Betriebsystem

„Be'triebs·sys·tem Programmbündel, das die Bedienung eines Computers ermöglicht.“ - Universalwörterbuch Rechtsschreibung

Prozesse

“Ein Prozess ist ein Programm zum Zeitpunkt der Ausführung”

Prozess-Kontext

- Codesegment (text)
  - Programmcode
- Datensegment (data)
  - Block-Storage-Segment - uninitialisierte Daten (globale Var.)
  - Datensegment - Vorinitialisierte Daten (globale Var.)
- Stapelsegment (stack)
  - Stack - Lokale Variablen
  - Heap - Dynamische Variablen
- Prozessorregisterinhalte
  - Instruktionszeiger
  - Stapelzeiger
  - Vielzweckregister
- Prozesszustand
- Benutzerkennung
- Zugriffsrechte
- Belegte Betriebsmittel (Datein, E/A-Gerälte, ...)
- ...

Unix-Prozesse

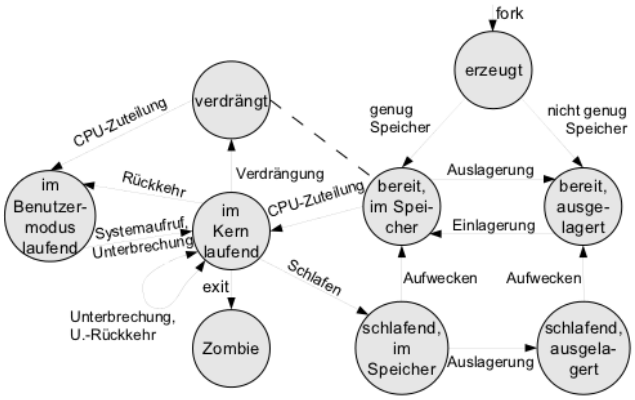


Abbildung 1: UNIX-Prozesszustände

- UNIX-Philosophie - Kurz: “Do one thing, do it well”

  - Write programs that do one thing and do it well.
  - Write programs to work together.
  - Write programs to handle text streams, because that is a universal interface.

- System startet mit dem swapper (PID 0) Prozess.
- Alle weiteren Prozesse werden durch fork erzeugt.
- Prozesse können Signale erhalten.
- Systemcalls:
  - **getpid**(2) liefert PID des laufenden Prozesses
  - **getppid**(2) liefert PID des Elternprozesses (PPID)
  - **getuid**(2) liefert die Benutzerkennung des laufenden Prozesses (UID)
  - **fork**(2) erzeugt neuen Kindprozess
    - Extrem teuer wegen Kontexerzeugung
    - Erzeugt Kind und kopiert:
      - Adressraum (code, stack, bss, data)
      - Benutzerkennung
      - Standard-E/A-Kanäle
      - Prozessgruppe, Signaltabelle
      - Offene Datein, Arbeitsverzeichnis
    - Heute: **copy-on-write** (Erzeuge erst den neuen Kontex wenn daten verändert werden.)
  - **vfork**(2) erzeugt leichtgewichtigen Kindprozess
    - (Hist.) Lösung um kosten bei Kinderzeugung zu reduzieren.
    - Parent suspendiert bis das Kind **exec**.. oder _exit() aufruft
    - Benutzt Parent-kontext bis dorthin.
    - Darf keine keine Daten verändern.
    - Heute: copy-on-write⇒**vfork** überflüssig
  - **exit**(3), **_exit**(2) beendet den lSIGCHLD - Kindprozessaufenden Prozess
    - Beim exit erbt der init-Prozess alle Kinder des terminierenden.
    - Terminierte Prozesse die vom Vater nicht weggeräumt wurden sind im Status **Zombie**
    - exit ruft die aufräumfunktionen auf und terminiert dann.
    - _exit tötet sofort, Kinder gehen an den init-Prozess und sendet SIGCHLD an Vater.
  - **wait**(2) wartet auf die Beendigung eines Kindprozesses
    - Räumt Zombies weg
  - **execve**(2) lädt und startet ein Programm im Kontext des laufenden Prozesses
  - **signal**(2) Überschreibt den Standartsignalhändler. Bsp Signale:

    SIGINT Prozess abbrechen

    SIGSTOP Prozess anhalten

    SIGWINCH Fenstergröße verändert

    SIGCHLD Kindprozess

    SIGSEGV Speicherschutzverletzung

    SIGKILL Prozess wird getötet

- Gewicht von Prozessen
  - Meint Bildlich die größe des Kontext
  - Klassische UNIX-Prozesse schwergewichtig
  - **Leichtgewichtige** Prozesse (**Thread**s)
    - 1:1-Beziehung zwischen Addressraum und Kontrollfluss aufgehoben
    - Thread teilen sich den Adresraum bis auf den Stack!
    - Profitieren von Multiprozessor-Systemen
    - In **Windows NT** von integraler Bestandteil(Der Scheduler verwaltet nur Threads!)
    - **Linux** implementiert POSIX Threads und unterscheidet der Scheduler nicht zwischen Prozess und Threads (alles sind **Tasks**)
  - **Federgewichtige** Prozesse (**Userlevel Threads**)
    - Über Bibliotheken realisiert
    - Scheduler weiß nichts davon
    - Kontextwechsel billig!
    - Ein Thread kann alles blockieren!
    - In **Windows NT** fibers genannt.

Scheduling

Abfertigungszustände

- Kategorisiert in short-, medium-, long-term scheduling
- **short-term scheduling** (kurzfristig)
  - bereit (**ready**)
  - laufend (**running**)
  - blockiert (**blocked**/waiting)
- **medium-term scheduling** (mittelfristig)
  - ausgelagert bereit (**ready suspend**)
    - Überführung in ready wird **swap-in** genannt.
  - ausgelagert blockiert (**blocked suspend**)
    - überführung in blocked suspend wird **swap-out** genannt.
- **long-term scheduling** (langfristig)
  - erzeugt (**new**)
  - beendet (**exit**)

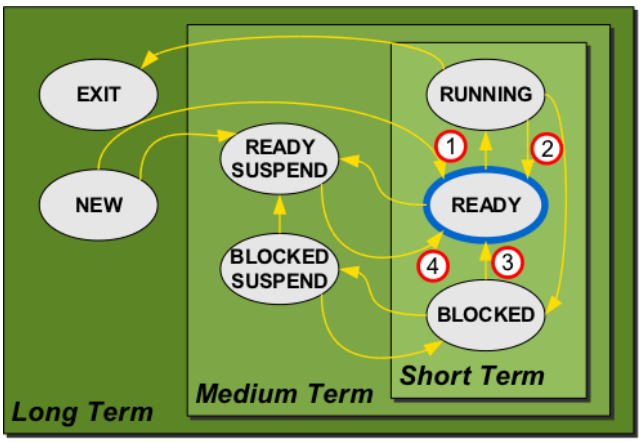


Abbildung 2: Abfertigungszustände

Einplanung (scheduling) bzw. Umplanung (rescheduling)

1. Prozesserzeugung
2. Prozess gibt kontrolle ab.
3. Erwartet Ereignis tritt ein.
4. Wiederaufnehmen eines ausgelagerten Prozesses

Zuteilungsstrategien

- **Prioritäten**
  - In Kombination mit den anderen Verfahren
  - Statische Prioritäten
    - Zuweisung bei der Prozesserzeugung

  - Dynamische Prioritäten
    - Zuweisen vom Benutzer oder System (Nicewerte?)
    - SPN, SRTF, HRRN, FB sind spezielfälle hiervon.

