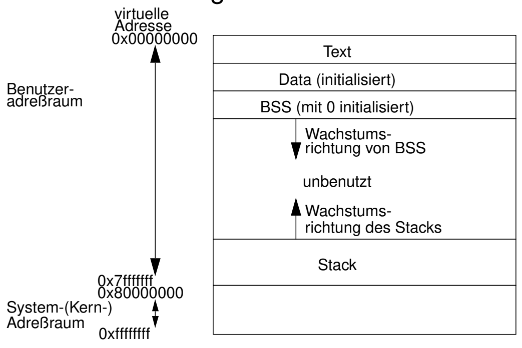
C Optimierung des Speicherzugriffs

C.1 Virtuelle Adressierung

1 Grundlagen

- Jeder Prozeß besitzt eigenen virtuellen Adreßraum

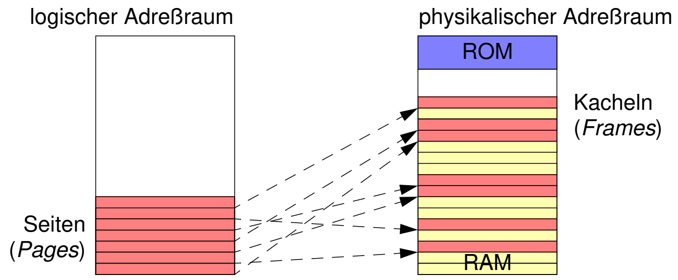


1 Grundlagen der virtuellen Adressierung (2)

- Ziel: Entkoppelung des Speicherbedarfs vom verfügbaren Hauptspeicher
  - ◆ Prozesse benötigen nicht alle Speicherstellen gleich häufig
    - bestimmte Befehle werden selten oder gar nicht benutzt (z.B. Fehlerbehandlungen)
    - bestimmte Datenstrukturen werden nicht voll belegt
  - ◆ Prozesse benötigen evtl. mehr Speicher als Hauptspeicher vorhanden
- Idee:
  - ◆ Vortäuschen eines großen Hauptspeichers
  - ◆ Einblenden benötiger Speicherbereiche
  - ◆ Abfangen von Zugriffen auf nicht-eingeblendete Bereiche
  - ◆ Bereitstellen der benötigen Bereiche auf Anforderung
  - ◆ Auslagern nicht-benötigter Bereiche

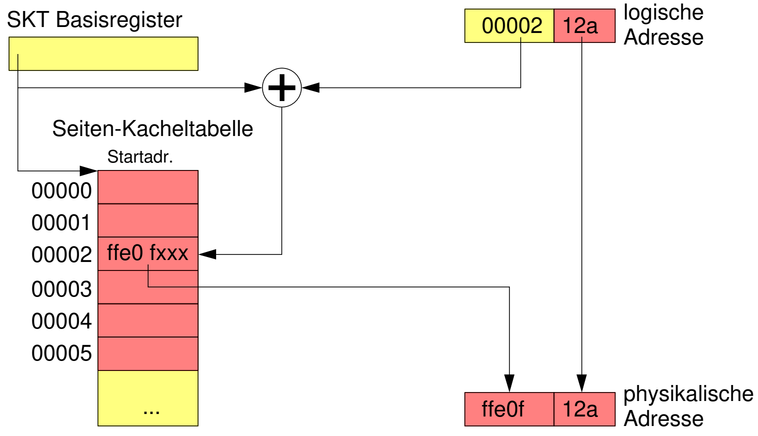
1 Grundlagen der virtuellen Adressierung (3)

- Einteilung des logischen Adreßraums in gleichgroße Seiten, die an beliebigen Stellen im physikalischen Adreßraum liegen können
  - ◆ Lösung des Fragmentierungsproblem
  - ◆ keine Kompaktifizierung mehr nötig
  - ◆ Vereinfacht Speicherbelegung und Ein-, Auslagerungen



