei großen Dateien einen zunehmenden Verwaltungs-Overhead mit sich.

Extents adressieren keine einzelnen Blöcke, sondern mappen stattdessen einen (möglichst großen) Bereich einer Datei auf einen Bereich zusammenhängender Blöcke auf der Platte. Dazu braucht man statt vieler einzelner Blocknummern nur noch drei Werte: Der Start und die Größe des Bereichs in der Datei (jeweils in Dateisystemblöcken) sowie die Nummer des ersten Datenblocks auf der Platte.



Extents mappen Bereiche einer Datei auf Bereiche der Festplatte.

Ext4 ist wie NTFS ebenfalls ein Journaling File-System und unterstützt ebenfalls Verschlüsselung aber keine native Datenkompremierung.

2.5.9 Netzwerk-Dateisysteme

2.5.9.1 Unix: Network File System - NFS

Das Network File System ist ein Protokoll (ursprünglich von Sun Microsystems entwickelt), das den Zugriffauf Dateien über ein Netzwerk ermöglicht. Dabei werden die Dateien nicht wie z. B. bei FTP übertragen,sondern die Benutzer können auf Dateien, die sich auf einem entfernten Rechner befinden, so zugreifen, als ob sie auf ihrer lokalen Festplatte abgespeichert wären.
NFS arbeitet auf dem Netzwerkprotokoll IP. NFSv4 arbeitet nur mit TCP (früher UDP) und benötigt nur noch einen Port (2049), was den Betrieb durch Firewalls erleichtert. NFSv4 wurde maßgeblich durch die IETF entwickelt, nachdem Sun die Entwicklung abgegeben hatte.

2.5.9.2 Windows: Server Message Block - SMB

erver Message Block (SMB), teils auch als LAN-Manager- oder NetBIOS-Protokoll bekannt, ist ein Netz- werkprotokoll für Datei-, Druck- und andere Serverdienste in Rechnernetzen. SMB implementiert ein Netzwerkdateisystem ähnlich wie NFS und ist damit vom zugrundeliegenden Dateisystem des Servers größtenteils unabhängig. In TCP/IP-Netzwerken wurde SMB ursprünglich in NetBIOS over TCP/IP (NBT) über die TCP/UDP-Ports 137–139 gekapselt, die Namensauflösung erfolgte häufig mittels WINS bzw. Broadcasts. Neuere Windows-Versionen nutzen SMB direkt auf dem TCP-Port 445 und lösen Namen per DNS auf.

2.5.9.3 Common Internet File System - CIFS

Der Begriff Common Internet File System (kurz CIFS) wurde 1996 von Microsoft eingeführt und beschreibt eine erweiterte Version von SMB. CIFS baut dabei auf NetBIOS over TCP/IP und SMB auf und bietet neben der Datei- und Druckerfreigabe weitere Dienste wie zum Beispiel den Windows-RPC- und den NT-Domänendienst an. Die Namensauflösung erfolgt dabei weiterhin über NBT-Rundrufe beziehungsweise allgemein den NBT Name Service oder über DNS, wenn NBT nicht zur Verfügung steht.

2.5.9.4 Samba

Samba ist ein freies Programmpaket, das es ermöglicht, Windows-Funktionen wie die Datei- und Druck- dienste unter anderen Betriebssystemen zu nutzen und die Rolle eines Domain Controllers anzunehmen. Es implementiert hierfür unter anderem das SMB/CIFS-Protokoll.
Da die Software unter der GPL frei verfügbar ist, wird sie als Alternative zu Microsoft-Windows-Server- Betriebssystemen eingesetzt. Klassische Verwendung ist der Zugriff auf Windows Ressourcen von Linux Systemen.

2.5.10 Dateisysteme für optische Medien

2.5.10.1 ISO 9660

ISO 9660 ist eine Norm der Internationalen Organisation für Normung (ISO), die ein Dateisystem für optische Datenträger (CD-ROM, DVD-ROM, Blu-ray Disc etc.) beschreibt. Das Ziel dieser Norm ist die Unterstützung verschiedener Betriebssysteme wie z. B. Microsoft Windows, Mac OS und UNIX-Systeme, so dass Daten ausgetauscht werden können - iso-files! ISO 9660 soll durch das Universal Disk Format abgelöst werden.

