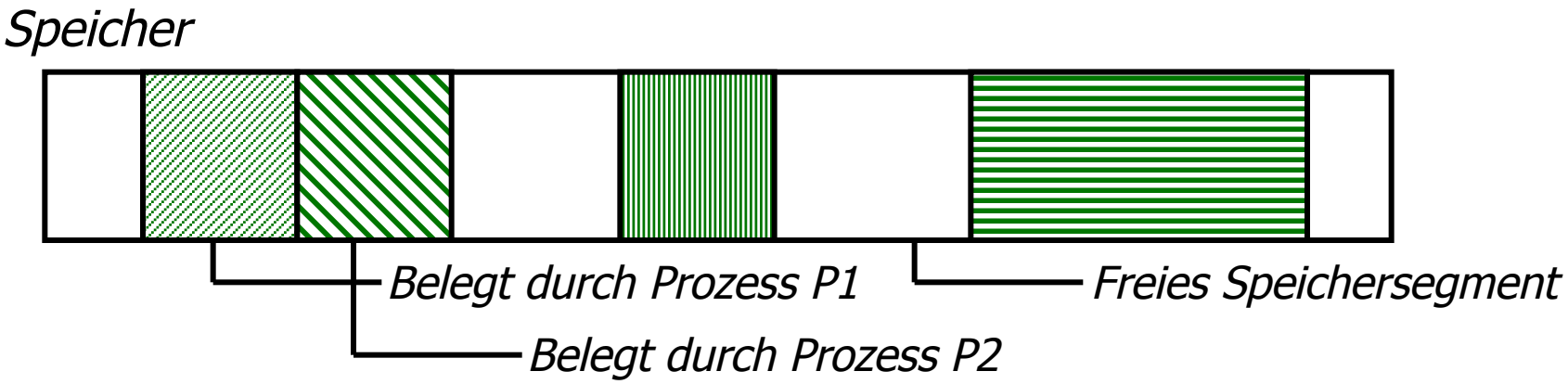
6. Speicherverwaltung

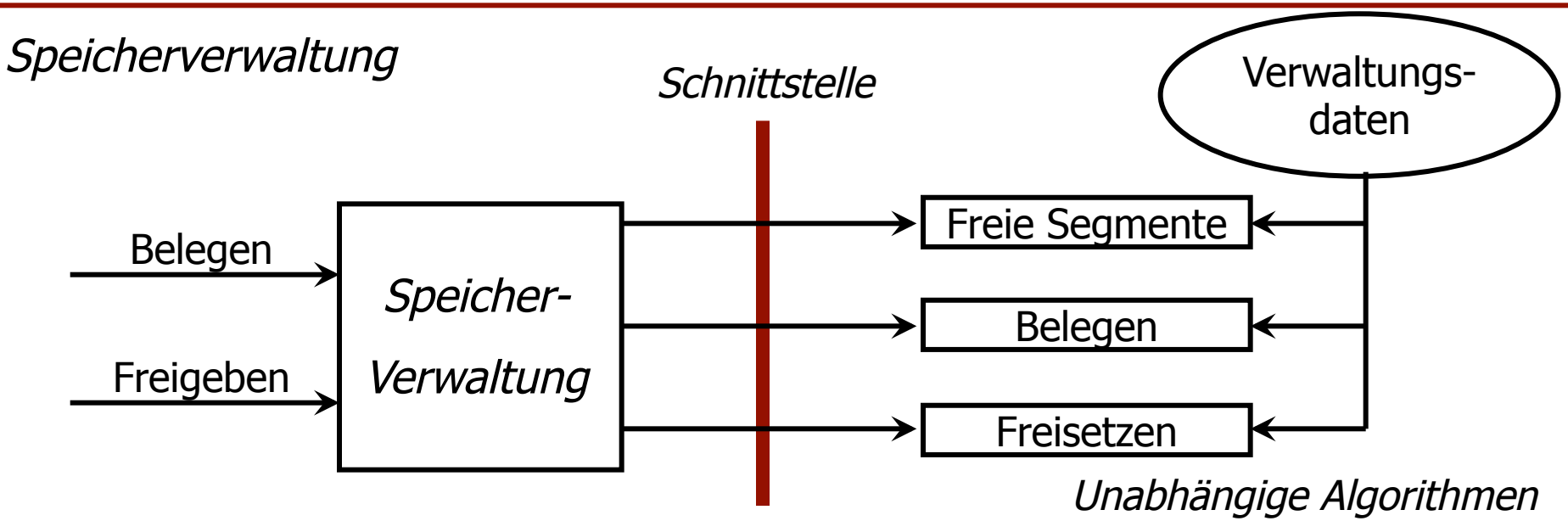
- Überblick
  - 6.1 Speicherverwaltung
  - 6.2 Auswahlstrategien
  - 6.3 Speicherallokation
  - 6.4 Dateisysteme

6.1 Struktur einer Speicherverwaltung

- Adressräume
  - Zusammenhängende Menge von Speicheradressen und deren Inhalte
  - Einer Anwendung wird mindestens ein Adressraum zugeordnet
- Virtueller Speicher
  - „Bereinigter" Adressraum
  - Technische Details sind verborgen
  - Bestehende Beschränkungen werden teilweise aufgehoben
- Zur Laufzeit: Abbildung Virtueller → Physikalischer Speicher
  - Prozessor referenziert virtuelle Adresse
  - Spezielle Hardwareeinheit MMU (Memory Management Unit) transformiert die virtuellen Adressen in Adressen des realen (physikalischen) Speichers