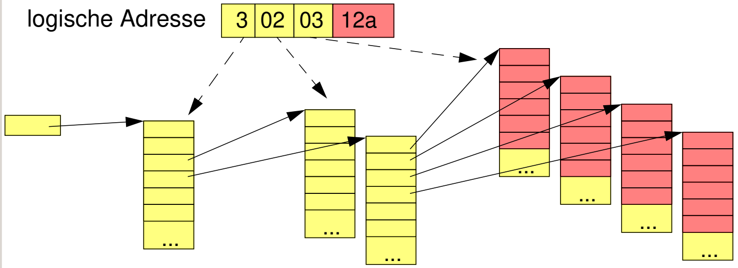
1 Grundlagen der virtuellen Adressierung (4)

- Tabelle setzt Seiten in Kacheln um

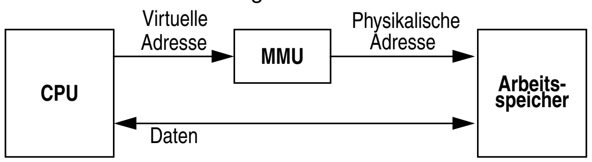


1 Grundlagen der virtuellen Adressierung (5)

- Mehrstufige Seitenadressierung



- MMU führt Adreßabbildung in Hardware durch

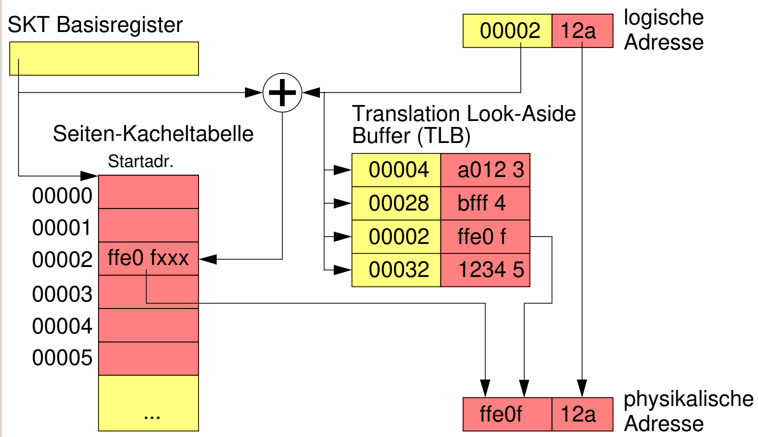


1 Grundlagen der virtuellen Adressierung (6)

- ▲ Seitenadressierung erzeugt internen Verschnitt
  - ◆ letzte Seite eventuell nicht vollständig genutzt
- Seitengröße
  - ◆ kleine Seiten verringern internen Verschnitt, vergrößern aber die Seiten-Kacheltabelle (und umgekehrt)
  - ◆ übliche Größen: 512 Bytes — 8192 Bytes (8192 Bytes bei UltraSPARC)
  - ◆ Systemaufruf: `getpagesize()`
- ▲ große Tabelle, die im Speicher gehalten werden muß
- ▲ viele implizite Speicherzugriffe nötig

2 Translation Look-Aside Buffer TLB

- Schneller Registersatz wird konsultiert bevor auf die SKT zugegriffen wird



2 Translation Look-Aside Buffer (2)

- Schneller Zugriff (< 1 Taktzyklus) auf Seitenabbildung, falls Information im voll-assoziativen Speicher des TLB
  - ◆ keine impliziten Speicherzugriffe nötig
- Bei Zugriffen auf eine nicht im TLB enthaltene Seite wird die entsprechende Zugriffsinformation in den TLB eingetragen
  - ◆ Ein alter Eintrag muß zur Ersetzung ausgesucht werden
    - Random
    - Round Robin
- TLB Größe
  - ◆ Pentium: Daten TLB = 64, Code TLB = 32, Seitengröße 4K
  - ◆ Sparc V9: Daten TLB = 64, Code TLB = 64, Seitengröße 8K
  - ◆ Größere TLBs bei den üblichen Taktraten zur Zeit nicht möglich