- FCFS - **Fist come - First served**
  - Konvoi-Effekt kann auftreten
    - hohe Antwortzeit
    - niedriger E/A-Durchsatz
  - **nicht präemtiv**
  - **kein Verhungern**
  - **nicht Vorhersagebasiert**

- RR - **Round Robin**
  - Jeder Prozess erhält Zeitscheibe (**timeslicing)**
  - Benachteiligt E/A-lastige Prozesse (effektive CPU zeit geringer da sie ihre Scheibe nie aufbrauchen)
  - **präemptiv** (Schedulingverfahren bei dem der Scheduler einen Prozess dazu drängt die CPU frei zu geben)
  - **kein Verhungern**
  - **nicht Vorhersagebasiert**

- VRR - **Virtual Round Robin**
  - Wie RR
  - Führe Vorzugsliste für Prozesse die ihre Scheiben nicht aufbrauchen.
  - Scheiben in den Listen unterschiedlich lang.
  - **präemptiv**
  - **kein Verhungern**
  - **nicht Vorhersagebasiert**

- SPN - **Shortes Process Next**
  - Gefahr: Aushungern von CPU-Lastigen Prozessen (**starvation**)
  - präemptiv
  - Stapelbetrieb: Programmier gibt erwartete dauer an.
  - Dialogbetrieb: Mittelwert über bisherige CPU-Stöße
    - Alle CPU-Stöße werden gleich gewichtet.

$$S_{n+1} = \frac{1}{n}\sum_{i=1}^{n} T_i = \frac{1}{n} \cdot T_n + \frac{n-1}{n} \cdot S_n$$

    - Problematischer, Daher Mittelwert mit expontieller Glätter über den Gewichtsfaktor $0 < \alpha < 1$

$$\begin{aligned} S_{n+1} = & \quad \alpha \cdot T_n + (1-\alpha) \cdot S_n \\ S_{n+1} = & \quad \alpha \cdot \textstyle\sum_{i=0}^{n-1} (1-\alpha)^i T_{n-i} + (1-\alpha)^n S_1 \end{aligned}$$

- SRTF - **Shortes Remaing Time Frist**
  - Verdränge Prozess wenn $T_{erwartet} < T_{rest}$
  - **starvation gefahr.**

  - **präemtiv**
  - **Vorhersagebasiert**

- HRRN - **Highest Response Ratio Next**
  - Abwandlung von SRTF
  - Wähle Prozess mit höchsten R-Wert:
$$R = \frac{w + s}{s}$$
  - w = "Wartezeit des Prozesses", s = "erwartete Laufdauer"
  - **nicht präemtiv**
  - **kein Verhungern**
  - **Vorhersagebasiert**

- FB - **Feedback**
  - Mehre Listen mit unterschiedlicher Priorität
  - Bestrafen (**penalization**) von Langläufern (verschieben auf Liste niederer Priorität)
  - **anti-aging** nach wartezeit
  - unterste ebene nach **RR**

Es gibt kein bestes Verfahren, der Anwendungsfall bestimmt die wahl des Schedulingverfahrens.

Strategie	präemptiv/ kooperativ	Vorhersage nötig	Implement.-aufwand	Verhungern möglich	Auswirkung auf Prozesse
FCFS	kooperativ	nein	minimal	nein	Konvoi-Effekt
RR	präemptiv (Zeitgeber)	nein	klein	nein	Fair, aber benachteiligt E/A-lastige Prozesse
SPN	kooperativ	ja	groß	ja	Benachteiligt CPU-lastige Prozesse
SRTF	präemptiv (bei Eingang)	ja	größer	ja	Benachteiligt CPU-lastige Prozesse
HRRN	kooperativ	ja	groß	nein	Gute Lastverteilung
FB	präemptiv (Zeitgeber)	nein	größer	ja	Bevorzugt u.U. E/A-lastige Prozesse

Abbildung 3: Qualitativer Vergleich der Schedulingverfahren

Synchronization

Race Condition Situation in der mehere Prozesse um gemeinsame Daten konkurrieren und diese manipulieren. Das Ergebnis ist nicht mehr vorhersagbar.

Synchronisation Koordination von Kooperation und Konkurrenz zwischen Prozessen. Dies sorgt für eine prozessübergreifende Sequentialität.

Kritischer Abschnitt Code-Abschnitt der auf Daten zugreift die zu einer Race Condition führen können.

Lösungsansätze zum schützen Kritischer Bereiche

- Schlossalgorithmen bieten keine lösung.
- Bäckerei-Algorithmus
  - Stellt nachweißbar sicher das der kritsche abschnitt schützt wird.
  - Implementierung aufwendig
  - acquire hat $O(n)$
- Unterbrechungen vermeiden
  - Muss von der Hardware unterstüzt werden.
  - Wird der Prozess nicht unterbrochen kann es nicht kritisch werden.
  - Nicht im Usermode verwendbar
- Atomare Operationen
  - Hardwareunterstütung notwendig
  - Würde Schlossalgorithmen brauchbar machen.
  - Aber aktives warten mit Scheduling schlecht vereinbar.
- Semaphor
  - **p-Operation (-1)** (down oder wait)
  - **v-Operation (+1)** (up oder signal)
  - wird durch Betriebsystem bereitgestellt.
- Monitor
  - Feature von Programmiersprachen (concurrent Pascal, PL/I, Java,...)
  - **wait-Operation** blockiert Prozess bis ein signal kommt
  - **signal-Operation** (Notify in Java) deblockiert einen (oder alle) blockierten Prozess(e).
  - Teuer weil viele Kontextwechsel
  - Aber ziemlich sicher.

Verklemmungen

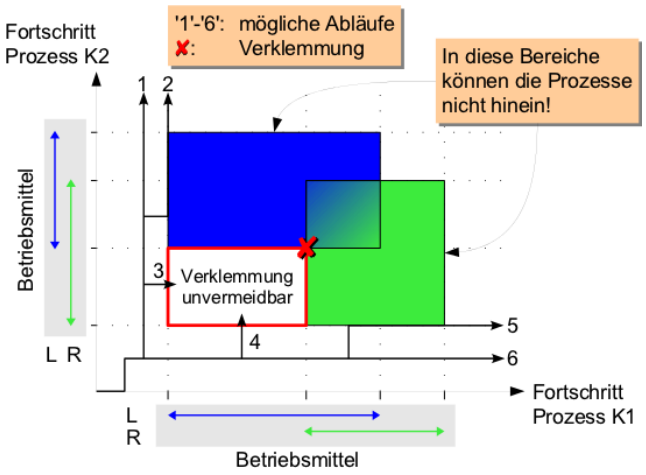


Abbildung 4: Verklemmung

Verklemmung "ein Zustand, in dem die beteiligten Prozesse wechselseitig auf den Eintritt von Bedingungen warten, dir nur durch Prozesse in dieser Gruppe selbst hesgestellt werden können."