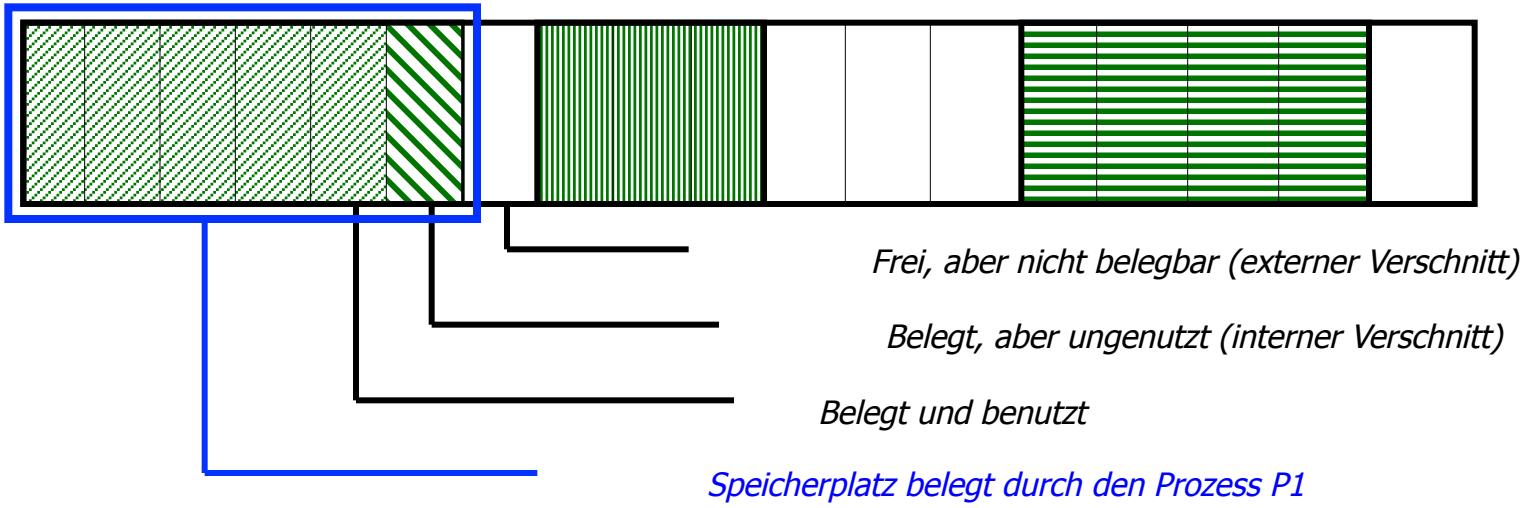
Struktur einer Speicherverwaltung





Speicherfragmentierung

- Zerstückelung des Speichers (*Fragmentierung*, *Verschnitt*) ist ein allgemeines Problem der Speicherverwaltung
  - Interner Verschnitt: Nicht benutzter Speicher innerhalb zugeteilter Segmente
  - Externer Verschnitt: Zu kleine Segmente können *für die aktuelle Anforderung* nicht belegt werden. Insgesamt (Summe aller freien Segmente) ist jedoch genug Speicher für die Anforderung vorhanden



Definitionen zum Verschnitt

- Absoluter interner Verschnitt

$$f_{int-abs} = \textit{Größe des belegten, aber nicht genutzten Speichers}$$

- Verhältnisgrößen
  - Interner Verschnitt

$$f_{int} = \frac{\textit{Größe des belegten, aber nicht genutzten Speichers}}{\textit{Größe des belegten Speichers}}$$

Die Größe kann ein konkreter Wert oder ein Erwartungswert sein

  - Externer Verschnitt

$$f_{ext} = \frac{\textit{Größe des freien, aber nicht belegbaren Speichers}}{\textit{Größe des belegten Speichers}}$$

  - Auslastung

$$\eta = \frac{\textit{Größe des belegten Speichers}}{\textit{Größe des gesamten Speichers}}$$

6.2 Auswahlstrategien

- First-Fit: Durchlaufe die nach Adressen sortierte Liste mit freien Segmenten und belege das erste Segment ausreichender Größe
  - Geringer Suchaufwand, aber hoher externer Verschnitt
  - Konzentration belegter Segmente am Anfang ⇒ Hoher Suchaufwand durch viele unpassende, kleine Segmente am Anfang
- Next-Fit, Rotating First-Fit: Analog zu First-Fit, die Suche beginnt allerdings beim letzten belegten Segment
  - Zyklisches Durchlaufen der Liste
  - Keine Konzentration der belegten Segmente am Listenanfang
  - Verkürzung der Suchzeiten
- Nearest-Fit: Eine gewünschte Adresse für die Belegung wird übergeben ⇒ Von da aus First-Fit-Suche
  - Minimierung der Armbewegungen bei Festplatten (veraltet durch SSD-Platten)
  - Günstige Positionierung oft benötigter Daten, wie z.B. Dateikataloge

Auswahlstrategien (2)

- Best-Fit: Belege das kleinste, aber hinreichend große Stück
- Worst-Fit: Suche und belege das größte freie Speicherstück ⇒ Möglichst großes Reststück noch vorhanden
- Quick-Fit: Extraliste für häufig vorkommende Belegungen ⇒ schnellere Auffindung passender Freistellen
  - Werden z.B. regelmäßig Nachrichten der Länge 1 KB verschickt, wird eine Extraliste für 1-KB-Belegungen geführt
  - ⇒ Solche Anfragen werden schnell und ohne Verschnitt erfüllt

Halbierungsverfahren (Buddy-System)

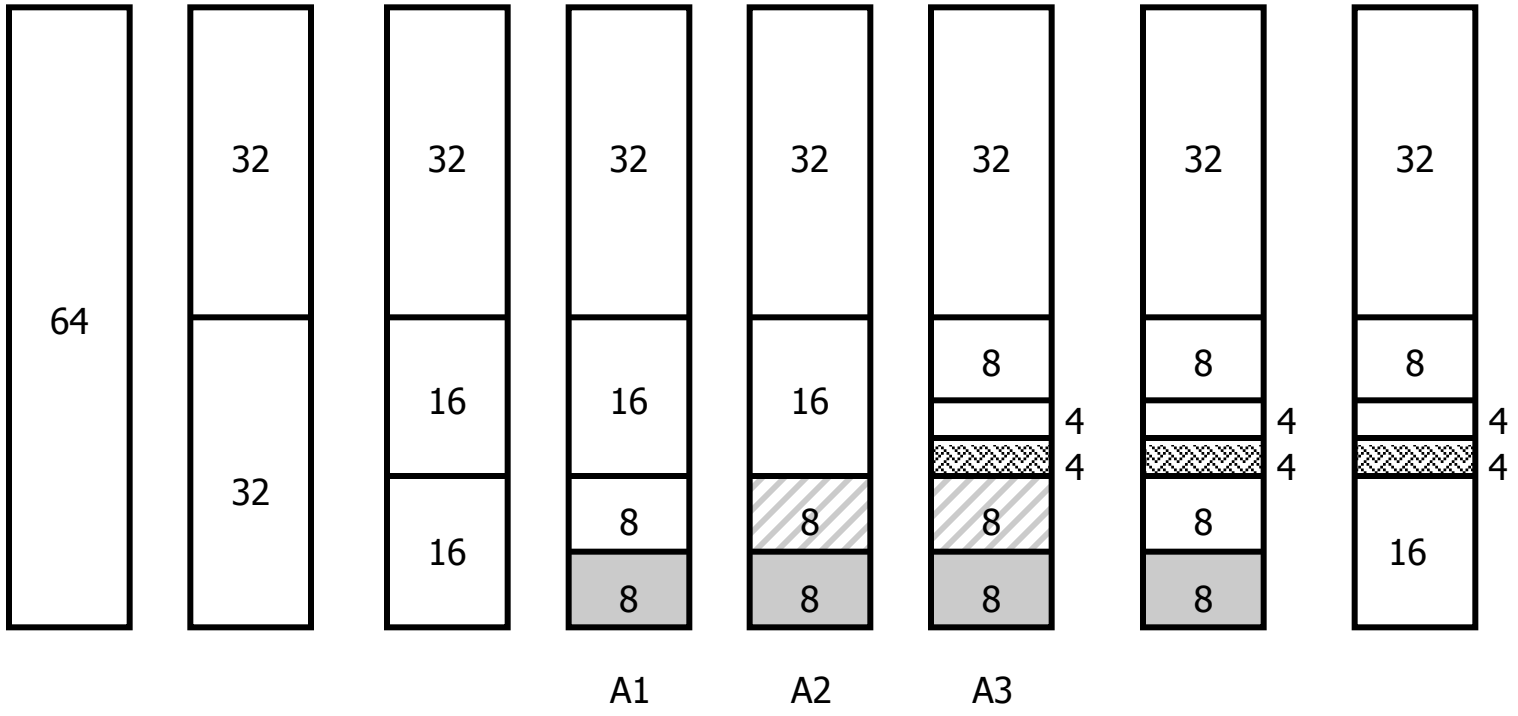
- Speicherverwaltung in früheren Versionen von Linux
- Speicher besteht aus 2^{max} Einheiten
  - Vergabe von Speicherstücken der Größe 2^0, 2^1,...,2^{max}
  - Aufrundung aller Anforderungen auf die nächste Zweierpotenz
  - Ist kein freier Block der Größe 2^k vorhanden, so wird ein freies Speicherstück der Größe 2^{k+1} in zwei Stücke von je 2^k Byte aufgeteilt
  - Die beiden Stücke heißen Partner (*buddy*) und sind genau gekennzeichnet
    - Die Anfangsadressen sind identisch bis auf das *k*-te Bit in ihrer Adresse, das invertiert ist
    - Dadurch schnelle Überprüfung möglich: Existiert zu einem freien Block ein ebenfalls freier Buddy. Falls ja ⇒ Verschmelzung