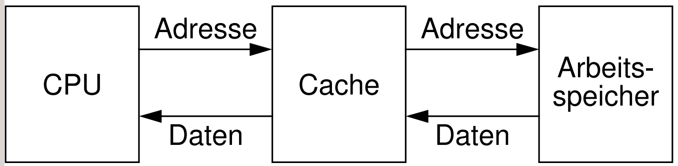
2 Translation Look-Aside Buffer (3)

- Der TLB kann nur einen kleinen Teil des Adreßraums abbilden.
  (z.B. UltraSPARC IIi 64 Einträge x 8 KB = 512 KB)
- Ein TLB-Zugriffsfehler wird in der Regel durch Software aufgelöst.
  (Durchsuchen der SKT und Einlagerung im TLB; < 32 Instruktionen).
  - ◆ Vorteil: Flexibles Design der SKT möglich
  - ◆ Nachteil: 1 Trap ist nötig
    (Overhead of precise exceptions in pipelined/superscalar CPUs)
  - ◆ Ausnahmen: Intel x86, teilw. Motorola/IBM PowerPC, HP PA-RISC)
- TLB-Zugriffsfehler erscheinen nicht in der OS-Statistik.
- 1 TLB-Zugriffsfehler = 2 Speicherzugriffe (bis zu 1000 Zyklen!)
- Schon eine Arbeitsmenge von 8KB kann den TLB überfordern.
  ➜ siehe Übung

C.2 Caches

1 Grundlagen

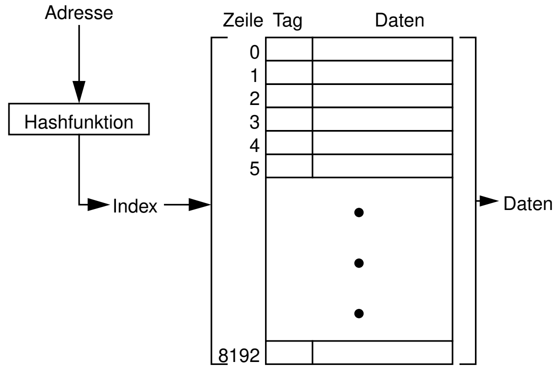
- Funktion



  - ◆ Pufferspeicher zur Ausnutzung der zeitlichen und örtlichen Lokalität
  - ◆ Zugriff auf gespeicherte Daten mit geringer Latenz und hoher Bandbreite
  - ◆ Reduktion der Zugriffe auf den Arbeitsspeicher (wichtig in Multiprozessen)
  - ◆ Puffer für asynchrone Prefetch-Operationen

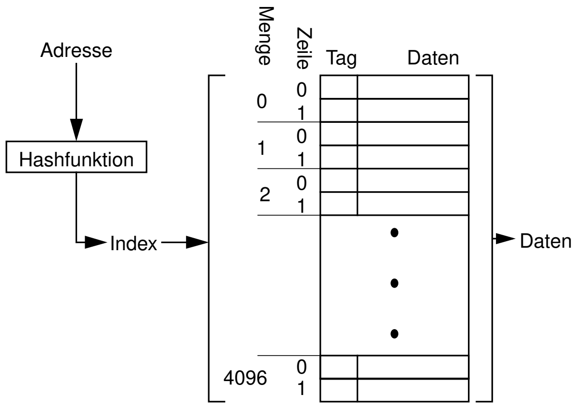
1 Grundlagen (2)

- Organisation
  - ◆ Aufbau aus Cache-Zeilen fester Länge (typischerweise 32/64 Bytes)
  - ◆ Abbildung der Addresse auf die Zeile (1):
    Beispiel: Direkt abgebildeter Cache (direct mapped cache)