Deadlock Passives Warten, Prozesse im Zustand **blocked**.

Livelock Aktives Warten, Prozesse im Zustand **ready** oder **running**.

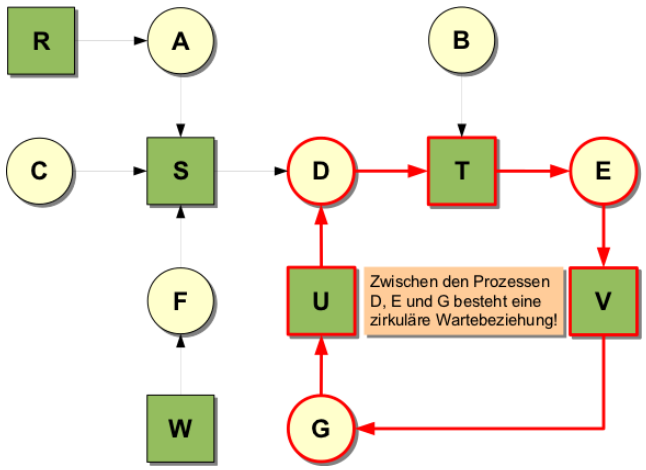


Abbildung 5: Betriebmittelbelegungsgraph mit Verklemmung. Quadarte sind Betriebsmittel, Kreise Prozesse.

Bedingungen

1. **mutal exclusion** - Exklusive Belegung von unteilbaren Betriebsmitteln.
2. **hold and wait** - Betriebsmittel nur schrittweise belegbar.
3. **no preemption** - Betriebsmittel nicht rückforderbar.
4. **circular wait** - geschlossene Kette wechselseitiger wartender Prozesse

Deadlock prevention (Verklemmungsvorbeugung)

- Methode greift zur Implementierungszeit
- indirekt Methoden (entkräften von 1.-3.)

1. nich-blockierende Verfahren verwenden.
2. Betriebsmittelanforderung unteilbar (**atomar**) auslegen.
3. Betriebmittel **Virtualisieren**
   - Virtueller Speicher, Virtuelle Geräte, virtuelle Prozssoren
   - Prozesse belegen nur **logische Betriebsmittel**
   - Die **physischen Betriebsmittel** können ohne Wissen entzogen werden. (3. wird entkräftet)

- direkte Mathoden (enkräften von 4.)

4. lineare/totale Ordnung von Betriebsmittel einführen

Deadlock avoidance (Verklemmungsvermeidung)

- Strategische Maßnahmes
- Durch Bedarfsanalyse zirkläres Warten ausschließen.
- System muss ständig auf "unsichere Zustände" prüfen.
- Prozess der ein unsicheren Zustand erzeugen würde **muss warten**.
- In der Praxis kaum eingesetzt.

Deadlock detection (Verklemmungserkennung)

- Methode Praktisch kaum eingesetzt.

1. Wartegraph und auf Zyklen untersuchen
   - Teuer da $O(n^2)$
   - Wenn zuu selten angewendet, liegen die Betriebsmittel brach.
   - Zu häufig $\Rightarrow$ Verschwendung von Ressourcen.
2. Verklemmungs auflösen
   - Gradwanderung zwischen Schaden und Aufwand
   - Prozess abbrechen
     - Schrittweise (auswahl schwierig) oder alle Prozesse auf einmal abbrechen.
   - Betriebsmittel enziehen.
     - Prozess muss zurückgefahren werden (großer Auffwand)

Interprozesskommunikatio

Synchronisation

Synchroner Nachrichtenaustausch

- Empfänger blockiert bis nachrichtig vollständig eingetroffen
- Sender blockiert bis Ankuft bestätigt
- Meist komplett ohne Pufferung (**Rendezvous**)

Asychroner Nachrichtenaustausch

- Sender gibt nachricht an das Betriebssystem
- Empfänger kann schauen ob Nachrichten für ihn hinterlegt sind
- Blocken ist optional
- Nachricht wird intern gepuffert

Addressierung

(nachrichtenbasierter Kommunikation)

- Direkte Adressierung über PID oder Kommunikationsendpunkte (Port, Socket)
- Indirekte Adressierung über Kanäle, Mailboxen oder Messagepuffer
- Gruppenadressierung

  Unicast Nachricht genau an einen
  Multicast Nachrift an eine Auswahl
  Broadcast Nachricht für alle

Dervises

- Nachrichtenformat
  - Stromorientiert / Nachrichtenorientiert
  - Feste / Variable Länge
  - Getypt / Ungetypt
- Übertragung
  - Unidirektion / Bidirektional (halb-duplex o. voll-duplex)
  - zuverlässung / unzuverlässung
  - (Nicht) erhalt der Nachrichtenordnung

Lokale Interprizesskommunikation

UNIX Signale

- Sender (z.B. Betriebssystem) hinterlässt ein Signal
- Empfänger ignoriert das signal oder führt eine Behandlungsfunktion aus.
- Bearbeitung von Signal **erfolgt** beim übergang vom Kernel in den User Mode.

UNIX Pipes

- unidirektinal, gepuffert (feste größe), zuverlässig, stromorientiert, ordnung bleibt erhalten und blockierend(wenn voll(Schreiben) und wenn leer (Lesen))
- Unbeannte Pipes
  - Signatur: int pipe (int fdes[2])
  - fdes[0] lesen mit read.
  - fdes[1] schreiben mit write
- Benannte Pipes
  - Als Spezialdatein ins Dateisystem
  - Signatur: int mkfifo (<Dateiname>, mode_t mode)
  - Rest über normale dateinutzung

UNIX Message Queues

- Über einen *key* wird die Queue erzugt
  - Signatur: int msgget(key_t key, int msgflg);
  - Return: lokal gültiger *msqid*
- Hat eine Prozesslokale ID: *msqid*
- Ungerichtete M:N Kommunkation
- puffer(flexible größe), getypt(long-wert), filterbar nach typ
- Sender über *int msgsnd(msqid, msgp, size msgsz, msgflg);*
- Empfangen über *int msgrv(msqid, msgp, size msgsz, msgtype, msgflg);*

Rechnerübergreifende Interprozesskommunikation

Sockets

- Sind Standartisiert und in der Praxis überall implementiert.
- Hohe abstraktion vom Kommunikationsystem
- Sind allgmein Gepuffert und Bidirketional
- Näheres wird beschrieben durch Domäne, Typ und Protokoll
- Domäne, bestimmt Protokolle und Addressefamilien
  - UNIX Domain - Sind wie Pipes
  - Internet Domain - Internetprotokolle
    - TCP/IP Protokoll
      - sicher, strom- und verbingsorientiert
    - UDP/IP Protokoll
      - nachrichtenorientiert, verbindungslos, unsicher, reihenfolge nicht sichergestellt, feste größe
      - ⇒Datagramm-Protokoll
  - Appletalk, DECnet, ...