Ablauf Anforderung/Freigabe

- Ablauf Anforderung
  - Aufrunden auf nächste Zweierpotenz
  - Zugriff auf erstes freies Stück der Liste
  - Falls Liste leer (*rekursiv*)
    - Zugriff auf Liste der nächsten Größe
    - Stück entfernen
    - Stück halbieren
    - Hintere Hälfte in entsprechende Liste einhängen
- Ablauf Freigabe
  - Buddy bestimmen
  - Falls Buddy belegt, freigewordenes Stück in die Liste einhängen
  - Falls Buddy frei
    - Vereinigung mit Buddy
    - Iteration, bis Buddy belegt oder maximale Größe erreicht

Beispiel zu Buddy-System

- Gegeben
  - Speichergröße 64 Einheiten
  - Anforderungen: A1:7 (Aufrundung 8) A2:6 (8) A3:3 (4)

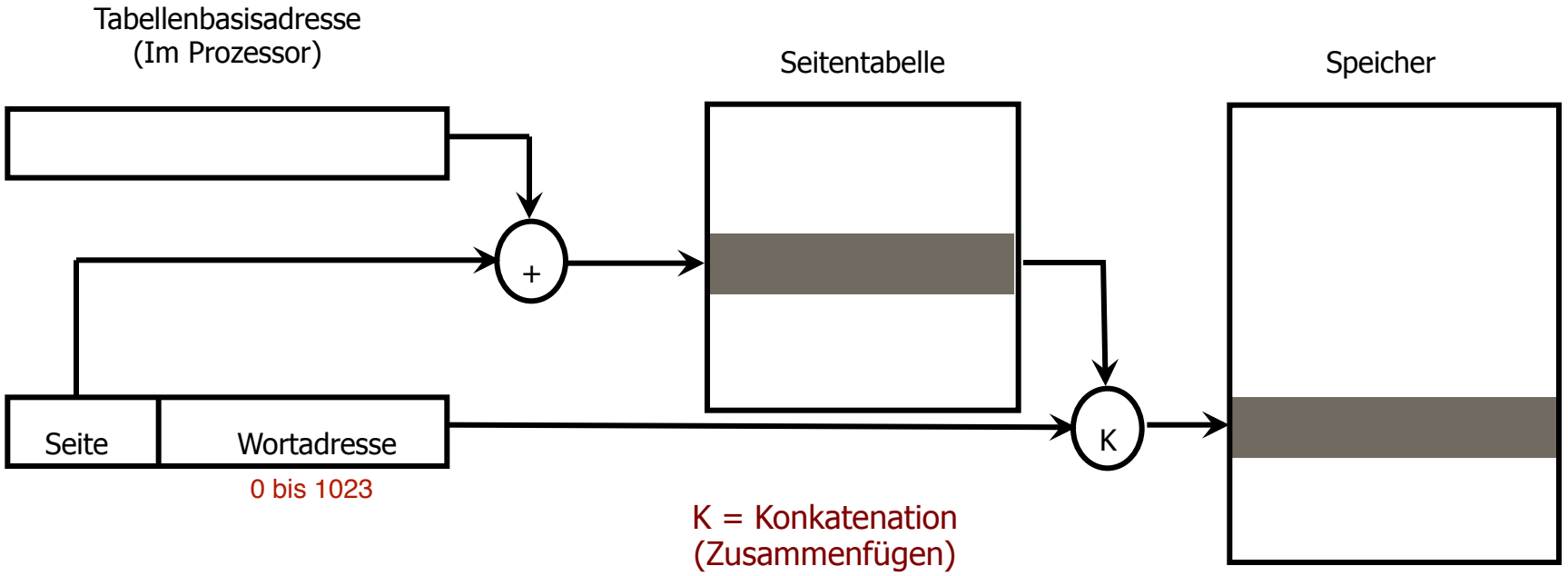


Interner Verschnitt beim Buddy-Verfahren

- Vorteile des Buddy-Verfahrens
  - Schnelle Operationen
  - Anpassung auf das Anforderungsprofil
  - Nach Einschwingen wenig Teilungs- und Vereinigungsvorgänge
- Nachteil: Relativ hoher interner Verschnitt
- Verdeutlichung am Beispiel
  - Anforderungsgröße *a*: 1 2 3 4 5 6 7 8 9 10 …
  - Größe des belegten Stücks *b(a)*: 1 2 4 4 8 8 8 8 16 16 …

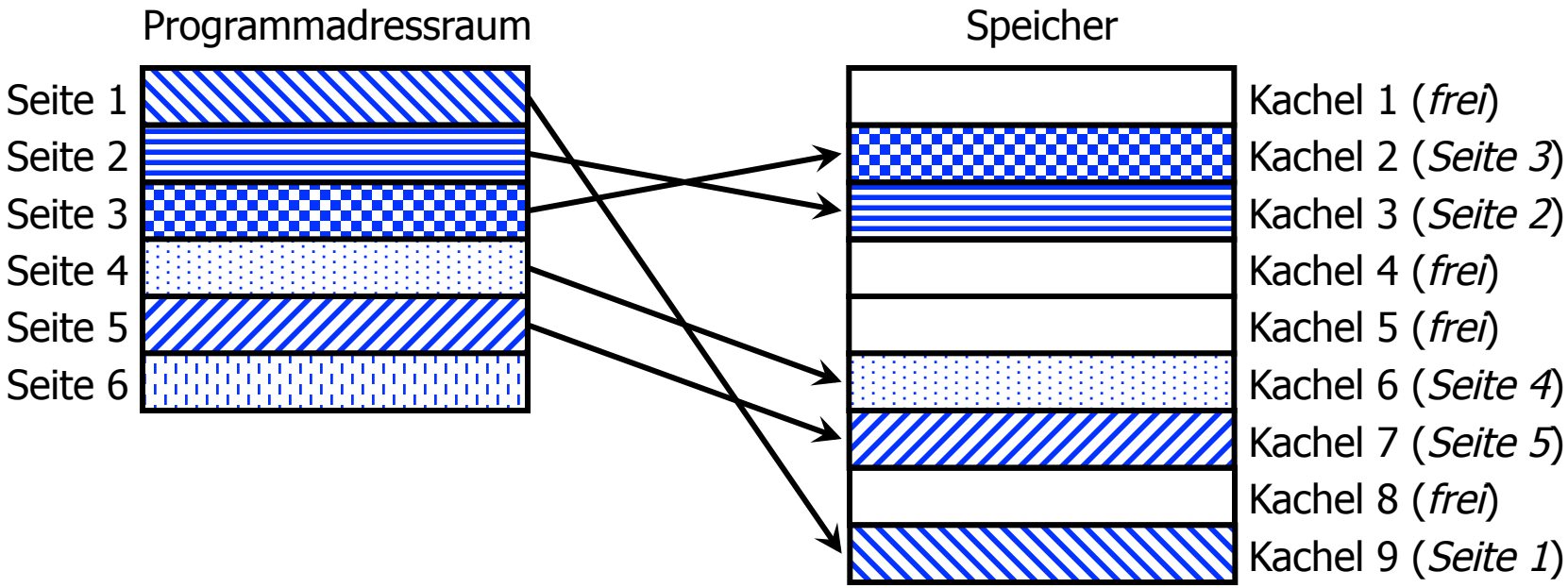
Seitenadressierung

- Seitenbasierte Speicherbelegung ist Standard in Betriebssystemen
  - Einzelne Seiten werden in Speicherkacheln eingelagert
- Notwendige Angaben
  - Wo befindet sich die Seitentabelle ⇒ Tabellenbasisregister (Page Table)
  - Welche Seite wird angesprochen ⇒ Seitennummer
  - Adresse innerhalb der Seite ⇒ Wortadresse (*offset*, *displacement*)