1 Grundlagen (3)

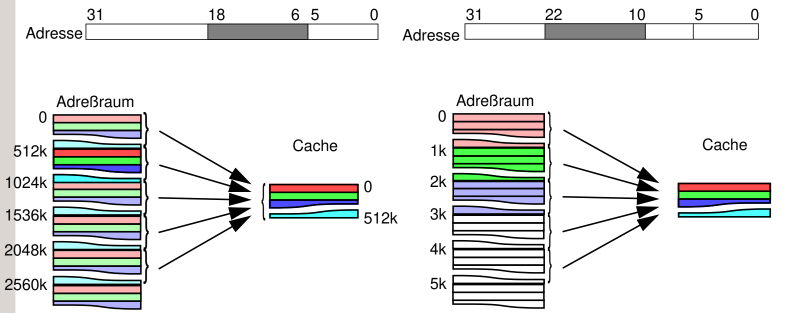
- ◆ Abbildung der Addresse auf die Zeile (2):
  Beispiel: Zweifach assoziativer Cache (two-way set associative cache)



1 Grundlagen (4)

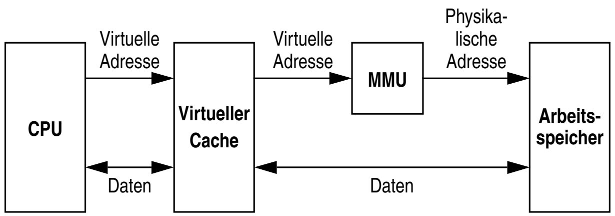
Hash-Algorithmen

- Modulo-Hashing, z. B. Bits 0 - 5 zur Auswahl des Bytes innerhalb einer Zeile, Bits 6 - 18 zur Auswahl der Zeile, Bits 19 - 31 sind Tag
- Hashing durch Ausschnitt aus der Adresse, z. B. Bits 10 - 22 als Zeilenadresse. Dieses Verfahren wird bei Caches jedoch nicht eingesetzt.



1 Grundlagen (5)

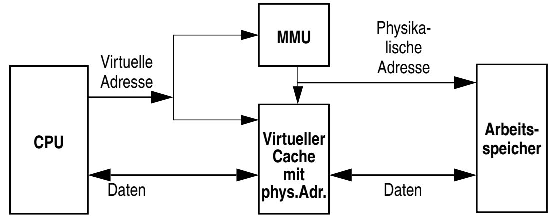
Virtuelle Caches



- Probleme:
  - Mehrdeutigkeit (ambiguity):
    Zwei identische virtuelle Adressen zeigen auf verschiedene Speicherbereiche
  - Bedeutungsgleichheit (alias):
    Verschiedene virtuelle Adressen zeigen auf den gleichen Speicherbereich
- Lösung: Systemsoftware kann Probleme mit viel Aufwand verhindern

1 Grundlagen (6)

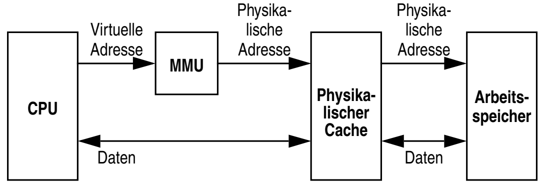
Virtuelle Caches mit physikalischen Adressen als Tags



- Vorteil:
  - Keine Mehrdeutigkeiten mehr
- Nachteile:
  - Bedeutungsgleichheiten noch möglich (➜OS Unterstützung nötig)
  - Suchgeschwindigkeit durch Umsetzgeschwindigkeit der MMU begrenzt
  - Problematischer Einsatz in Multiprozessoren mit gemeinsamem Speicher (Welche Zeile soll invalidiert werden?)