RPC - Remote Procedure Call

- Aufruf von Funktionen über Prozessgrenzenhinweg.
- Sehr hohe Abstraktion

Speicherverwaltung

Aufgaben

- Freie Speicherbereiche kenne, verwalten und sinnvoll vergeben
  - Platzierungsstrategie (**placement policy**) - Woher den benötigten Speicher nehmen?
- Ein- uns Auslagern von Prozessen
  - Ladestrategien (**fetch policy**) - Wann Inhalten aus dem Hintergrundspeicher laden.
  - Ersetzungstrategie (**replacement policy**) - Wann inhalte auslagern?
- Relokation von Programmbefehlen
- Hardware optimal nutzen

Freispeicherverwaltung

- Datenstruktur
  - Bitlisten
  - Verkette Liste
  - Verkette Liste im freien Speicher
- Verschmelzung von Lücken, nach der Speicherfreigabe

Platzierungsstragegien

- **Externer Verschnitt** - Außerhalb der zugeteilten Speicherbereich entstehen Speicherfragmente, die nicht mehr genutzt werden können
  - **First Fit** - Man nehme die erste lücke.
  - **Rotating First Fit / Next Fit** - Nächste Lücke von der letzten gefüllten aus gesehen.
  - **Best Fit** - Kleine passende Lücke
  - **Worst Fit** - größte passende Lücke
- **Interner Verschnitt** - Innerhalb der zugeteilten Speicherbereiche gibt es ungenutzten Speicher
  - **Buddy-Verfahren**
    - Speicher wird in dynamische Bereiche der größe 2^n eingeteilt
    - Beim freigeben verschmelzen diese wieder

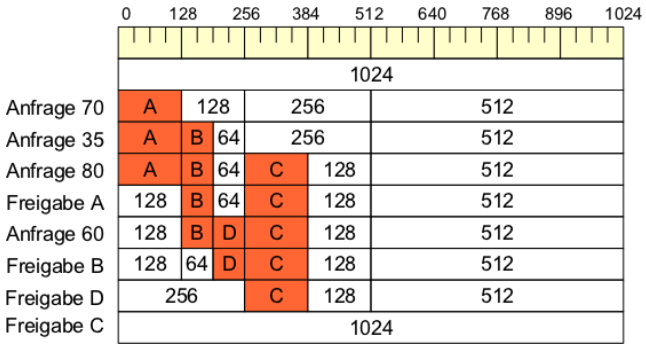


Abbildung 6: Beispiel Buddy-Verfahren

Swapping (Ein-/Auslagerung)

- Problem: Code benutz abs. Adressen ⇒ Programm kann nur wieder an die gleiche Speicherstelle geladen werden?
  - **Absolutes Binden** (Compile Time)
    - Adressen stehen fest
    - ⇒ Programm funktioniert nur an einer Speicherstelle.
  - **Statisches Binden** (Load Time)
    - Beim Laden werden die Adresse angepasst (**reloziert**)
    - ⇒ Binary muss Relokationsinformationen enthalten
  - **Dynamisches Binden** (Execution Time)
    - Code greift nur indirekt auf Operanden zu.
    - ⇒ Programm verschiebbar
- Alternative **keine** abs. Adressierung im Programm.
- oder Hardware unterstüzung ⇒ Adressmapping?

Adressabbildung

Segmentbasierte Adressabbildung

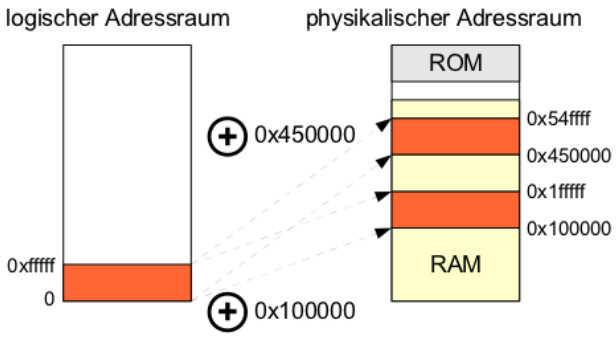


Abbildung 7: Segmentifzierung

- Hardwareunterstützte übersetzung von einem logischen Adressraum in physikalischen Adressraum
- Dat ding heißt **MMU** (Memory Management Unit)
- Kann leicht zu **externen Verschnitt** führen. Verschieben (**Kompaktifzierung**) ziemlich teuer.
- Realisiert mit Segmentabelletabelle

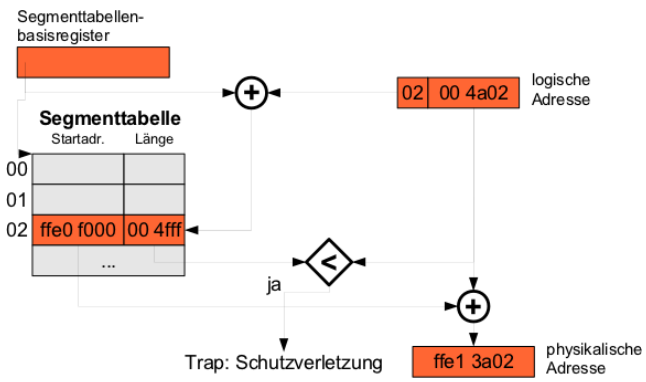


Abbildung 8: Berechnung der Physischen Adresse mit einer Segmentabelle

Seitenbasierte Adressabbildung (paging)

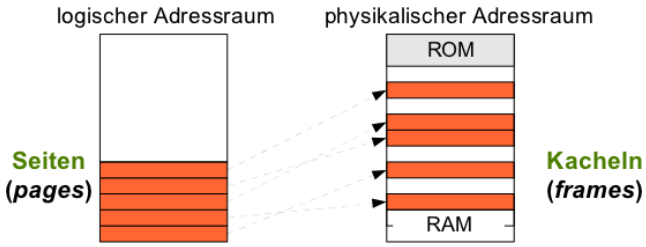


Abbildung 9: Paging

- Einteilen des logischen Adressraum in gleichgroße Seiten.
- logische Abschnitt heißen **pages** (Seiten), physiklaischen **frames** (Kacheln)
- Erzeugt **intern Verschnitt**, keine Fragmentierung, keine Kompaktifizierung nötig
- Tabelle im Speicher wird größer
- Mehr implizite Speicherzugriffe

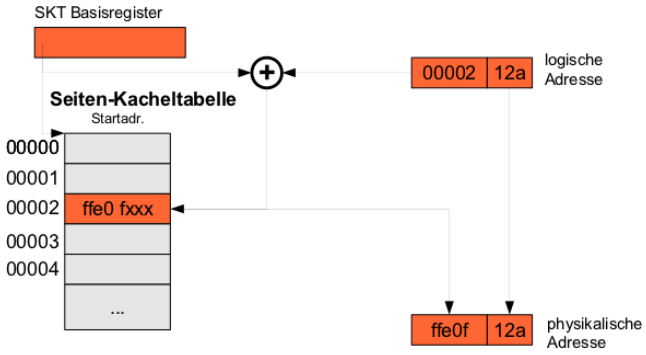


Abbildung 10: Berechnung der Physischen Adresse mit einer Seitentabelle

Segment- und Seitenbasierte Adressabbildung

- Verringert Speicherverbauch.
- Das Spiel kann zu einer mehrstufigen Seitenadressing ausgebaut werden.
  - Mehr implizite Speicherzugriffe