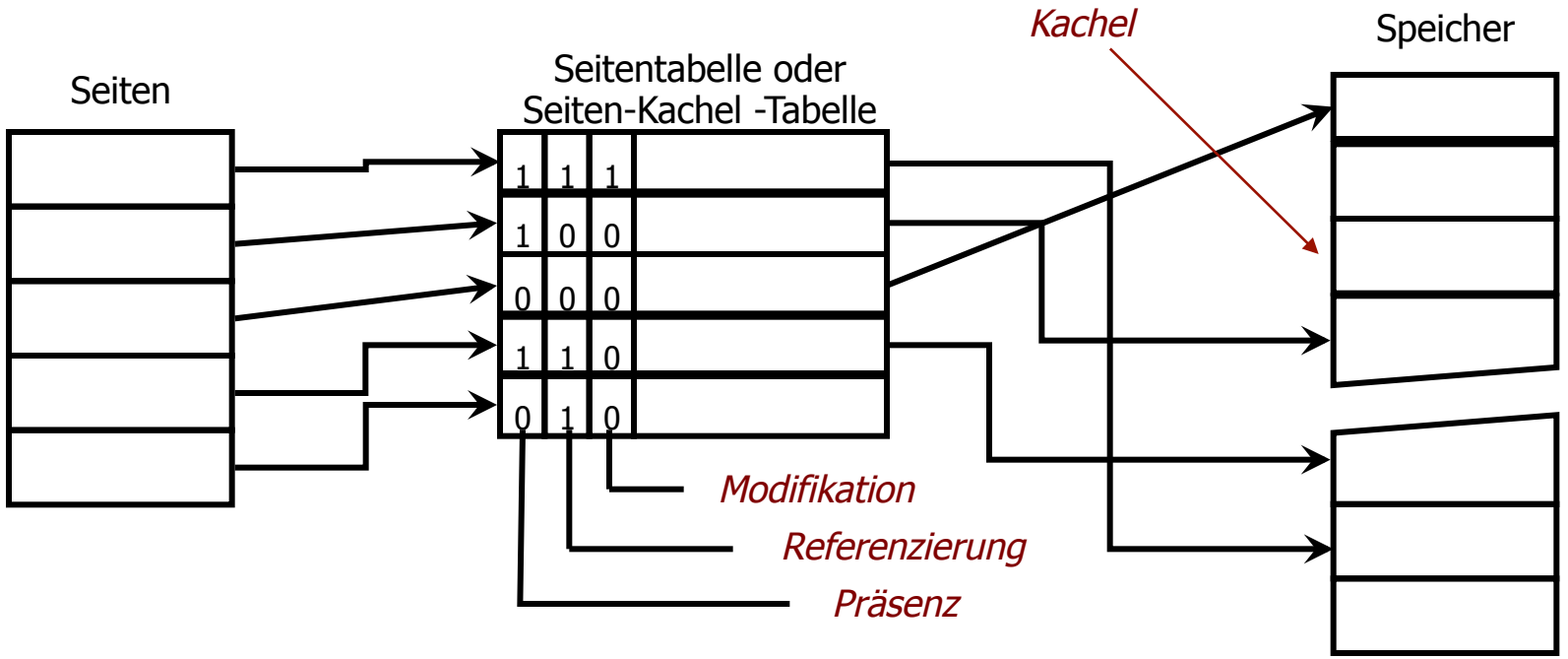
Seitenadressierung

- Bei Seitenadressierung (*paging*) werden der virtuelle und der physikalische Speicher in Stücke fester Länge eingeteilt
  - Stücke im logischen Adressraum heißen Seiten (*pages*)
  - Stücke im physikalischen Speicher heißen Kacheln (*page frames*)
  - Seiten und Kacheln sind gleich groß



Zusätzliche Informationen der Seitentabelle

- Neben der physikalischen Adresse enthält jeder Eintrag der Seitentabelle noch Informationen über
  - Seite im Hauptspeicher vorhanden? Präsenzbit (*presence/valid bit*) Gültigkeit der Adresse
  - Wurde auf die Seite bereits zugegriffen? Referenzbit (*reference bit*)
  - Wurde die Seite modifiziert? Modifikationsbit (*dirty bit*)



Detaillierter Ablauf

Page Fault: Daten sind nicht im Hauptspeicher -> Startet Prozess um Seite wieder zu laden