1 Grundlagen (7)

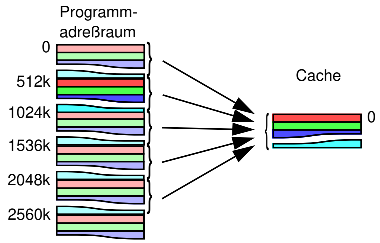
Physikalische Caches



- Vorteile
  - Absolute Transparenz aus Sicht des Prozessors
  - Keine performancekritische Systemunterstützung nötig
  - Multiprozessoren mit gemeinsamem Adreßraum können mit einem Cache-Kohärenzprotokoll in Hardware aufgebaut werden.
- Nachteil:
  - Bei jedem Zugriff ist Adreßumsetzung durch MMU erforderlich

2 Zugriffskonflikte

■ Virtueller Cache (z.B. PA-RISC 1.1 CPUs in Convex SPP 1600)



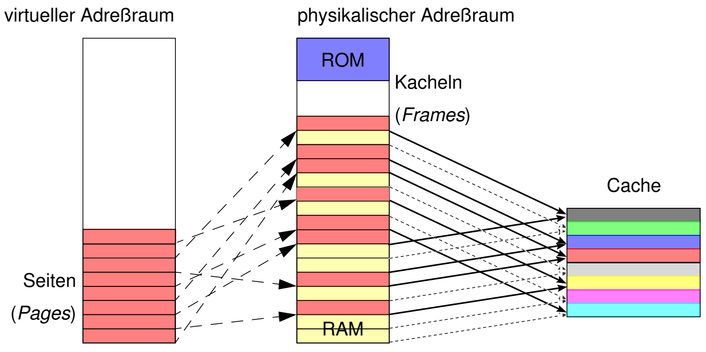
◆ Datenstrukturen, bei denen die Differenz der Adreßbereiche einem Vielfachen der Cachegröße enspricht, werden auf die selbe Cachezeile abgebildet.

◆ Virtuelle Caches mit physikalischen Tags werden häufig als first-level Caches eingesetzt (z.B. UltraSPARC II 16 KB direct mapped).
➜ siehe Übung

2 Zugriffskonflikte (2)

■ Physikalische Caches

◆ Seitenkonflikte durch zufällige Anforderung von physikalischem Speicher:



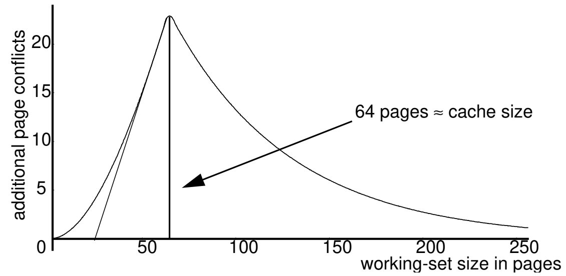
Zusammenhängender virtueller Speicher wird in der Regel auf beliebige freie physikalische Seiten abgebildet

2 Zugriffskonflikte (3)

■ Physikalische Caches (2)

◆ Auswirkungen des "*random page coloring*":
➜ Es kommt zu Konflikten beim Cache-Zugriff
➜ Nur ein Teil des Caches wird ausgenutzt
➜ Erhebliche Laufzeitschwankungen!

$$X(\text{p in bin}) = \binom{P}{p}\left(\frac{1}{C}\right)^{p}\left(1-\frac{1}{C}\right)^{P-p}$$



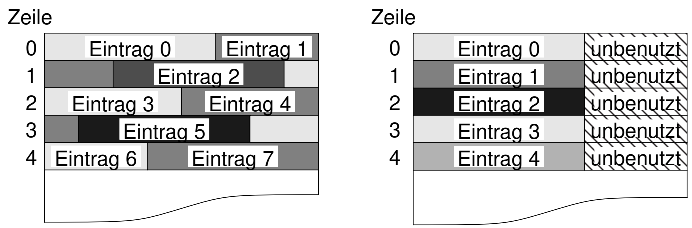
◆ Abhilfe: "Frischen" Speicher anfordern und im Speicher festhalten (mlock)

2 Zugriffskonflikte (4)

■ Einfluß der Zeilenlänge des Caches auf die Datenstrukturen

◆ Problem: Mehrfache Fehlzugriffe durch zeilenübergreifende 'non aligned' Datenstrukturen