2.5.10.2 Universal Disk Format - UDF

Das Universal Disk Format (UDF) ist ein von der Optical Storage Technology Association (OSTA) entwickeltes und spezifiziertes, vor allem bei DVDs verwendetes, plattformunabhängiges Dateisystem, welches zunehmend das ältere ISO-9660-Format ablöst. Im Vergleich zu ISO 9660 fallen bei UDF einige Beschränkungen weg.

2.5.11 Virtuelle Dateisysteme

Ein virtuelles Dateisystem (VFS / Virtual Files System) ist eine Schnittstelle zwischen dem Kernel eines Betriebssystems und einem realen Dateisystem.
VFS bietet eine Abstraktions-Schicht, die Applikationen Zugriff auf verschiedene Arten an Dateisystemen sowie auf lokale und Netzwerk-Storage-Geräte ermöglicht. Aus diesem Grund ist VFS auch als Virtual File System Switch bekannt. Es verwaltet ebenfalls das Data-Storage und Dateierfassung zwischen Betriebssystem und Storage-Subsystem. VFS hält außerdem einen Cache mit so genannten Verzeichnis-Lookups vor. Somit lassen sich die Verzeichnisse leicht und schnell finden, auf die oft zugegriffen wird. Sun Microsystems hat eines der ersten VFS-Implementierungen auf Unix-ähnlichen Systemen vorgestellt. VMFS (VMware Virtual Machine File System), NTFS, GFS (Linux Global File System) und OCFS (Oracle Clustered File System) sind alles Beispiele für virtuelle Dateisysteme.

3 Virtualisierung

Virtualisierung in der IT bezieht sich auf die Erstellung virtueller Versionen von Ressourcen wie Computerhardware, Betriebssystemen, Speichergeräten oder Netzwerken. Diese virtuellen Instanzen ermöglichen es, mehrere unabhängige Umgebungen auf einem einzelnen physischen Gerät auszuführen.
Die Virtualisierung funktioniert durch eine Software namens **Hypervisor**, die physische Ressourcen in mehrere isolierte virtuelle Umgebungen aufteilt. Der Hypervisor ermöglicht es mehreren Betriebssystemen, gleichzeitig auf derselben Hardware zu laufen, ohne sich gegenseitig zu beeinflussen. Jede virtuelle Umgebung, auch als **Virtual Machine (VM)** bezeichnet, verhält sich wie ein eigenständiges System mit eigenen Ressourcen wie Prozessor, Speicher und Netzwerkanschlüssen.

3.1 Hypervisor

Es gibt zwei Hauptarten von Hypervisoren: den Typ-1-Hypervisor und den Typ-2-Hypervisor.
Typ-1-Hypervisor (Bare-Metal-Hypervisor):

- Ein Typ-1-Hypervisor wird direkt auf der physischen Hardware des Host-Systems installiert. Er fungiert als Betriebssystem und verwaltet die Ressourcen, um virtuelle Maschinen zu erstellen und auszuführen.
- Diese Hypervisoren arbeiten direkt auf der Hardware und haben direkten Zugriff auf die Ressourcen, was eine effizientere Leistung ermöglicht.
- Beispiele für Typ-1-Hypervisoren sind VMware vSphere/ESXi, Microsoft Hyper-V, Xen und KVM (Kernel-based Virtual Machine).

Typ-2-Hypervisor (Hosted-Hypervisor):

- Ein Typ-2-Hypervisor wird auf einem herkömmlichen Betriebssystem installiert, das bereits auf der physischen Hardware läuft.
- Dieser Hypervisor läuft als Anwendung innerhalb des Host-Betriebssystems und erstellt virtuelle Maschinen. Er nutzt die Ressourcen des Host-Betriebssystems, was möglicherweise zu Leistungseinbußen führen kann.
- Typ-2-Hypervisoren eignen sich häufig für Entwicklungs- oder Testumgebungen sowie für den persönlichen Gebrauch.
- Bekannte Beispiele für Typ-2-Hypervisoren sind z.B. Oracle VirtualBox, VMware Workstation oder Hyper-V.

Beide Hypervisortypen haben ihre eigenen Vor- und Nachteile. Typ-1-Hypervisoren bieten in der Regel eine höhere Leistung und Effizienz, während Typ-2-Hypervisoren einfacher einzurichten und für den persönlichen Gebrauch geeignet sind. Unternehmen wählen ihre Hypervisoren basierend auf ihren spezifischen Anforderungen, wie Leistung, Verwaltbarkeit, Sicherheit und Kosten.