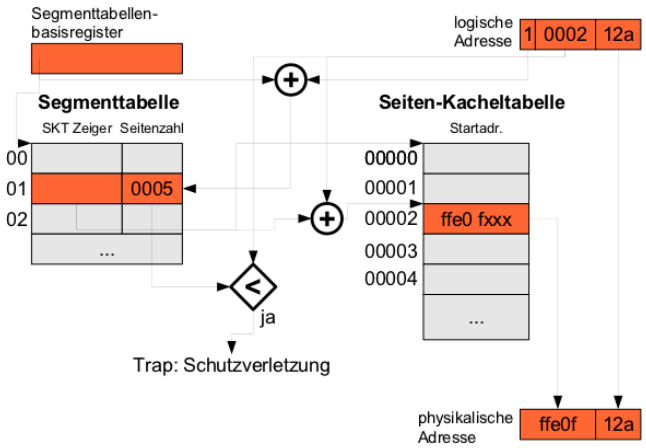


Abbildung 11: Berechnung der Physischen Adresse mit einer Seitentabelle

Swapping bei Adressabbildunge

- Präsenzbit in der SKT (Seitenkacheltabelle) speichert ob ein oder ausgelagert.
- Wenn ausgelagert: wird (Hardwareunterstützung nötig) Unterbrechung **(page fault)** ausgelößt und das Betriebsystem kann die Seite laden.

Translation Look-Aside Buffer (TLB)

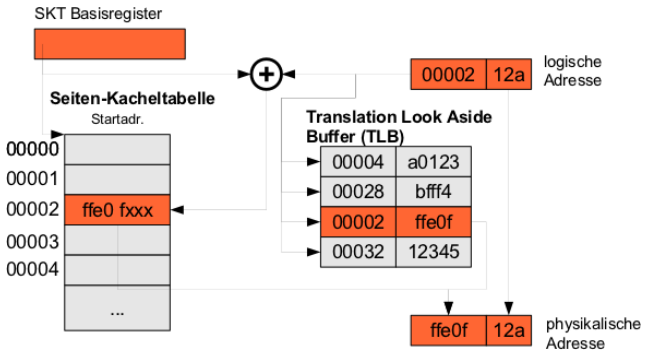


Abbildung 12: Tanslation Look-Aside Buffer(TLB)

- Ist ein schneller speicher der einen Auszug aus der SKT enthält.
- Bevor in der Großen tabelle gesucht wird, schaut man ob im TLB ein Eintrag vorliegt.
- Muss beim Kontextwechel gelöscht werden.
- Wird auf ein Eintrag zugeriffen der nicht im TLB ist so wird er dort eingetragen

Invertie Seiten-Kacheltabelle

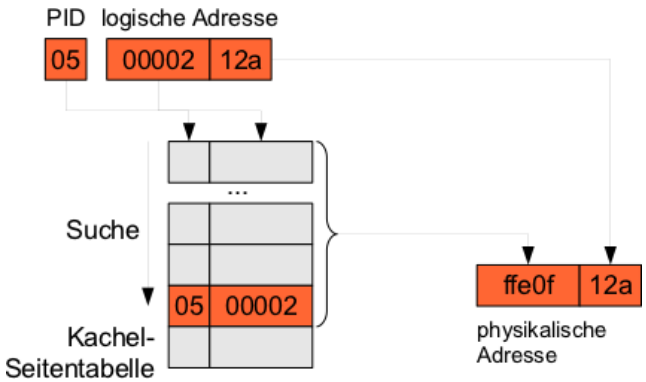


Abbildung 13: Adressberechung in der Invertierten Seiten-Kacheltabelle

- geringer Speicherverbauch ⇒ Tabelle muss nicht ausgelagert werden.
- *Sharing* schwer zu realisieren.
- Suche in der Tabelle aufwendig.
- Viel in 64Bit System eingesetzt.

Virtueller Speicher

Entkopplung des Speicherbedarf vom Verfügbaren Hauptspeicher

- Momentan nichtbenötiger Speicher kann solange anderweitig genutzt werden.
- So kann mehr Speicher verteilt werden als Physikalisch vorhanden.

Demand Paging

Meint das Ein- und Auslagern von Kacheln in den Hintergrundspeicher.

- Performance liegt ohne Seitenfehler bei 10ns bis 200ns
- Mit Seitenfehlerrate p bei etwa $100\text{ns} + 24999900\text{ns} \cdot p$
  - p sollte daher nahe Null sein.
- Seiten müssen bei Ein-/Ausgabeoperationen gesperrt werden.

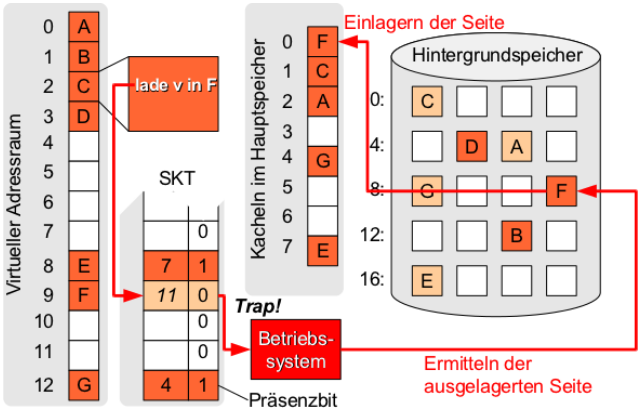


Abbildung 14: Vorgehen beim **page fault**

Demand Segmentation

- Zu grob Granuliert.
- Schwierigere Haupt- und Hintergrundspeicherverwaltung

Ersetzungsstrategien

First-In, First-Out (FIFO)

- Älteste Seite wird ersetzt. Dies muss gespeichert werden!
- Es kann zur FIFO-Anomaly kommen
  - Eine gerade Ausgelagerte Seite, wird wieder Eingelagert.

Optimate Strategie (OPT)

- "Ersetze immer die Seite mit dem größten Vorwärtsabstand!"
- Nicht Implementierbar, da es nicht (oder nur mit großen Aufwand) vorraussagtbar ist wann die Kachel nächste mal benutzt wird.

Least Recently Used (LRU)

- "Ersetzte die Seit emit dem größten Rückwärtsabstand!"
- Hat keine Anomalie
- Benötigt Hardwareunterstüzung
  - Benutze CPU-Zähler (für Speicherzugriffe)
  - Bei jeden Zugriff wird der Wert zum Seitendeskriptor geschrieben
- Aufwändig zu Implementieren
  - Speicherbedarf für Zählwert
  - Zusätzliche Speicherzugriffe
  - Minimumsuche bei Seitenfehler

Clock

- Es wird ein Ring benutzt. Neue seiten bekomme 1 als Referenzbit.
- Gesucht nach Seiten wird Reihum, dabei werden 1 zu 0, und die erste 0 wird ersetzt!