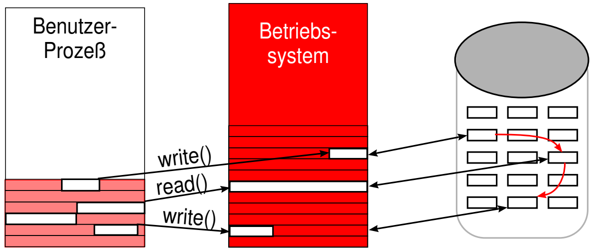
◆ Lösung: Auffüllen der Strukturen auf Vielfache der Zeilenlänge (Padding)



C.3 Ein/Ausgabe

1 "Klassisches" Lesen/Schreiben in Dateien

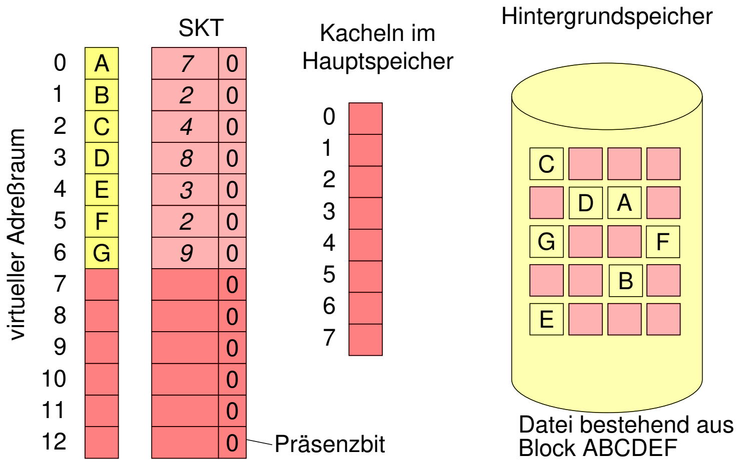
- open/seek/read/write Operationen
- Kopieren der Daten in den Kern
- Blockpufferung der Plattenblöcke im Betriebssystem