3.2 Vorteile von Virtualisierung

Vorteile der Server-Virtualisierung:

1. **Ressourcenauslastung und Kosteneffizienz:** Durch Konsolidierung mehrerer virtueller Server auf einer physischen Hardwareplattform können Unternehmen ihre Serverressourcen besser nutzen, was die Kosten reduziert.
2. **Flexibilität und Skalierbarkeit:** Server-Virtualisierung ermöglicht es, schnell und einfach neue virtuelle Maschinen zu erstellen, zu implementieren und zu verwalten. Dies erlaubt eine agile Reaktion auf sich ändernde Anforderungen und eine einfache Skalierung der Infrastruktur je nach Bedarf.
3. **Höhere Verfügbarkeit und Redundanz:** Funktionen wie Live-Migration ermöglichen das Verschieben virtueller Maschinen während des laufenden Betriebs von einem physischen Server auf einen anderen, ohne die Benutzer zu beeinträchtigen. Dies verbessert die Verfügbarkeit von Diensten.
4. **Effizientes Management:** Die zentrale Verwaltung von virtuellen Maschinen erleichtert die Konfiguration, Überwachung und Aktualisierung von Servern. Administratoren können Ressourcen schnell zuweisen, Systeme überwachen und Sicherheitsrichtlinien zentral verwalten.

3.2.1 Snapshots

Snapshots sind ein nützliches Feature von Typ-1-Hypervisoren, das es ermöglicht, den aktuellen Zustand einer virtuellen Maschine zu speichern. Ein Snapshot ist eine Momentaufnahme des Zustands der VM zu einem bestimmten Zeitpunkt. Das Erstellen eines Snapshots erfasst den aktuellen Zustand der VM, einschließlich der Festplatten, des Arbeitsspeichers und des Systemstatus. Dadurch können Administratoren später zu diesem Snapshot zurückkehren, falls Änderungen oder Experimente in der VM unerwartete Probleme verursachen.

4 Container

Container sind eine Form der Virtualisierung, jedoch unterscheiden sie sich in ihrer Funktionsweise und ihrem Ansatz von der traditionellen Virtualisierung auf Hypervisor-Ebene, die bei Virtual Machines (VMs) zum Einsatz kommt.
Bei der Virtualisierung mit Containern werden **Anwendungen** und deren **Abhängigkeiten** in **isolierten** Containern ausgeführt. Im Gegensatz zu virtuellen Maschinen, die einen vollständigen Satz an Betriebssystem-Ressourcen replizieren, teilen Container das Betriebssystem des Hosts und verwenden den Kernel des Host-Betriebssystems.

4.1 Geschichte

Die Popularität von Containern begann sich in den frühen 2010er Jahren zu entwickeln, insbesondere mit der Einführung von Docker im Jahr 2013. Docker ist eine der ersten Plattformen, die Containerisierung auf breiter Basis vereinfacht hat und einen großen Einfluss auf die Verbreitung und Akzeptanz von Containertechnologien hatte.
Seit der Einführung von Docker hat sich die Container-Technologie schnell weiterentwickelt und ist zu einer der grundlegenden Technologien für die Bereitstellung von Anwendungen in der Cloud, DevOps-Praktiken und der Microservices-Architektur geworden. Plattformen wie Kubernetes, die die Orchestrierung von Containern ermöglichen, haben ebenfalls dazu beigetragen, die Verwendung von Containern auf Unternehmensebene zu fördern.
Die Vorteile von Containern, wie die effiziente Ressourcennutzung, die Portabilität von Anwendungen und die Skalierbarkeit, haben dazu beigetragen, dass Container sehr beliebt wurden und in vielen modernen Softwareentwicklungs- und Bereitstellungsprozessen eine wichtige Rolle spielen.

4.2 Vor- und Nachteile

4.2.1 Vorteile von Containern:

- **Leichte und schnelle Bereitstellung:** Container ermöglichen eine schnellere Bereitstellung von Anwendungen im Vergleich zu virtuellen Maschinen, da sie weniger Overhead haben und schneller gestartet werden können.
- **Ressourceneffizienz:** Da Container den Betriebssystemkernel des Hosts nutzen, teilen sie sich die zugrunde liegenden Ressourcen effizienter als virtuelle Maschinen, was zu einer höheren Ressourcenauslastung führt.
- **Portabilität:** Container sind portabel und können in verschiedenen Umgebungen laufen, unabhängig von der zugrunde liegenden Infrastruktur, was die Bereitstellung und Skalierung erleichtert.
- **Isolation:** Container isolieren Anwendungen voneinander und vom Host-Betriebssystem, was Sicherheitsvorteile bietet und das Risiko von Konflikten zwischen Anwendungen reduziert.