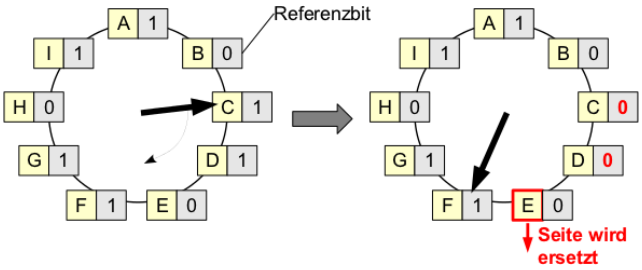


Abbildung 15: Vorgehen beim Clock

Ersetzungsstrategien

- Es wird immer eine Menge an Freien Seiten gehalten.
- Verringert die Ersetzungszeit auf die Einlagungszeit.
- Seiten die zu Auslagerung makiert sind aber noch nicht Ausgelagert wurden können sofort wiederbenutzt werden.

Seitenanforderung

Wieviel eingelagerte Seiten soll ein Prozess erlaubt werden.

- Maximal Anzahl der Kacheln
- Min abhängig von der Architektur
- Gleiche Zuordnungen
  - Anzahl der Prozesse legt Kachelmenge pro Prozess fest.
- Größenabhängige Zuordnung
  - Größe des Programms fließt ein.
- Lokales Anfordern von Seiten

  - Ein Prozess erstetzt immer nur seine Eigenen Seiten
  - Seitenfehler-Verhalten bestimmt der Prozesse nur selbst.
- Globales Anfordern von Seiten
  - Prozess ersetzt auch Seiten anderer Prozess
  - Effizienter. Ungenutzte Seiten anderer Prozesse können Problemlos verdrängt werden.
- Verbindung mit dem Scheduler
  - Inaktive Prozesse brauchen keine Seiten im Hauptspeicher

Thrashing (Seitenflattern)

Phanomän bei dem das beheben von page faults länger dauert als die Eigentlich Ausführung

Workingset (Arbeitsmege)

Wird über die letzten Nutzungen angenähert.

- Zugriffe pro Zeit pro Seite
  - Schwierig, muss gemessen und gespeichert werden.
  - Wäre über Interrupts zu realisieren. Ist Referenzbit gesetzt, so füge 1 zum Seitencounter hinzu.
  - Menge an Seiten über einen Wert Δ sind das Working Set.
- Bestimmung mit WorkSetClock

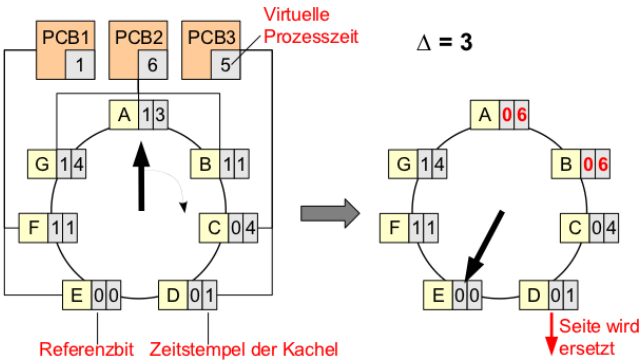


Abbildung 16: Beispiel für WorkSetClock

Eingabe und Ausgabe

- Zeichenorientierte Geräte
  - sequentieller Zugridd, selten position wahlfrei
  - Bsp: Tastatur, Drucker, Modem, Maus
- Blockorientierte Geräte
  - wahlfreier blockweise Zugriff
  - Bsp: Festplatte, Diskette, DVD, Bandlaufwerke
- Vieles passt nicht so wirklich in ein Schema
  - Graphikkarte
  - Netzwerkhardware
  - Zeitgeberbausteine

Ein-/Ausgabe Adressraum

1. Separater E/A-Adressraum
   - Es muss Spezielle Maschieneninstruktionen geben
2. Gemeiersamer Adressraum **(Memory-Mapped I/O)**
3. Hybride Architektur (1&2)

E/A-Handling

Polling ("Programmierte E/A")

- Bedeutet **aktives Warten** auf E/A

Unterbrechungsgetriebene E/A

- Prozess wird unterbrochen
- Die Hardware lößt eine Unterbrechung auf
- Das Betriebsystem weckt den prozess
- Gefahr des Unterbrechungsverlust, manchmal mehrstufig Behandlung im Betriebsystem
  - UNIX: Top Half, Botton Half
  - Linux: Tasklets
  - Windows: Deferred Procedures

DMA (Direkt Memory Access)

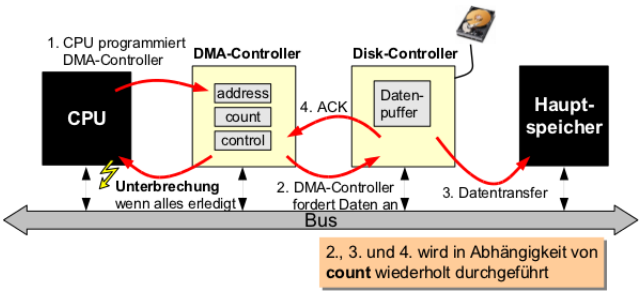


Abbildung 17: Durchführung eines DMA-Transfers

- Daten werden unabhänig von der CPU daten transferiert.
- Benötigt Datenpuffer in demen die Ergebnisse hinterlegt werden.
- DMA läuft an den Daten-Caches und den MMU (damit auch Speicherschutz) **vorbei!**
- Daher sollten DMA-Controller nicht aus dem User-Mode programmiert werden.

Betriebsystem Aufgaben

Zusammenfassung

- Geräteabstraktion bereitstellen
- Pufferung von Daten
- Ein-/Ausgabeprimitiven breitstellen
- Gerätesteuerung
- Ressourcenzuordnung verwalten
- Plug & Plays

Pufferung

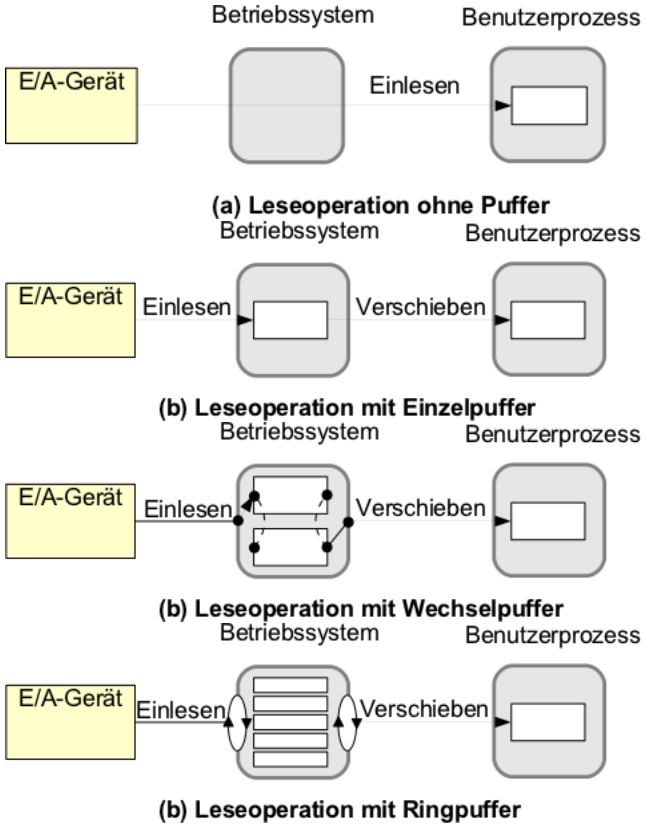


Abbildung 18: Übersicht Datepuffer

- Keine Pufferung
  - Daten die Eintreffen gehen verlohren, wenn kein *read* ausgefrüht wird.
  - Schreibvorgänge scheitern oder blockieren wenn *read* gerade arbeitet.
- Einzelpuffer

  - Es kann schon geschrieben werden obwohl das Gerät noch nicht bereit ist.
- Wechselpuffer
  - Es musst nicht mehr gewartet werden bis daten aus dem Puffer zum Gerät tranportiert wurden.
- Ringpuffer
  - Mehrfaches Lesen und Schreiben möglich bevor blockiert werden muss. Erlaubt das Vorrausschauende Laden.