- Interessenkonflik zwischen Speicherverwaltung und Buffercache

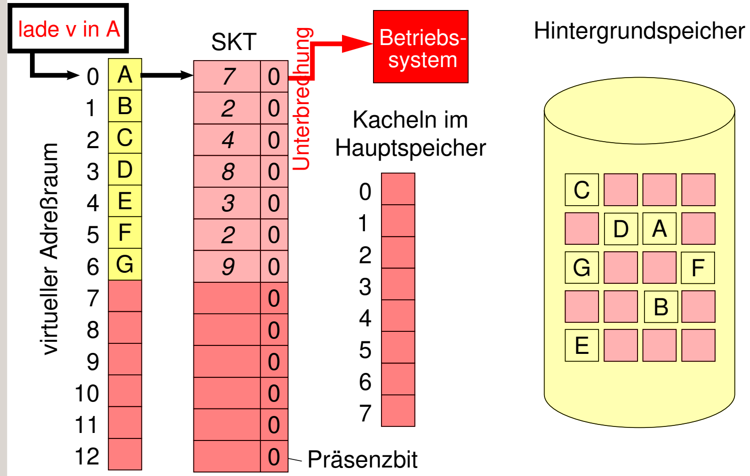
2 Speicherabgebildete Dateien

- Abbilden einer Datei auf virtuelle Seiten



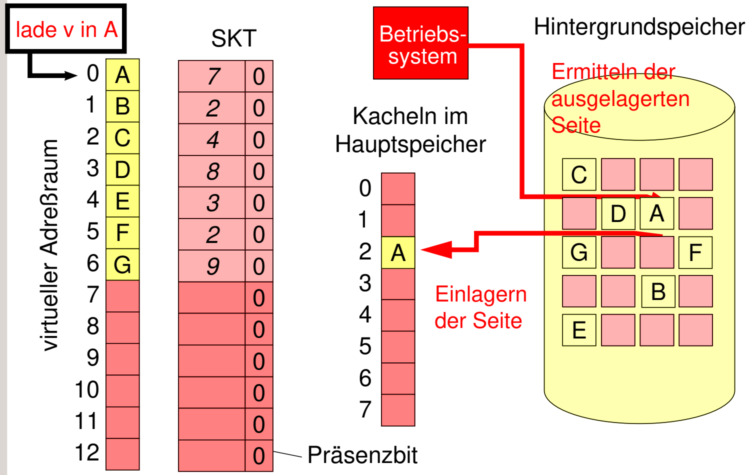
2 Speicherabgebildete Dateien (2)

- Reaktion auf Seitenfehler



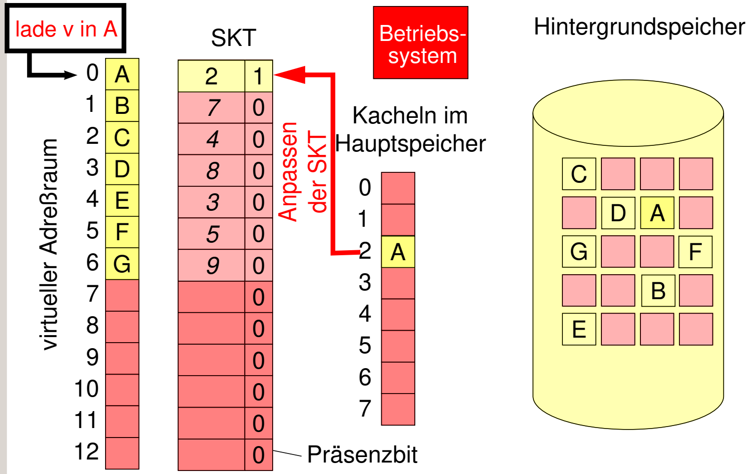
2 Speicherabgebildete Dateien (3)

- Einlagerung



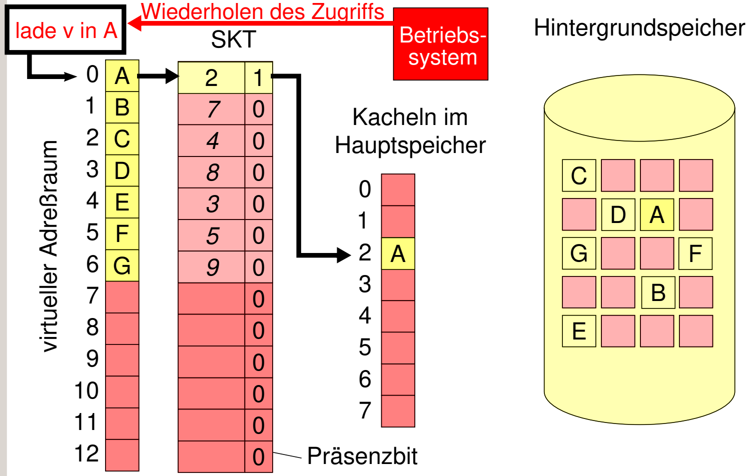
2 Speicherabgebildete Dateien (4)

- Einblenden der physikalischen Seite



2 Speicherabgebildete Dateien (5)

- Zugriff auf Seite



2 Speicherabgebildete Dateien (6)

- Aufrufe: `mmap()`, `munmap()`, `madvise()`, `msync()`
- Vorteile:
  - Kein Kopieren der Daten vom Benutzer in den Kernadreßraum
  - Kein Interessenkonflik zwischen Seitenadressierung und E/A Puffern
  - Wesentliche Reduktion der Systemaufrufe (kein read/write)
  - Angabe von Zugriffsstrategien (NORMAL, RANDOM, SEQUENTIAL, WILLNEED, DONTNEED)
  - Einfaches Adressierung der Daten innerhalb der Datei über Adreßpointer
  - Mehrere Prozesse können (ohne Kohärenzproblem) auf der gleichen Datei abeiten.