4.2.2 Nachteile von Containern:

- **Eingeschränkte Betriebssystemabhängigkeit:** Container teilen sich den Kernel des Host-Betriebssystems, was bedeutet, dass sie inkompatibel mit verschiedenen Betriebssystemen sein können.
- **Sicherheitsrisiken:** Eine unsachgemäße Konfiguration oder Sicherheitslücken in Containern können zu Sicherheitsproblemen führen, insbesondere wenn mehrere Container auf demselben Host laufen.
- **Performance-Overhead:** Während Container weniger Overhead als virtuelle Maschinen haben, besteht immer noch ein gewisser Overhead aufgrund der zusätzlichen Abstraktionsschicht.
- **Mehr Speicherplatzbedarf:** Im direkten Vergleich zu einer **nativen Anwendung** kann der Speicherplatzbedarf eines Containers daher größer sein, da der Container alles enthält, was zur Ausführung der Anwendung in einer isolierten Umgebung notwendig ist. Dies kann dazu führen, dass der Speicherplatzverbrauch höher ist als bei einer nativen Anwendung, die nur die benötigten Ressourcen des Betriebssystems verwendet.

Obwohl Container viele Vorteile bieten, müssen potenzielle Nachteile berücksichtigt werden, um eine angemessene Konfiguration, Sicherheit und Leistung zu gewährleisten.

4.3 Docker unter Linux installieren und testen

1. **Docker installieren**
   Docker kann in den meisten Linux Distributionen einfach über die Packetverwaltung installiert werden. Zum Beispiel in Debian basierten Distros mittels:
   `sudo apt install docker.io`
   Dann den eigenen User zur Docker-Group hinzufügen:
   `sudo usermod -a -G docker $USER` Nun müsste man sich ab- und wieder anmelden damit man zur Docker Gruppe gehört, oder wenn man ungeduligt ist, kann man mit `newgrp docker` gleich zur Gruppe docker wechseln.
   Eventuell kann es zusätzlich notwendig sein, den Docker-Service neu zu starten.
   `sudo systemctl restart docker`
   Die Docker Images werden lokal meist unter `/var/lib/docker/` gespeichert.

2. **Docker Installation testen**
   Mittels dem Befehl:
   `docker run hello-world` kann man die Installation testen. Der lokale Docker-Service versucht dabei den hello-world Container von https://hub.docker.com/ zu laden und auszuführen.
   Möglicher Output:

```
markus@pc:~$ docker run hello-world
Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
c1ec31eb5944: Pull complete
Digest: sha256:ac69084025c660510933cca701f615283cdbb3aa0963188770b54c31c8962493
Status: Downloaded newer image for hello-world:latest

Hello from Docker!
This message shows that your installation appears to be working correctly.

To generate this message, Docker took the following steps:
 1. The Docker client contacted the Docker daemon.
 2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
    (amd64)
 3. The Docker daemon created a new container from that image which runs the
    executable that produces the output you are currently reading.
 4. The Docker daemon streamed that output to the Docker client, which sent it
    to your terminal.

To try something more ambitious, you can run an Ubuntu container with:
 $ docker run -it ubuntu bash

Share images, automate workflows, and more with a free Docker ID:
 https://hub.docker.com/

For more examples and ideas, visit:
 https://docs.docker.com/get-started/
```

   Mit dem Befehl `docker images` kann man sich die lokalen Images anzeigen lassen. Zu diesem Zeitpunkt sollte nur das frisch heruntergeladene hello-world image vorhanden sein:

```
markus@pc:~$ docker images
REPOSITORY    TAG       IMAGE ID       CREATED        SIZE
hello-world   latest    d2c94e258dcb   7 months ago   13.3kB
```

4.4 Selbst einen Container erstellen - Beispiel Python-Programm

Ziel: Man will aus einem kleinen Python Programm, welches von einem externen File abhängt, einen Container bzw. ein Docker-Image erstellen.
Das Python Programm soll einfach ein .txt File öffnen und den Inhalt Zeile für Zeile ausgeben.
Hier der Python-Code des Programms:
`my_app.py`

```
'''
little python app to test docker
just opens a file and prints its content line by line
markus
dec.2023
'''
with open("numbers.txt","r") as fobj:
    for line in fobj:
        print(line.strip())
```

