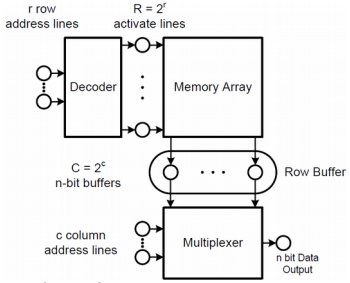
Bsys2 Cheatsheet FS17

Speicher

Random Access Memory



- **Static RAM (SRAM)**: für Register und Cache
- **Dynamic RAM (DRAM)**: für grosse Speicher, höhere Speicherdichte, günstiger
- Bursts sind schneller als Einzelzugriffe, weil Reihenpuffer schon gefüllt
- Speicherchip auf -Rang auf -Modul
- **Lokalitätsprinzip** (nur kleine Teile des Speichers nötig)
  - **Zeitlich**: gleiche Speicherstelle mehrmals gelesen/geschrieben → bsp: Aggregation
  - **Räumlich**: Benachbarte Speicherstellen gelesen/geschrieben → bsp: Array Iteration

Cache

Fully Associative Cache: Jede Speicheradresse kann in jedem Cacheeintrag gespeichert werden T(A)=A
Direct-Mapped Cache: Jede Speicheradresse kann nur in genau einem Cacheeintrag gespeichert werden: Schnell, aber viele Kollisionen
K-Way Set Associative Cache: Besteht aus K Direct-Mapped Caches
Adresse besteht aus **Tag**, **Set** und **Offset**
Look-Aside: Cache und Speicher gleichzeitig
Look-Through: Zugriff zuerst an Cache, bei Miss an Speicher; Reduziert Daten auf Speicherbus
Write-Through: Synchrones Schreiben in Cache und Speicher; Blockiert CPU
Write-Back: Asynchrones Schreiben in Cache, Cache -> Speicher; Komplexer bei Multiprozessorsystemen

Stack / Heap

Stack: Lokale, nicht-statische Variablen, Argumente und Rücksprungadresse
Heap: Dynamisch, reservierbarer Speicher
Interne Fragmentierung: Waste im reservierten Speicher / **Externe Fragmentierung**: Nicht zuweisbare Blöcke
Speicherung der Metadaten: Zentral (Separat) / Dezentral (Metadaten liegen vor und/oder nach Blöcken; kein Schutz; schnelle Rekombination)

Suchalgorithmen

First Fit: Erste Lücke → Lücken am Anfang
Best Fit: Kleinste Lücke → langsam; grösster Verschnitt
Next Fit: Nächste passende Lücke → verteilte Lücken; schlechter als FF
Worst Fit: grösste Lücke → langsam; schlecht
Quick Fit: Bereiche mit Metainfos → schnell; langsame Rekombination; aufwändig

Heap-Verteilstrategien

Variable Zuordnungsgrösse, **Ganzzahliges Mehrfaches**, **Grössenklassen** (Keine Rekombination; bereiche gleicher Grösse; Freilisten), **Buddy-System** (minimal 2^k; unterteilt für best fit; Buddy = Bits der Startadresse gleich ausser Bit n; Freilisten); **Kombination** (Buddy im OS, malloc in Applikation nach FF)

Monoprogrammierung

OS unterstützt nur einen einzigen Prozess; Verwendet absolute direkte Adressen, geteilt mit dem OS; meist Embedded-Systeme

Multiprogrammierung

Mehrere Prozesse quasi-parallel; Konflikte bei Prozessen, die nur Monoprogrammierung kennen; Prozesse müssen verschoben werden

Adressräume

Statisch: Vom Linker festgelegt; zur Compile-Zeit; Erweiterung der Monoprogrammierung möglich; Volle Kontrolle; z.B. Overlays oder absolute Verteilung
Dynamisch: vom OS festgelegt; Linker nutzt; z.B. Relokation oder relative Adressen

Overlays

Reduktion gleichzeitig geladener Prozesse; Können ersetzt werden; Statisch

Relokation bei Prozessstart

Compiler → absolute Adressen → Relokationstabelle; OS führt Relokation durch und aktualisiert Tabelle

Relative Adressen

CPU-Support (Basis/Segment-Register); Compiler erzeugt Programm mit Adressen relativ zum Basis-Register → schnelle Relokation