Belegen des VS

Ersatzspeicher belegen
Initialisierung

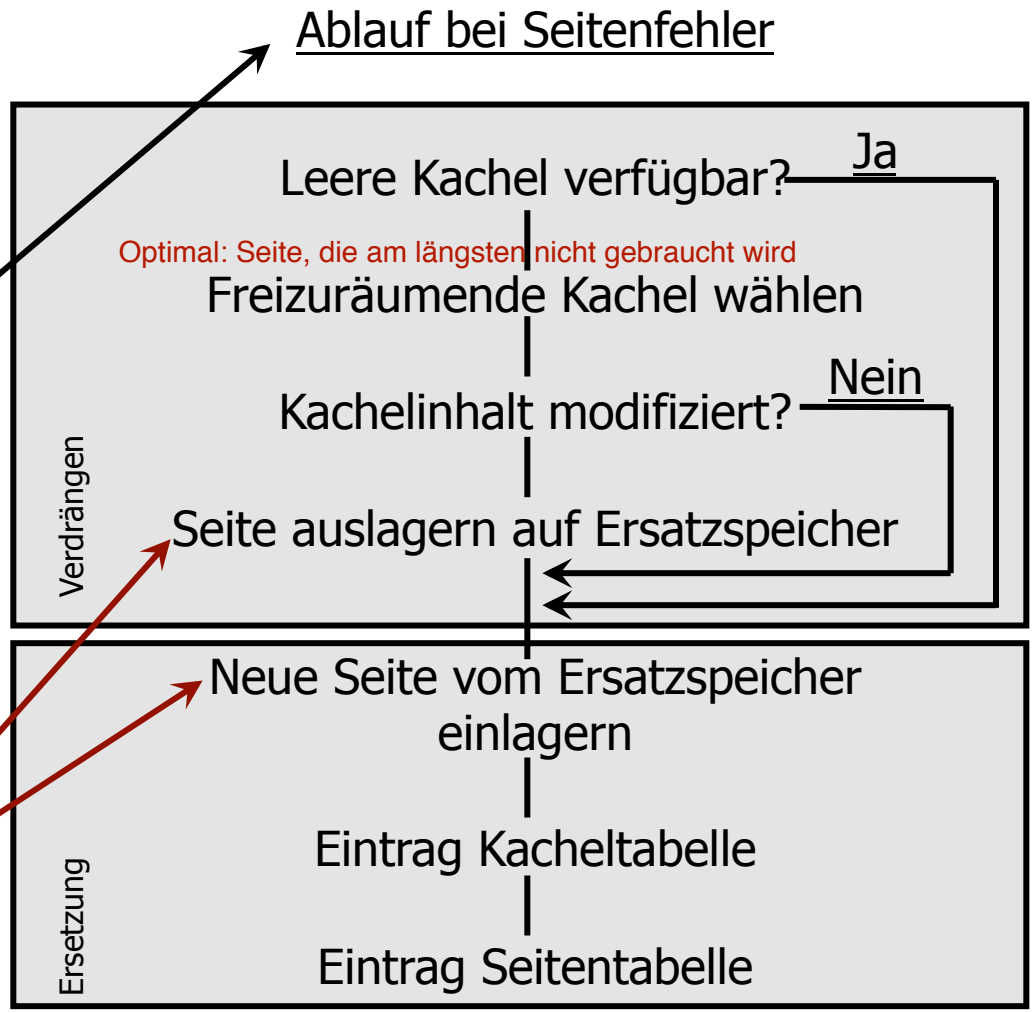
Zugriff

Seite vorhanden?
Nein: Seitenfehler aktivieren
Ja: Setze Verarbeitung fort

Freigabe des VS

Belegte Kacheln freigeben
Ersatzspeicher freigeben

Zeitintensive Verarbeitungsschritte



Invertierte Seitentabellen

- Wichtiges Problem der konventionellen Seitentabellen ist der hohe Speicherbedarf
  - 2^{32} Byte Adressraum/4KB Seite ⇒ Seitentabelle ≥ 4 MB
  - 2^{64} Byte Adressraum/4KB Seite ⇒ Seitentabelle mit 2^{52} Einträgen. Bei 8 Byte/Eintrag gilt Seitentabelle ≥ 32 Petabyte!

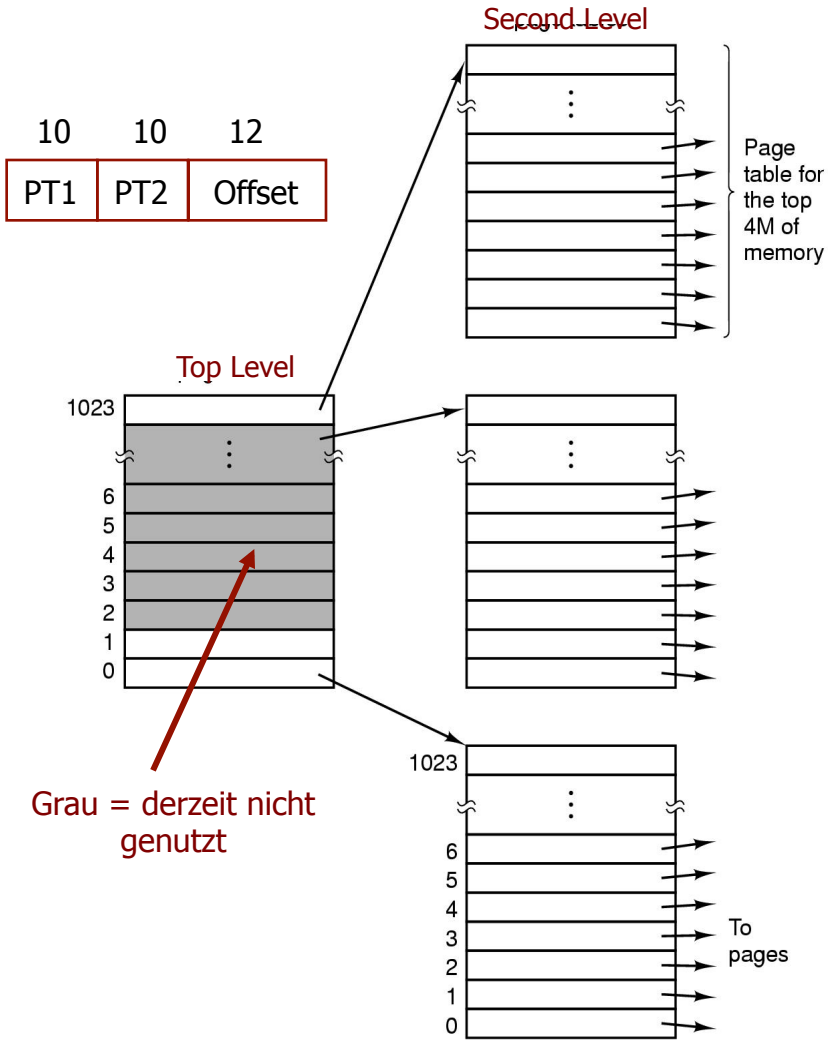
- Zur Adressierung von 64 Bit virtueller Räume andere Lösungen wie Invertierte Seitentabellen notwendig
  - Jeder Eintrag enthält Informationen über eine Kachel
  - Abbildung der virtuellen Adressen unvertretbar rechenaufwendig, da in worst case alle Einträge durchsucht werden müssen
  - Beschleunigungsmöglichkeit z.B. durch Einführung von Seitenverkettung aufgrund von Hashing-Werten

Vorteil: ggf sehr einfach zu berechnen (Schieben von Registern)
Nachteil: Kollisionen

Ungelöstes Problem! -> Workaround in modernen OS = Mehrstufige Tabellen

Mehrstufige Seitentabellen

- Erweiterung der Seitenadressierung
  - Teil der Seitentabellen wird ausgelagert, da i.d.R zwischen Heap und Stapel ungenutzte Speicherbereiche liegen
  - Realisierung durch Aufteilung z.B. einer 32 Bit Adresse in 3 Bereiche
    - PT 1: 10 Bit, Startadresse der Seitentabelle
    - PT 2: 10 Bit, Startadresse der entsprechenden Seite (je 4 KByte)
    - Offset: 12 Bit, Wortadresse innerhalb der Seite
- Im Beispiel: Obwohl über 1 Million Adressen verfügbar, werden nur 4 Seitentabellen tatsächlich gebraucht



6.3 Speicherallokation

- Speicherallokation = Zuordnung von Speicher
- Statische Speicherallokation
  - Einteilung des Arbeitsspeichers in mehrere Laufbereiche
    - Nehmen einen Prozess vollständig auf
    - Prozess blockiert ⇒ Verdrängung aus aktuellem Laufbereich und Auslagerung auf den externen Speicher
    - Wiedereinlagerung nach Wechsel *blockiert* → *bereit*
  - Nachteil: Nur ganze Prozesse werden verdrängt ⇒ ineffizient
- Dynamische Speicherallokation nur die Teile laden, die wirklich gebraucht werden
  - Prozessteile werden – transparent für den Benutzer – zur Laufzeit und erst bei Bedarf ein- und ausgelagert
  - Grundlage: Virtuelle Adressierung
  - Beim Zugriff auf fehlenden Inhalt wird ein Interrupt (Seitenfehler, Page Fault) ausgelöst, der das Nachladen des Inhalts initiiert