Das fertige Image soll sowohl das Programm, das File (numbers.txt) und eine komplette Python-Umgebung enthalten, damit es Plattform unabhängig ausgeführt werden kann.
Annahme: Das Python-File und das numbers.txt File liegen im folgenden Verzeichnis:
`/home/markus/FSST/my_app/`

1. **Docker-File erstellen**
   Man erstellt eine Datei mit dem Namen Dockerfile (keine File-Endung!) im gleichen Verzeichnis wie das Beispiel Python-Programm. Der Inhalt des Files sieht so aus:

   ```
   # Verwendung des offiziellen Python-Images als Basis (wird sehr groß)
   # FROM python:3
   # Oder man kann auch spezielle Python-Images verwenden wie z.B.:
   # FROM python:3.10-alpine -> dadurch wird das image file nicht so groß
   FROM python:3.10-alpine

   # Arbeitsverzeichnis im Container festlegen
   WORKDIR /usr/src/app

   # Kopieren aller lokalen Files (my_app.py und numbers.txt) in das
   # Arbeitsverzeichnis im Container
   COPY . .

   # Installation von benötigten Abhängigkeiten (falls vorhanden)
   # Beispiel: RUN pip install <Paketname>

   # Befehl zum Ausführen des Python-Programms
   CMD ["python", "./my_app.py"]
   ```

2. **Container/Image erstellen**
   Im Ordner, in dem die Python-App, das numbers.txt File und das Dockerfile sind folgenden Befehl ausführen:
   `docker build -t my-first-container .`

3. **Neues Image anzeigen**
   Mittels
   `docker images`
   wird das neue erstellte Image angezeigt:

3. Schritt: Container erstellen Öffnen Sie die Befehlszeile oder das Terminal und navigieren Sie zum Verzeichnis, in dem sich das Dockerfile und das Python-Programm befinden. Führen Sie den folgenden Befehl aus, um den Container zu erstellen:
`docker build -t my-first-container .`
Dabei werden alle Abhängigkeiten z.B. die Python3 Umgebung ebenfalls als Image von Docker-Hub heruntergeladen und dann das neue Image gebaut.

Output:

```
markus@pc:~/Beispiele/Docker$ docker build -t my-first-container .
DEPRECATED: The legacy builder is deprecated and will be removed in a future release.
            Install the buildx component to build images with BuildKit:
            https://docs.docker.com/go/buildx/

Sending build context to Docker daemon  4.096kB
Step 1/4 : FROM python:3.10-alpine
3.10-alpine: Pulling from library/python
661ff4d9561e: Pull complete
44cda88cd45d: Pull complete
48c91875651e: Pull complete
4a3bf9ff3965: Pull complete
8c653295a5ef: Pull complete
Digest: sha256:8a66e74c0581e0c7b8b481fe7f0444b2fd599685d4fb525c1047da4f6f44fdaa
Status: Downloaded newer image for python:3.10-alpine
 ---> 5fee4875cce7
Step 2/4 : WORKDIR /usr/src/app
 ---> Running in 338252bc27db
Removing intermediate container 338252bc27db
 ---> 1b9d01516720
Step 3/4 : COPY my_app.py .
 ---> fc4353acd6af
Step 4/4 : CMD ["python", "./my_app.py"]
 ---> Running in e138fcb4f649
Removing intermediate container e138fcb4f649
 ---> 915a60af5b31
Successfully built 915a60af5b31
Successfully tagged my-first-container:latest


markus@pc:~/Beispiele/Docker$ docker images
REPOSITORY           TAG           IMAGE ID       CREATED          SIZE
my-first-container   latest        915a60af5b31   11 seconds ago   50.2MB
python               3.10-alpine   5fee4875cce7   12 days ago      50.2MB
hello-world          latest        d2c94e258dcb   7 months ago     13.3kB
```

Man sieht, dass das neue Image über 50MByte groß ist. Wenn die offizielle Python3 Umgebung verwendet worden wäre, wäre es über 1GByte groß geworden!

4. **Container ausführen** Nachdem das Image erstellt wurde, kann man es nun mit folgendem Befehl ausführen (es wird zum Container):
`docker run -it my-first-container`