Speicherschutz

Erfordert Hardware-Support; Nur OS editiert Metadaten; Absolute Adressen → Code Table (Adressbereich und Code) → Kontrolle Code-Register; Relative Adressen → Base- und Limit-Register; Bei Zugriff erfolgt Check

Virtuelle/Logische/Lin. Adressen

Werden von MMU in reale/physische Adressen übersetzt und prüft Berechtigung; MMU durch OS konfiguriert; Cache arbeitet mit realen Adressen, CPU mit virtuellen;
Ungültiger Zugriff → MMU signalisiert **Fault-Interrupt** → CPU aktiviert OS-Interrupt-Handler
Ermöglicht Effekte:
- Speicher auslagern → Fault-Handler lädt Speicherbereche nach → Prozess fortsetzen
- Mehr speicher pro Prozess
- Monoprogrammierung → keine Rücksicht auf andere Prozesse nötig

Seitenbasierter Speicher (Page)

Alternativen: Segmentbasiert, Kombiniert (Segment- und Seitenbasiert)
Page Frames: Hauptspeicher; Speicherplatz für genau eine Page;
Pages: Virtueller Adressraum; repräsentiert Daten;
Page Frame/Page Number = Startadresse ohne Offset-Bits
Pro Prozess: virtueller Adressraum, Paging-Table

Memory Management Unit (MMU)

Benötigt **Page Table Register** → wird bei Prozesswechsel aktualisiert
Virtuelle Adresse wird aufgeteilt in Offset (12 LSB) und Page Number → Page Number wird auf Frame Number übersetzt → Frame Number + Offset = reale Adresse

Page Table

Wird von OS konfiguriert; zusätzliche Metainformationen (letzte Verwendung, Status, etc.)

Single-Level

Array mit Page Frame Number und Page-Status-Bits; Schneller Lookup über Index (=Page Number); Grösse abhängig vom virt. Addressraum

Two-Level (spez. Multi-Level)

Page Number wird aufgeteilt → Directory Index und Page Index → Page Directory zeigt auf Page Tables; Reduktion des Speichers für PT: Alle Used-Bit in PT = 0 → Used-Bit in PD = 0

Multi-Level

Zusätzliche **Page Directory Pointer Table** oder **Page Map Level 4**.
Vergrössert virtuellen Adressraum

Inverted

Übersetzung Page Frame Number → Page Number; eine PT für alle Prozesse → Suche dauert lange → nicht Praktikabel

Hashed

Index = Hash von Page Number; **Hash-Kollision** → Iteration über Linked List; verwendet in TLB

Translation Lookaside Buffer (TLB)

Cache für häufige Adressen; Profitiert vom Lokalitätsprinzip; Kann hierarchisch implementiert sein; Teils getrennt für Instruktionen und Daten

Interprozesskommunikation (IPC)

Shared Memory (Schneller)
- Transparent → wie normaler Speicher
- Prozesse können sich gegenseitig beeinflussen
- Selber Page Frame für beide Prozesse

Message Passing (Sicherer)
- OS kopiert Daten zwischen Prozessen
- Beide Seiten müssen bereit sein
- Produziert Overhead

Paging

1. OS prüft Lokalisierungsinformationen auf Gültigkeit
2. Wenn Hauptspeicher bereits voll, OS entscheidet welche Page daraus entfernt wird
3. OS schedulet Page-Transfer von sekundärem Speicher und hält Prozess an
4. OS schedulet nächsten Prozess

Sobald Page geladen wird Prozess dort fortgesetzt, wo Interrupt geworfen wurde → wiederholter Zugriff

Thrashing / Dreschen

Häufiges Pagen (viele Prozesse, wenig Hauptspeicher → System mit Paging ausgelastet)
Vermeidung: Mehr Hauptspeicher, Lastensteuerung (beschränke Anz. Prozesse) oder bessere Paging-Strategie

Ladestrategien (Fetch Policies)

Demand Paging: Lade Pages bei Interrupt; minimaler Auwand; lange Wartezeiten
Prepaging: Selten; Laden vor Verwendung; benötigte Pages müssen ermittelt werden
Demand Paging with Prepaging: Lädt ganze Cluster bei Bedarf; nutzt Lokalitätsprinzip; Weniger Page-Faults und Blocktransfer; Viel Auwand; Nicht benötigte Pages werden geladen → in der Praxis bewährt
Heuristics: Algorithmen zur Bestimmung