Nachschub und Verdrängung

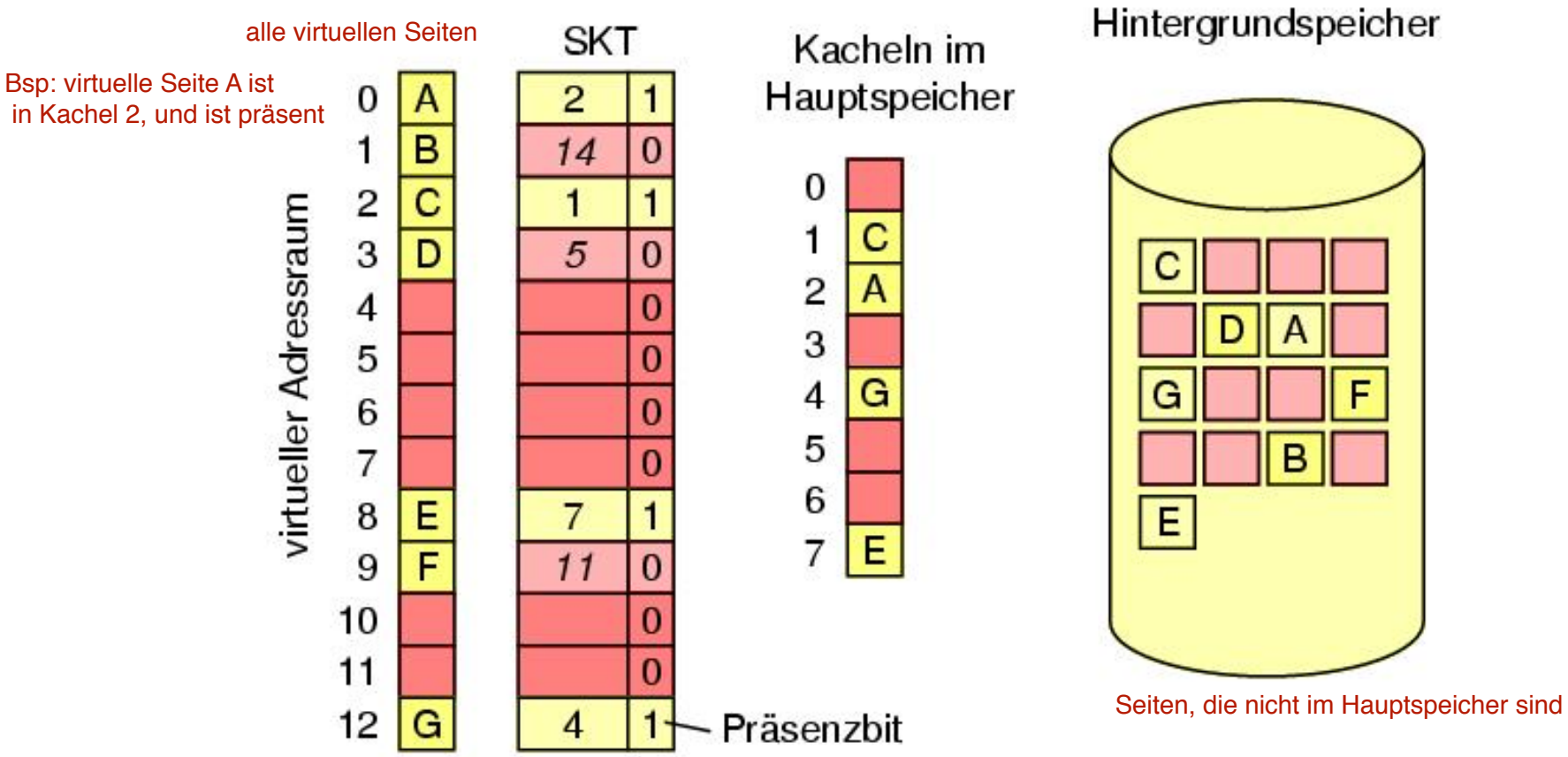
- Wichtig für dynamische Speicherallokation
  - Realokierbare Prozesse können verdrängt und ohne zusätzlichen Programmieraufwand in anderem Speicherbereich ausgeführt werden
- Festgelegte Strategien, wann
  - Daten in den Speicher zu laden sind (*Nachschubstrategie*)
  - Prozessdaten aus dem Speicher verdrängt werden müssen (*Verdrängungsstrategie*)
- Nachschub (Fetch Policies)
  - Auf Verlangen (Demand paging): Seite wird erst bei Bedarf nachgeladen
  - Vorgeplant (Pre-paging)
- Verdrängung (Replacement Policies)
  - Ein Prozess oder Teile eines Prozesses werden vom Arbeitsspeicher auf einen Hintergrundspeicher ausgelagert
  - Dies wird immer dann notwendig, wenn der einem Prozess zugeteilte Arbeitsspeicher nicht ausreicht

Nachschubstrategien (Fetch Policies)

- Auf Verlangen (Demand paging)
  - Die Seite wird erst bei aktuellem Bedarf nachgeladen
  - Interrupt Seitenfehler (page fault) aktiviert die Allokation
  - Diese Strategie wird üblicherweise angewendet
- Vorgeplant (Pre-paging)
  - Voraussetzung: Prozessverhalten im Voraus bekannt
  - Transport einer Seite in den Arbeitsspeicher so, dass die Seite zum Referenzierungszeitpunkt zur Verfügung steht
  - Die notwendige Kenntnis über das exakte Programmverhalten steht meist nicht zur Verfügung
- Kombinierte Ansätze (Meist in Praxis verwendet)
  - Beim Start Pre-paging, z.B. Anfang des Programmcodes, statische Daten, Seiten für Heap und Stapel, …
  - Während Laufzeit: Demand paging