E/A-Scheduling

- First In First Out (**FIFO**)
- Shortest Seek Time First (**SSTF**) - Es wird der Auftrag mit der kürzesten Positionierzeit vorgezogen
- Elevators (Fahrstuhlstrategie) - Mechanik in eine richtig bis es keine Aufträge mehr gibt.
- E/A-Scheduling ist wichtig. Wird aber immer schwieriger und Verliert an Bedeutung, da Hardware immer Intelligenter wird.

UNIX als Beispielsystem

- Repräsentation von Hardware als Spezialdatein
  - Blockorientierte Spezialdatien (**block devices**)
  - Zeichenorientierte Spezialdatein (**character devices**)
  - Jedes Gerät ist ein Tupel mit *(Gerätetyp*, *Major Number*, *Minor Number*)
    - Major Number: Wählt Treiber
    - Minor Number: Wählt Geräte im Treiber
- UNIX Zugriffprimtiven
  - int open(const char *devname, int flags)
    - „Öffnen“ eines Geräts. Liefert Dateideskriptor als Rückgabewert.
  - off_t lseek(int fd, off_t offset, int whence)
    - Positioniert den Schreib-/Lesezeiger – natürlich nur bei Geräten mit wahlfreiem Zugriff.
  - ssize_t read(int fd, void *buf, size_t count)
    - Einlesen von max. count Bytes in Puffer buf von Deskriptor fd.
  - ssize_t write(int fd, const void *buf, size_t count)
    - Schreiben von count Bytes aus Puffer buf auf Deskriptor fd
  - int close(int fd)
    - „Schließen“ eines Geräts. Dateideskriptor fd kann danach nicht mehr benutzt werden.
  - int ioctl(int d, int request, ...)
    - Erlaubt das ansprechen von spezielle Gerätefunktionen.
  - int select (int nfds, fd_set *readfds, fd_set *writefds, fd_set *errorfds, struct timeval *timeout)
    - Erlaubt das Warten auf mehrere Geräte. Returnt wenn alle Geräte breit sind. (Sprich nicht mehr blockieren)
    - nfds legt fest, bis zu welchem Dateideskriptor select wirken soll.
    - ...fds sind Dateideskriptoren, auf die gewartet werden soll:
      - readfds — bis etwas zum Lesen vorhanden ist

- writefds — bis man schreiben kann
- errorfds — bis ein Fehler aufgetreten ist

* Timeout legt fest, wann der Aufruf spätestens deblockiert.

Dateisysteme

- Stellen eine Logische sicht auf Hardware bereit.

Dateien

Kontinuierliche Speicherung

- Datei wird in Blöcke sequentiell hintereinader gespeichert.
- Vortei: Dateiposition leicht bestimmtbar. Zugriff auf alle Blöcke leicht.
- Nachteil: Vergrößern von Datein, sowie Finden von freienplatz der groß genug ist, ist schwierig.

Verkette Speicherung

- Im Kopf eines Block zeiger auf den nächsten Block der Datei.
- Zeiger kosten Speicher.
- Hohe Fehleranfälligkeit.
- Schlechter direkter zugriff auf Dateipositionen
- Bei **FAT** (File Allocation Table) wird ei Verkettung in einer redudanten Tabelle gepeichert.
  - Caching der FAT oder laden notwednig
- **Chunks**(auch Extends oder Clusters)
  - Datein in kontinuierlich gespeicherte Folgen von Blöcken.
  - Verbinden von Verketter und Kontinuierlicher Speicherung.
  - Zusätzliche Verwaltungsinformationen

Indiziertes Speichern

- Spezieller Block enthält alle Blocknummern einer Datei.s
- UNIX-Inodes sind Beispiel hierfür:
  - ein **Inode** enthält mehrere Zeiger. Ersten **9** sind direkt. die Letzten 3 sind ein bis dreifach indirekt.
  - Erlaubt große Datein
  - mehrere Blöcke müssen geladen werden.

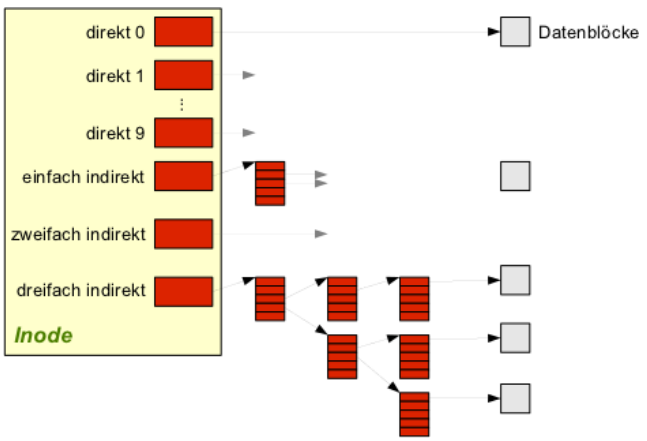


Abbildung 19: UNIX-Inodes Struktur

Baumsequentielle Speicherung

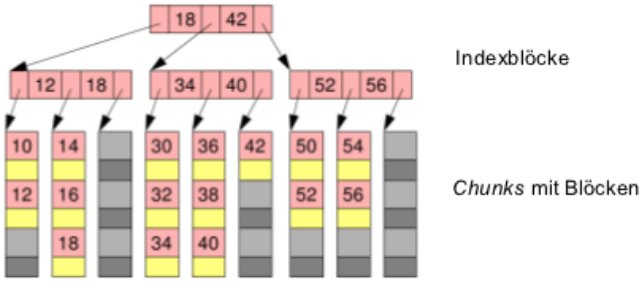


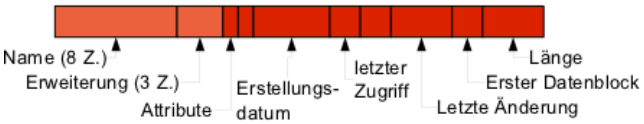
Abbildung 20: Baumsequentielle Speicherung

Freispeicherverwaltung

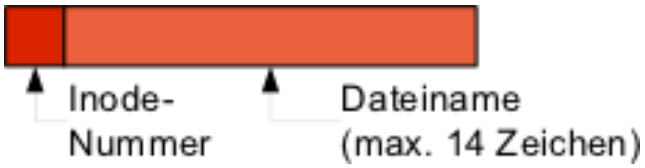
- Bitvektoren
- Vekette Liste
  - Vekettung oft in den Freien Blöcken
  - Verschmelzen ist recht einfach.
- Baumsequentielle Speicherung
  - Suche nach freien Blöcken extrem schnell

Verzeichnisse

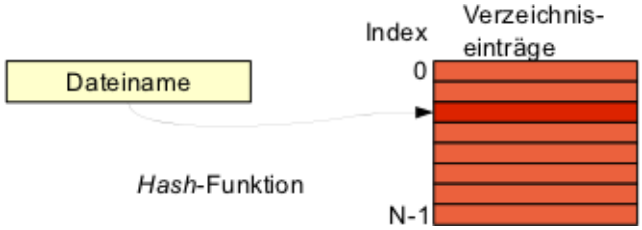
- Verzeichnis als Liste.
- Bei vFAT werden mehere Eintrage verwendet, um längere Dateinamen zu erlauben. Suche aufwendig.