Entladestrategien (Cleaning Policies)

Beeinflusst Verzögerung bei Page-Fault

Demand Cleaning: Page wird bei Wiederverwendung von Frame zurückgeschrieben; minimaler Aufwand; erhöhte Wartezeit
Precleaning: Vorausschauendes Schreiben modifizierter Pages; reduzierte Wartezeiten; mehr Aufwand, falls Page erneut verändert
Page Buffering: Listen mit un-/veränderten Pages; gruppenweise geschrieben; schnellere Auwahl als Preleaning, sonst vergleichbar

Verdrängungss. (Replacement Policies)

Massiver Einfluss auf Systemperformance
Zusammenspiel von MMU (setzt Statusbits) und OS (löscht Statusbits)
Statusbits: R = read; M = modified; U = used
Optimal: Entferne späteste benötigte Page
FIFO: Ersetzt älteste Page; Problem: Häufig benutzte Pages werden entfernt und gleich wieder geladen; Problem: Beladys Anomalie → Mehr Hauptspeicher → Mehr Page-Faults
Second Chance: Erweitertes FIFO um Statusbits; Falls R = 1 → wieder einfügen; Schreiben falls alle verwendet
Clock: Effiziente Impl. Second Chance → Pointer
Least Recently Used: Ersetze längste unbenutzte Page → MMU setzt Timestamp; hoher Aufwand in HW; SW-Lösung mit Timer-Interrupt; nahe am Optimium
Not/Least Frequently Used: OS zählt Intervalle wo R=1 → Counter pro Entry; Page mit niedrigstem Counter wird entfernt → alte Pages bleiben lange → Aging: c = c/2 + R*2^n
Working Set: Behalte zuletzt verwendete Pages; Fault-Interrupt → Speichern des Timestamp wo R=1 (kürzlich verwendet); Entferne, wo R=0 und jetzt – T › threshold → Älteste, falls keine, Bevorzuge M =0
Working Set mit Clock: Clock enthält Working Set Informationen; keine Iteration durch PTEs; Praxistauglich

Memory-Mapped Files

Paging-System bildet virtuelle Pages auf Dateien ab (Pagefile, Swap-Partition); Beliebige Dateien in Primärspeicher laden → Array-basierter Zugriff → Automatisches laden/schreiben
Einsatzbeispiel: Programmausführung → OS mappt Page auf Executable, lädt erste Page mit Einstiegspunkt → schneller Start

Ein-/Ausgabe

Register: Command, Status, Data
Unterschied: Beim Speicher sind geschriebene und gelesene Daten identisch.

Memory-Mapped I/O

Geräte am Speicherbus → **pro Gerät ein Adressbereich** → Speicher verdeckt; CPU-Instruktion wie Speicherzugriff → transparent
Probleme bei Compiler-Optimierung → volatile

Port-Mapped I/O

Geräte besitzen **separaten Bus**; Extra CPU-Instruktion und Systemunterstützung → gesamter Speicher
Via Speicherbus: Zusätzliche Steuerleitung zur Selektion Gerät/Speicher → 1 Bit mehr Adressraum; einfaches Systemdesign

Kommunikationsmechanismen

CPU arbeitet asynchron zu Gerät → Rendezvous

Programmgesteuert (Polling)

- Busy Wait: Blockiert, schnelle Geräte
- Interval: Embedded, genauer Zeitpunkt

Interruptgesteuert

Gerät unterbricht Software, wenn bereit
Interrupt-Controller: mit jedem Gerät und CPU-Interrupt verbunden → Interrupt# pro Gerät
Interrupt-Vektor-Tabelle: Pointer auf Interrupt-Service-Routine; Interrupt# als Index
Achtung: Kein OS-Support in Routine; Kritischer Abschnitt → Crash; Verschachtelte Interrupts; Reihenfolge
Interrupts **maskieren/deaktivieren** ausser Non-Maskable Interrupt.