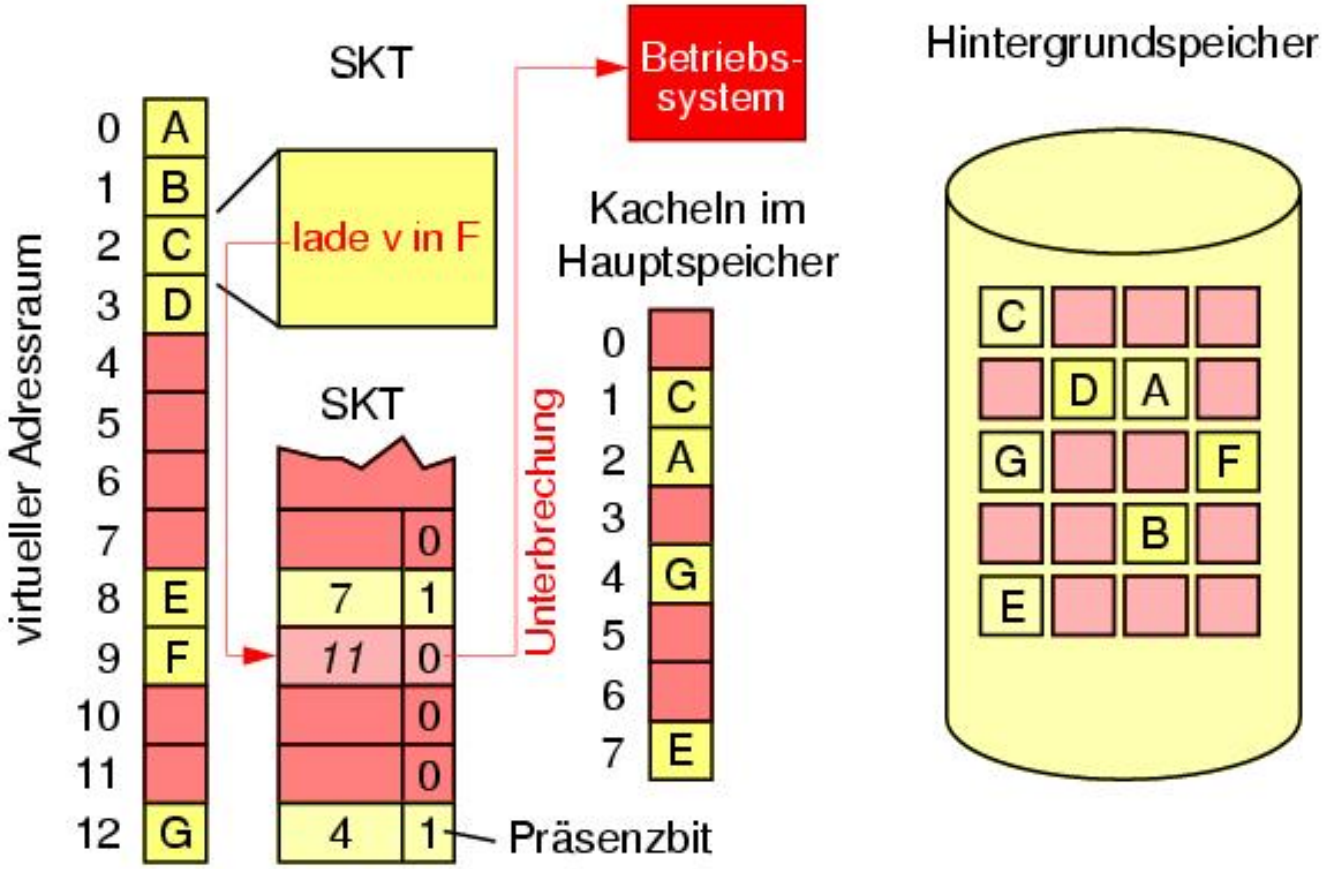
Demand Paging

SKT = Seite-Kachel-Tabelle

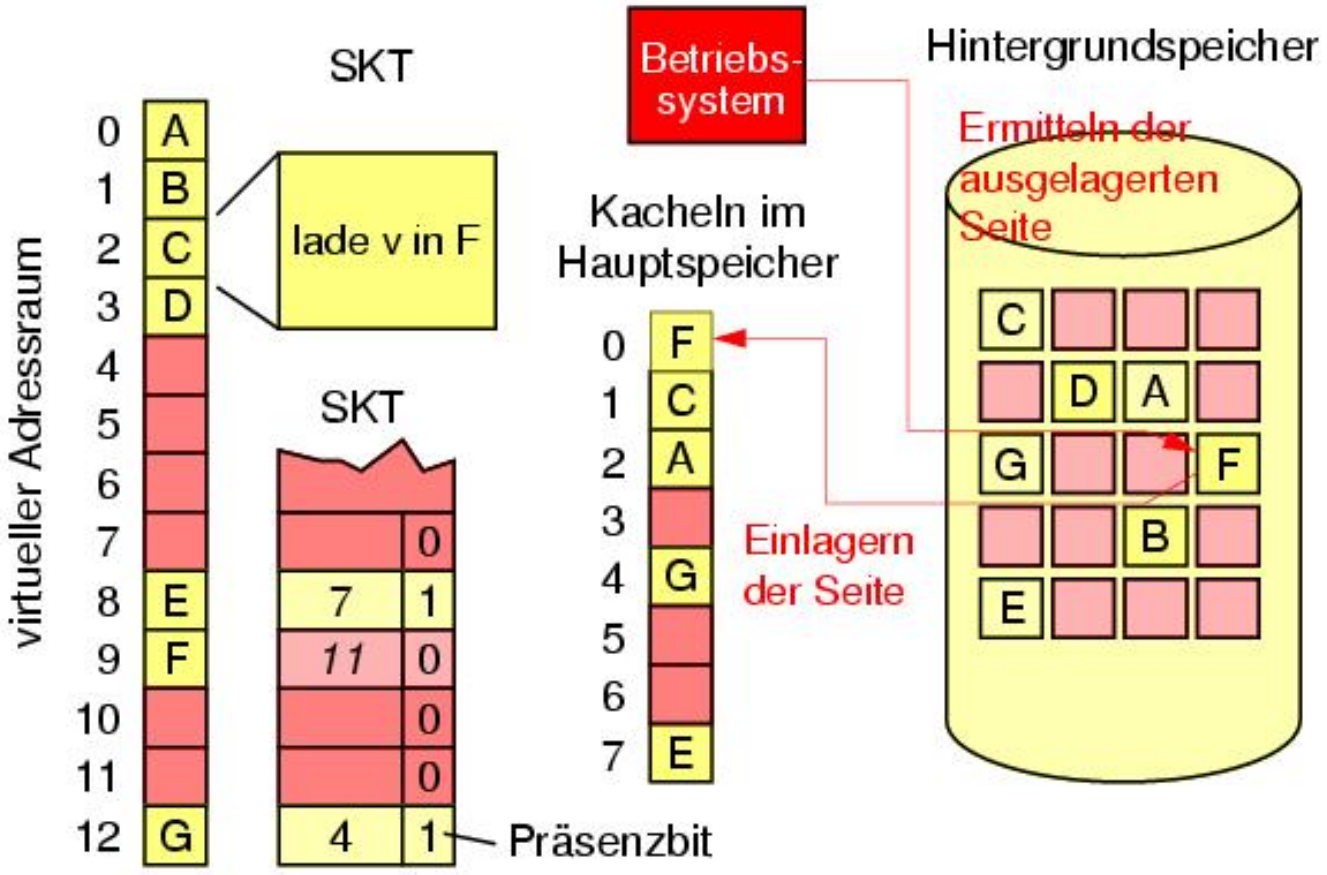


Demand Paging (2.)