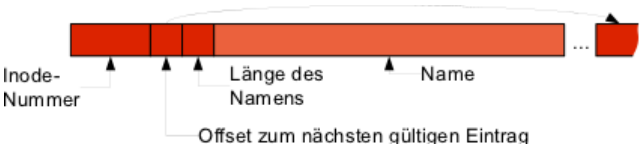
- Liste kann sortiert werden.⇒Suchen Schnell, Einfügen aufwendig.



- Hashing bildet Namen auf den Index in der Liste ab. Kollisionen können aber probleme bereiten.



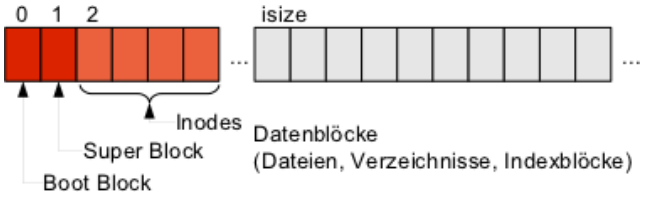
- Variable lange Listenelemte (4.2 BSD, System V Rel. 4, ...). Problematische ist der Umgang mit freien einträgen.



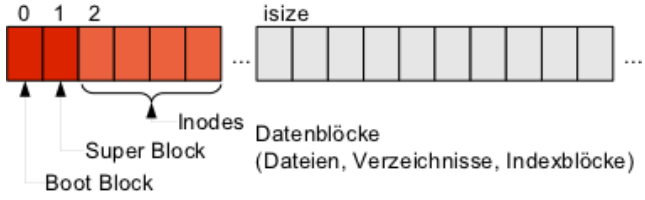
Dateisysteme

- **Boot Block** enthält Informationen zum Laden Betriebssystems
- **Super Block** enthält Verwaltungsinformation für ein Dateisystem
  - Anzahl der Blöcke, Anzahl der Inode
  - Anzahl und Liste freier Blöcke und freier Inodes
  - Attribute (z.B. Modified flag)
- **Zylinder** meint die Menge aller Spruen auf der Festplatte die übereinander liegen.

UNIX System V File System

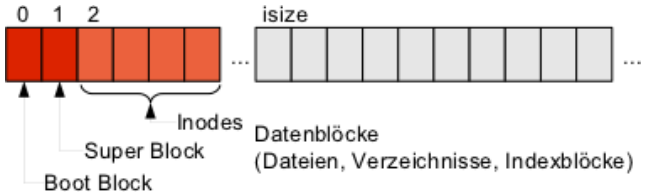


BSD 4.2 (Berkely Fast File System)



- Datei möglichst in einer Zylinder-Gruppe speichern.
- Redudanten Kopien der Superblocks in jeder Zylindergruppe.
- Vorteil von kürzeren Positionierungszeiten

Linux Ext2/3 File System



- Blockgruppen unabhängig von Zylindern

Block Buffer Cache

- Verwaltung und Algor. ähnlich Kachelverwaltung

Read ahead Beim Sequentiellen Lesen wird auch das Laden des Folgeblocks angestoßen.

Lazy write Es wird nicht sofort auf die Platte geschrieben.

Freiliste

- Kanidaten werden LRU Verfahren bestimmt.
- Freiliste ist **rot** in der Abbildung

Reclaim Bereits freie aber noch nicht benutzte Blöcke können reaktiviert werden.

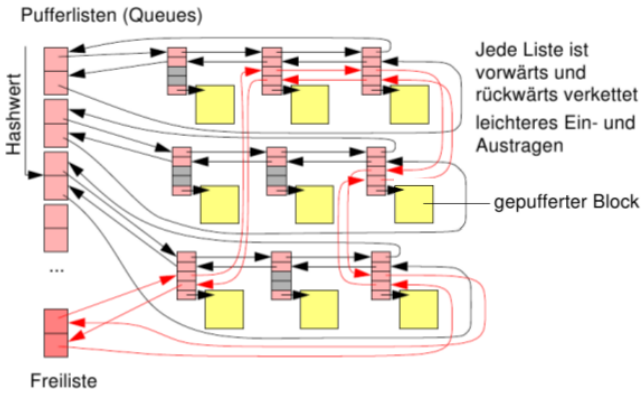


Abbildung 21: Block Buffer Cache mit Freiliste

Journaled File Systems

- Änderungen am Dateisystem werden als **Transaktionen** gesehen.
- Transaktionen werden in einem **Log File** gespeichert.
  - Der Protokolleintrag wird immer **vor** der eigentlich Änderung auf die Platte geschrieben.
- Beim Bootvorgang wird überprüft ob eine differenz zwischen Log und Dateisystem sind.
  - Transaktion abschlielien/wiederholen (**Redo**)
  - Angefange Transaktion rückgängig machen (**Undo**)
- Vorteil: Konsistenz des Dateisystem sichergestellt.
- Nachteil: Ineffizient, jede Operation braucht 2.
- Viel schneller als tools wie **chkdsk** bei großen Platten (Es müssen ja nur änderungen betrachtet werden)

Systemsicherheit

Safty Schutz Menschen vor Fehlern, Störungen, Ausfällen, bei Katastrophen

Security Schutz von Menschen und Rechnern von intendierten Fehler (Angriffen)

Rechteverwaltung

- TODO

Hardwarebasierter Schutz

Memory Management Unit

- Kontrolliert jeden Seitenzugriff über die Schutzbits
- Jeder Prozess sieht nur die Virtuellen Seiten
  - Sprich nur Virtuelle Addressen
  - Nur seien eigenen Speicherbereich
- Isolation des Restlichen physikalischen Speichers.

Schutzringe

- CPU-seitiger Schutz
- Jeder Ring hat nur einen eingeschränkten Befehlssatz

Softwarebesierter Schutz

- Identifikationsmechanismen
  - UNIX: Benutzerindentifikation, Gruppenidentifikation
    - Numerischer Wert
- Jede Ressource hat zugreornete Besitzer und Rechte
- Authentisierungsmechanismen
  - Unix login - Benutzername + Passwordabfrage
  - Hinterlegtes Passwort nicht für jeden sichtbar (shadow-Datei)
- Kryptographe Sicherung von Daten

Softwarefehler

Durchschnittlich einfehler pro 1000 Zeilen Code!

- Überläufe können Probeleme Verursachen