Direct Memory Access (DMA)

DMA-Controller steuert Speicherbus für CPU → Gerät kann direkt Daten in Speicher kopieren

1. CPU programmiert DMA für Transfer
2. CPU gibt Speicherbus an DMA frei
3. DMA lässt Gerät direkt in Speicher kopieren
4. Für jedes Datum legt DMA Adresse und Gerät legt Daten auf Speicherbus
5. Nach Beendigung wirft DMA Interrupt

Betriebsarten: **Einzeltransfer, Blocktransfer**

Treiberarchitektur

OS verwaltet Hardware → OS-API
HW-Klassen → Abstraktion → Schichten-Modell
Microkernel: Standardisierte Interrupt-Handler steuern Treiber-Prozesse → Separation

Windows GUI

Client-Area: Sichtbarer Fensterinhalt
Handle zeigt auf Objekt (OS-Ressource)
Manipulation Objekte über OS-API (Black Box)
Window-Procedure: Callback für OS, verarbeitet jeweils eine Message
Window-Class: Style, Icon, Cursor, Menu
Controls = Window: selbe Window-Procedure für alle Instanzen; Kommunikation über Messages

Messages

OS verteilt Messages → **Message-Loop**
WM_PAINT, WM_TIMER, WM_QUIT werden vom OS aus Performance-Gründen besonders behandelt → zurückgehalten bis MQ leer
Keyboard-Messages → **TranslateMessage**

Message Queue

FIFO-Prinzip, pro GUI-Thread erzeugt
PostMessage: Einreihen in MQ, Non-blocking
SendMessage: umgeht MQ, Blocking; z.B. Child
GetMessage: Blocking; im GUI-Thread → Loop
DispatchMessage: Aufruf Window-Procedure

Unicode

Maximal 17 * 64k = 1'114'112 Code-Points (CP)
[D800, DFFF] reserviert, keine gültigen CP
UTF-16/UTF-32: Little/Big Endian (innerhalb CU)

UTF-32: 4 Byte

Häufig intern in Programmen; nur 21 Bit verwendet; alles in einer CU

UTF-8: 1/2/3/4 Byte

8-Bit pro CU → pro CP 1 bis 4 CU; ASCII-Kompatibel (bis 7F)

UTF-16: 2/4 Byte

16-Bit pro CU → pro CP 1 oder 2 CU; Bei 2 CU→ Surrogate-Pairs

$Q = P - 0x1'0000$ (also ist Q in [0,F'FFFF])

$U_1 = 1101'10Q_{19}...Q_{10} = D800 + Q_{19}...Q_{10}$

$U_0 = 1101'11Q_9...Q_0 = DC00 + Q_9...Q_0$

C-Runtime verwendet ASCII-Codepages→ char mit 8 bit; Windows UTF-16 → wchar_t mit 16 bit; Win-API: UTF-16 W-Suffix, char mit A-Suffix

Windows Ressourcen

Resource-Script → Resource Compiler → res-File → Linker + .obj → .exe
Ressourcen müssen vom Programm geladen werden → Handle

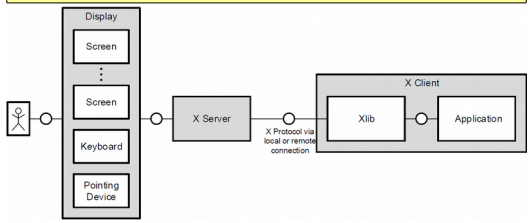
Device-Context

Abstraktion von Geräten; gibt an was Gerät kann; Bitmap in DC; Virtueller DC → Window

X Window System

Ziele: Hardware-Unabhängig, Netzwerkfähig über verbundene verteilte Systeme, Portabilität
X legt Gestaltung nicht fest, unterschiedliche Bedienphilosophie möglich
GUI-Architektur: **X Window System, Window Manager** (Verwaltung sichtbarer Fenster; Umrandung; Knöpfe) und **Desktop Manager** (Taskleiste; Filemanager; Papierkorb)

X Client / X Server



X Protocol

Requests: Dienstanforderung Client → Server (Draw Line, Change Colour Table, get window position)
Replies: Antwort auf bestimmte Requests (oft ohne) Server → Client
Events: Spontane Ereignismeldungen Server → Client (Mouseclick, Mouse move over window)
Errors: Fehlermeldungen auf vorangegangene Requests Server → Client
Nachrichtenpufferung:

- Requests (Clientseitig): Nur nötige Pakete werden übertragen, Gruppierung für Effizienz
- Events (Server- und Clientseitig): Netzwerkverfügbarkeit; Verarbeitung auf Clientseite

Xlib / X Toolkits

Xlib: C Interface für X Protocol; Funktionen und Datentypen; sehr Low-Level
Toolkits verwenden Xlib: Standardelemente (Widgets) für z.B. command buttons, menus

X Ressourcen

Server-Seitige Datenhaltung zur reduktion des Netzwerkverkehrs; halten Informationen im Auftrag des Clients → Ressourcen-Id; komplexe Datenstrukturen müssen nicht kopiert werden
Beispiele: Window, Pixmap, Colormap, Fonts, GC

X Grafikfunktionen

Kompression der darzustellenden Farben → Colormap; Zeichnen von Formen und Text (benötigen GC) → Window oder Pixmap als Ziel

X Event Handling

Eingaben und Systemevents; Client verarbeitet Events oder leitet sie weiter; Filterung möglich; Events müssen abgefragt werden; Alle sind gleich gross

Dateisysteme

Teil des OS / Struktur des Datenträgers
Logische Datei: Verwaltete Einheit von Bytes; Vollständiger Inhalt kann verwendet werden
Metadaten/Dateiattribute: Zur logischen Datei gehörende Informationen; nicht alles sichtbar
Absoluter Pfad: beginnt mit „/“, startet im Root
Relativer Pfad: relativ zu Workdir
Kanonischer Pfad: logisches Verzeichnis, max. 1
Berechtigung (Oktal): r=4, w=2, x=1

POSIX API

Direkter, unformatierter Zugriff auf Dateiinhalt
File-Descriptor: Index auf File Table des Prozess → Index auf Global File Table; merkt sich Offset; STDIO sind 0-2

C API (Unix-Orientiert)

Streams: Abstraktion von File/Pipe
Buffered (mit Offset)/Unbuffered
FILE Structs: Stream-Informationen

Datenträger

Partition: Teil eines Datenträgers; gleich behandelt
Volume: Datenträger oder Partition davon
Sektor: Kleinste log. Einheit eines Volumens mit Header, Daten und Error-Correction-Codes (4KB)
Format: Layout der logischen Strukturen

FAT32

File Allocation Table (FAT) → Cluster
32-Bit Cluster-Adressen
- **Reservierter Bereich**: 32 Sektoren; Metadaten fürs Volume; Sektor 0 = Bootsektor; Angaben zu Grösse, Volume ID, Label, Root-Directory
- **FAT-Bereich**: FAT+Kopie; abhängig von Volume
- **Bereich für Datei- und Verzeichnisdaten**

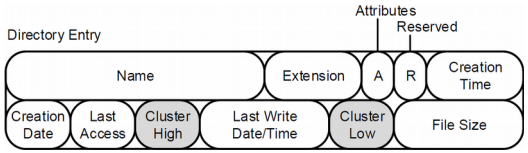
FAT-Eintrag: (Unterste 28 Bit relevant) 0/1 → Not used; 2-FFF'FFF5 → Next Cluster; FFF'FFF7 → Damaged; FFF'FFF8-FFF'FFFF → End of Chain
Dateien und Verzeichnisse sind **Cluster-Chains**

Cluster

Sektoren in Cluster zusammengefasst; zusammenhängend; erste Cluster# = 2

Verzeichnisse

Tabelle von Einträgen à 32Bit; alle Informationen über Datei/Unterverzeichnis; Blätter sind Inhalt von Files oder Cluster 0 (Empty File/Dir)



NTFS

Wiederherstellbarkeit durch **Transaktionsprotokoll**; Individuelle **Zugriffsrechte**; Automatische und transparente **Kompression** und **Verschlüsselung**; **Proprietär**
- Boot-Sektor: Absolut notwendige Metadaten für Volume; Referenz auf MFT
- Datenbereich: Alles andere → MFT und Daten

Master File Table (MFT)

Selbst eine Datei → Vorreserviert → Erweiterungsfähig; Strukturiert das Volume in Dateien

File-Record (in MFT)

Record-Header: Infos zur Konsistenz; Feste Grösse; Used-Flag; Offset für Attribute
Liste von Attributen: Länge variabel
Endmarker: FFFF'FFFF

Attribute / Streams

Besitzen Typ und Namen
- **Attribut Header**: 16B; Typ, Resident, Flags, etc.
- **Resident Header**:(Offset und Länge) oder Non-Resident Header (Information zu Speicherort)
- **Attribut Value**
- **Typ**: $STANDARD_INFORMATION → Klassische Dateiattribute (Erstelldatum, Änderungsdatum)
- **Typ**: $FILE_NAME → Name, Referenz auf Parent; Mehr
- **Typ**: $DATA → Eigentliche Datei/Stream; <1KB → Resident; ADS; Non-Resident → Runlists

Runs / Runlists

Runs: Daten ausserhalb der MFT; Mehrere aufeinanderfolgende Cluster; Dynamische Länge
Runlist: Sequenz von Runs, 0-Terminiert; können fragmentiert sein

Alternate Data Streams (ADS)

$DATA:<name> → zusätzliche Daten; wird kaum benutzt, wenig Anwendungs-Support

Komprimierung

Sparse Files/Run ohne Adresse/Offset → aufeinanderfolgende Nullen
Ansonsten LZ77, falls > 1 Cluster eingespart

Verzeichnisse

Wie Dateien, $DATA-Attribut durch Index ersetzt; B+-Baum → sortierter Inhalt

Hard-/Softlinks

Hard-Link: Mehrere $FILE_NAME-Attribute für selbe Datei; Änderung am Link kann Datei beeinflussen; nur Datei ohne Links wird gelöscht
Soft-/Symbolic-Link: Datei enthält Pfad auf andere Datei → keine Löscheinschränkung

EXT Filesystem

Ext: Erstes Filesystem → Erweiterung von Minix-FS;
Ext2: Kommerzielle Qualität, nicht Ext4 kompatibel;
Ext3: Journaling → **Ext4**: Verbesserte Geschwindigkeit und Zuverlässigkeit

Block (vgl. Cluster)

Gesamtes Volume ist in Blöcke aufgeteilt; Speicher als Blöcke alloziiert (Daten einer einzigen Datei pro Block)
Logische Block#: Relativ, Applikationssicht
Physische Block#: Nummer auf Volume

Inode (Index Node)

Beschreibung der Datei (Metadaten), **ausser Name und Pfad**; Referenz auf Datenblöcke (Direkt oder 1/2/3-fach Indirekt); Alle Inodes aller Blockgruppen gelten als eine grosse Tabelle
Indireke Blöcke referenzieren Blöcke, 32-Bit Nr
File-Holes: Block aus nullen → Inode-Referenz auf 0 (vgl. NTFS Sparse File)
Neue Inodes möglichst nahe am Verzeichnis-Inode platziert; Verzeihcnis-Inode in möglichst freie blockgruppen

Verzeichnis

Wie Inode, dessen Datenbereich Dateieinträge (Inode, Dateiname, Typ) enthält; „.“ und „..“ werden automatisch hinzugefügt; Root ist Inode# 2

Hard-/Soft-Link

Hard-Link: Gleicher Inode, verschiedene Pfade
Soft-Link: Datei enthält Pfad

Blockgruppe

Unterteilung im Volume, umfasst mehrere Blöcke; Enthalten Meta-Informationen (Block Usage, Inode Usage, etc); Kopie des Superblock

Superblock

Enthalten im ersten Block der Blockgruppeo; Meta-Informationen über Volume (Blockgrössen, Gruppengrössen, Statusbytes) Startet immer ab Byte 1024
Sparse Superblocks: Reduktion der Kopien auf reine Potenzen von 3, 5 oder 7; genügend Sicher

Ext4

Journaling

Es müssen nur Daten im Journal überprüft werden; Reservierte Datei aus sehr grossen Extents; Journal enthält Transaktionen → Einzelschritte die das FS vornehmen soll; sehr schnell schreibbar
- Journal: Metadaten und Dateiinhalte im Journal; maximale Datensicherheit; Grosse Geschwindigkeitseinbussen → doppelte Op.
- Ordered: Metadaten im Journal; Dateiinhalte werden direkt vor Commit geschrieben;
- Writeback: Metadaten ins Journal; beliebige Reihenfolge beim Schreiben und Commit; Schnell, aber Datenmüll in Dateien möglich

Extent Tree

Ähnlich wie NTFS Runs, aber als Baum; Header, index (Nodes) und Extent(Leaves); Extent verweist auf fortlaufende Anzahl Blöcke
Index-Knoten: Index-Eintrag und Index-Block
Index-Eintrag: Physische Nummer Index-Block; logisch kleinste Blocknummer aller Kinder
Index-Block: Referenz auf Kind-Knoten (Index-Knoten oder Extents)

Vergleich FAT, NTFS, EXT2

FAT-Verzeichnisse enthalten alle Daten über Datei → einziges Verzeichnis, keine Hard-Links
Ext2 Dateien werden durch Inodes beschrieben, Link zu Inode von Verzeichnis, nicht von Datei zu Verzeichnis
NTFS Dateien werden durch File-Records beschrieben, Verzeichnis enthält Namen und Link auf Datei; Link zu Name und Verzeichnis im Attribut → Hard-Links

Programme

POSIX

fork: erzeugt neuen Prozess/Duplikat (andere Metadaten); Parent erhält Child-Id, Child erhält 0
exec: ersetzt Prozess durch Executable; Start am Entry-Point; mehrere Versionen: v→ vector; l→ list; p→ PATH, e→ environment
Prozess-Baum: Prozess ist nachkomme des ersten Prozess (init oder systemd bei Linux)
Prozess normal beenden: **return** x oder **exit**(x)
Prozess abnormal beenden: **abort**
atexit: Cleanup-Funktion callback; wird beim Beenden aufgerufen in umgekehrter Reihenfolge
_Exit: freigabe Ressourcen, Transfer Child- und Zombie-Prozesse auf Systemprozess
waitpid: wartet auf beendigung; erhält x
Zombie: beendeter Prozess, welcher nicht abgewartet wurde; belegt weiterhin Ressourcen
fork, exec, exit, waitpid als **Wrapper um System-Calls**

C Toolchain

Präprozessor

Parst Datei, bis keine Ersetzungen mehr vorgenommen werden: Entfernt Kommentare, ersetzt makros, Verarbeitet Direktiven (z.B. #include)
Erzeugt reine c-Datei (**Translation-Unit**)

Compiler

Übersetzt Translation-Unit nach Assembler; Erstellt Abstract Syntax Tree; Optimiert; Erzeugt Assembly File (Text) mit referenzen auf externe Variablen oder Funktionen

Assembler

Übersetzt Assembly File in binäre Maschinensprache (Object File); Referenzen bleiben nicht aufgelöst; Format für Linker vorbereitet

Linker

- Verknüpft Object-Files miteinander → statische Bibliotheken
- Verknüpft Object-Files und statische Bibliotheken → Dynamische Bibliotheken

Executables und dyn. Bibliotheken müssen vollständig aufgelöst sein

Executable and Linking Format ELF

Loader lädt Executables und dynamische Bibliotheken in den hauptspeicher
Linking View: SHT, Sections
Execution View: PHT, Segments
ELF Header: Beschreibt Aufbau der Datei, Entry-Point, relative Referenz auf SHT/PHT
Program Header Table: Beschreibt Segmente, arbeitet mit virtuellen Adressen
Segments: Speicherbereiche, vom OS zur Laufzeit verwendet
Section Header Table: Beschreibt Sections; Unterschieldiche Typen
Sections: Von Linker verwendet; Fasst unterschiedliche Daten (z.B. String-Tabelle, Relokations-Informationen, Symbol-Tabelle)
String-Tabelle: typische Namen von Symbolen, keine String-Literale
Symbol-Tabelle: Einträge identifizieren Symbol
Segmente und Sektionen **überlappen sich** → Unterschiedliche Einteilungen für gleiche Speicherbereiche; Loader sieht nur Segmente

Bibliotheken

Statische Bibliotheken

Archive von Objekt-Dateien: Linker behandelt statische Bibliotheken wie mehrere Objekt-Dateien; Geringer Mehraufwand für Linker, dafür kein von OS und Compiler

Dynamische Bibliotheken

Werden **zur Ladezeit oder Laufzeit** des Programms gelinkt; Executable enthält Referenz auf Bibliothek; **Entkopplung des Lebenszyklus**; nur benötigte Bibliotheken laden; Erweiterung um Funktionalität ohne Anpassung der Executable; müssen verschiebbar sein; **Loader übernimmt** Aufgabe des Linkers
Geteilter Code → Bibliothek soll nicht mehrfach geladen werden → Position-Independent Code
Bieten mehr **Flexibilität**, aber auch **Overhead**.

Position-Independent Code

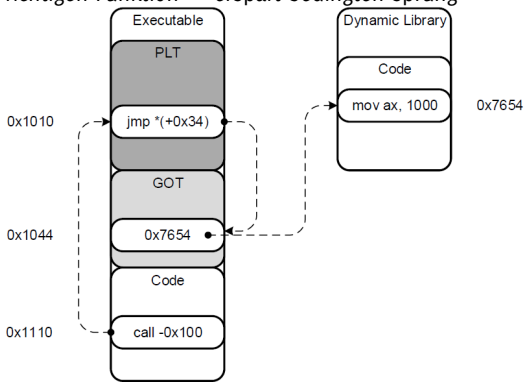
Hängt nicht von seiner Adresse ab → Adressen relativ zum Instruction-Pointer
Relative Move/Call Instruktionen in CPU: Emulatio bei x86_32 mit relativen Calls

Global Offset Table (GOT)

Einmal pro dynamischer Bibliothek und Executable → ein Eintrag pro fremdem Symbol → Linker/Loader setzt zur Laufzeit Adressen

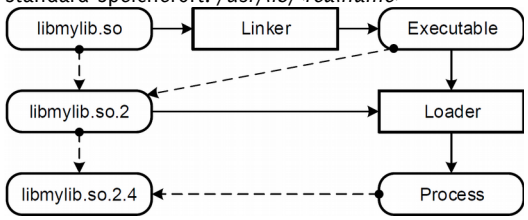
Procedure Linking Table (PLT)

Lazy-Binding für Funktionen; pro Funktion ein Eintrag
Proxy-Funktion: Ersetzt sich selbst in GOT mit Link zur richtigen Funktion → erspart bedingten Sprung



Shared Objects API

dlopen: öffnen einer dynamischen Bibliothek → Handle
dlsym: findet Adresse des Symbols in Handle
dlclose: schliesst Handle
dlerror: abfragen von Fehler
Shared Objects können von OS **automatisch geladen** werden, falls Dependency definiert
Linker name: „lib‹Bibliotheksname›.so“, beim Build verwendet → neuste Version
Soname: „‹linker name›.‹Version›“, in Executable → neuster Bugfix
Real name: „‹soname›.‹SubVersion›“
Standard-Speicherort: */usr/lib/‹realname›*



Statische Bibliothek erstellen

Normales kompilieren (gcc -c) → Zusammenfügen zu Archiv (ar)
Konvention: „lib‹Bibliotheksname›.a“

Dynamische Bibliothek erstellen

Kompilieren als Position-Independent Code (gcc -fPIC -c) → Erzeugen eines speziellen Image

```
gcc-shared-Wl,-soname,libmylib.so.2
-o libmylib.so.2.1 f1.o f2.o -lc
```

Verwenden von Bibliotheken

1. **Statische Bibliothek**: Suchpfad und Bibliothek
2. **Dynamische Bibliothek**, mit Programm laden: ohne Suchpfad → Standardsuchpfad
3. **Dynamische Bibliothek**, mit **dlopen**: Bibliothek für dynamisches Laden libdl.so

```
1# gcc main.c -o main -L. -lmylib
2# gcc main.c -o main -lmylib
3# gcc main.c -o main -ldl
```

Referenzen debuggen: readelf, ldd (erkennt indirekte Referenzen, Programm wird ausgeführt)
ld-linux.so: Lädt Dependencies nach, indirekt vom OS aufgerufen
Einbindzeiten: Link time / Load time / Run time

Anhang

2^0	1	2^5	32	2^{10}	1‘024	2^{15}	32‘768
2^1	2	2^6	64	2^{11}	2‘048	2^{16}	65‘536
2^2	4	2^7	128	2^{12}	4‘096	2^{17}	131‘072
2^3	8	2^8	256	2^{13}	8‘192	2^{18}	262‘144
2^4	16	2^9	512	2^{14}	16‘384	2^{19}	524‘288