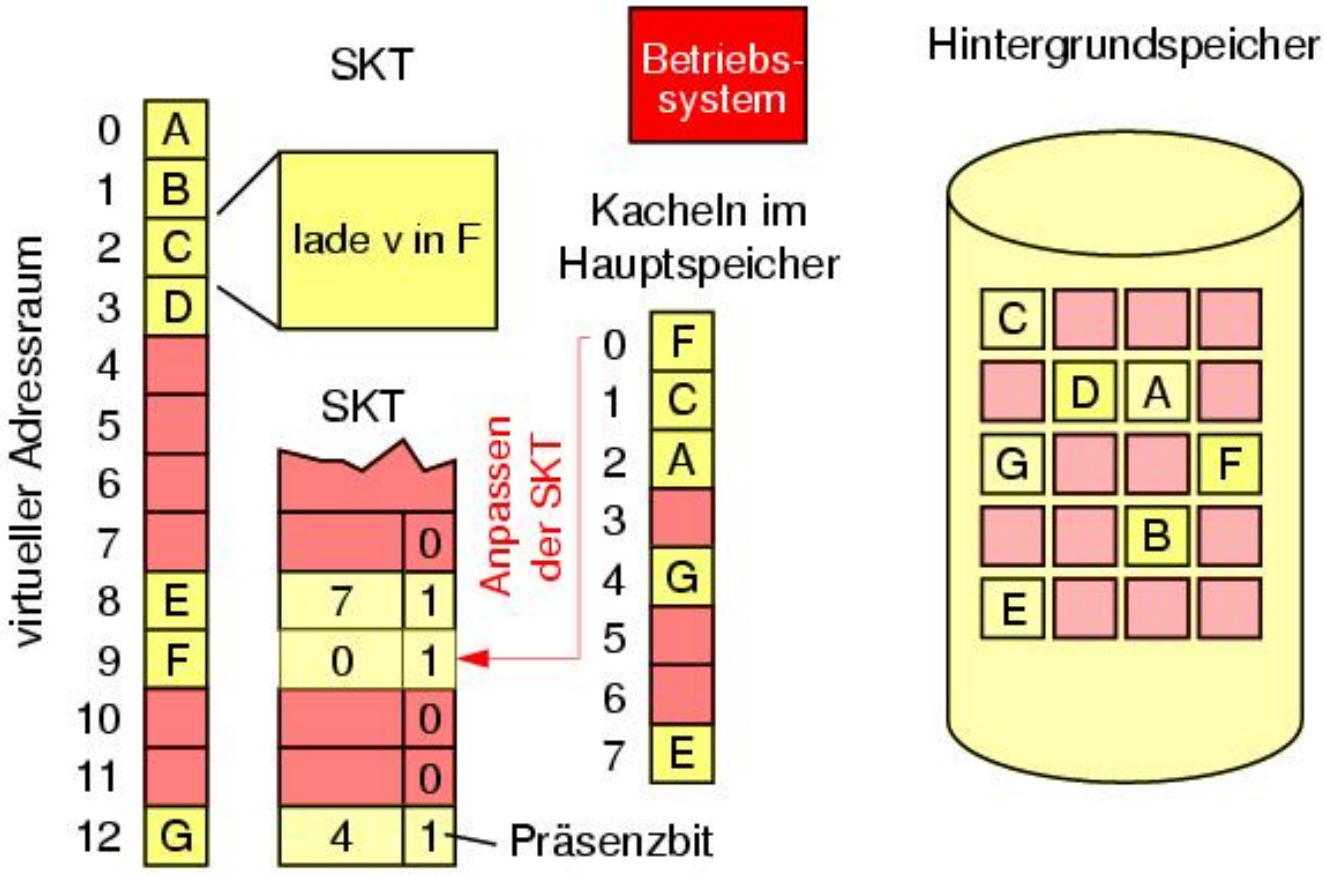
Da Präsenzbit = 0: Seite F ist nicht im Hauptspeicher -> liegt nur auf Festplatte
-> Page Fault und Interrupt durch OS



Demand Paging (3.)

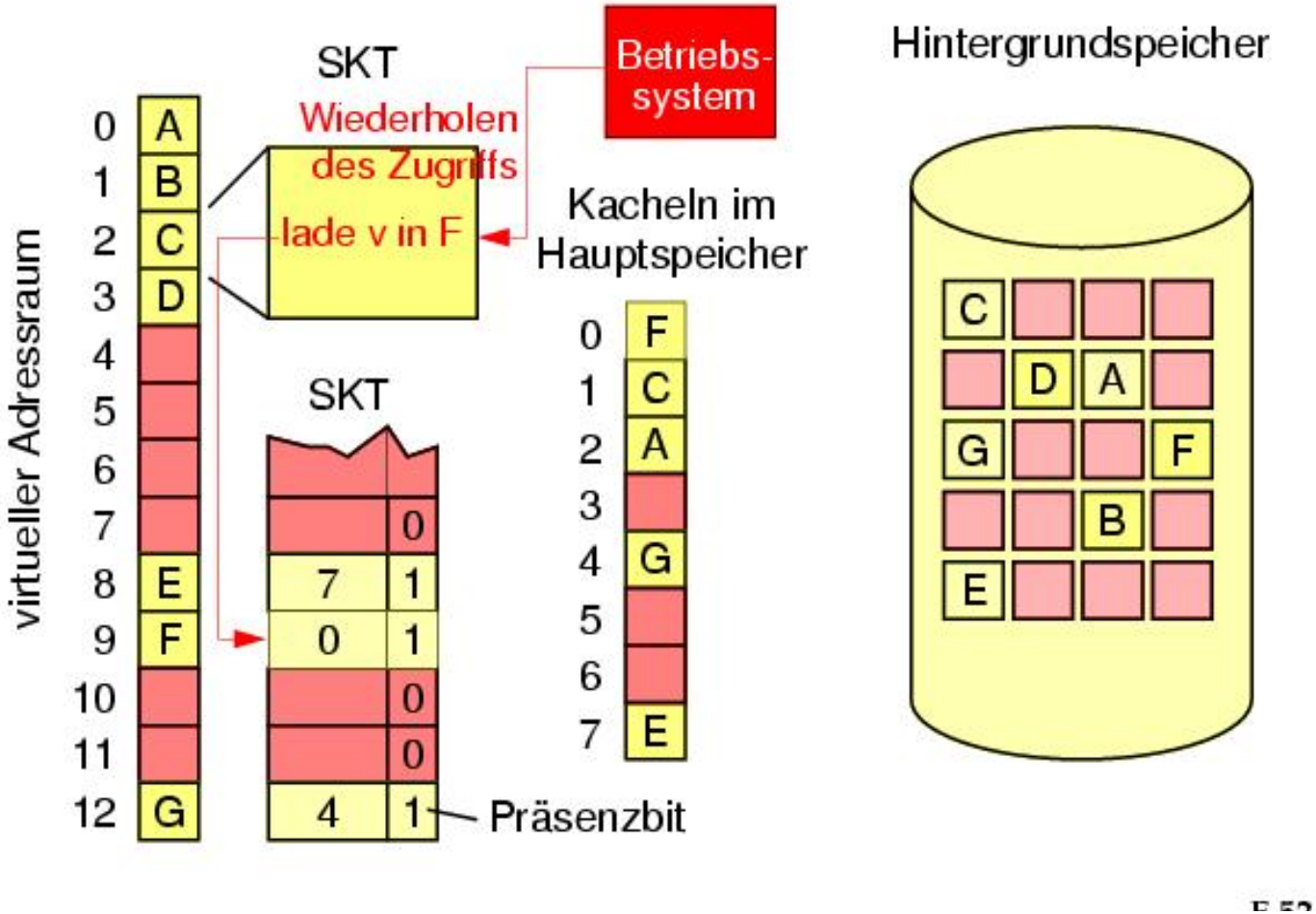


Demand Paging (4.)



Demand Paging (5.)

Jeder Prozess hat eine eigene Seitentabelle!
-> auch bei 32-bit großer Speicherbedarf



Verdrängungsstrategien (*Replacement Policies*)

- Demand Paging Leistungsfähigkeit ist stark von der Anzahl der Seitenfehler abhängig
  - Seitenfehler durch intelligente Verdrängungsstrategien minimieren
- Welche Kachel soll geleert und der Inhalt ausgelagert werden?
  - Lokale Auswahlstrategie
    - Es wird eine Kachel geleert, welche dem Seitenfehler verursachenden Prozess zugeordnet ist
  - Globale Auswahlstrategie
    - Eine beliebige Kachel – auch von fremden Prozessen – darf geleert werden
- Modellierung der Seitenersetzung
  - Ist r_i die Nummer der Seite, auf der zum Zeitpunkt t zugegriffen wird, so heißt $R = r_1, r_2, r_3, \ldots, r_n$ Seitenreferenzfolge

Optimale Auswahlstrategie

- Verdränge diejenige Seite, die am längsten nicht mehr benötigt *werden wird* ☺
  - Minimierung der Seitenfehleranzahl bei gegebener Speichergröße
  - Nicht realisierbar ⇒ wird als Messlatte für realisierbare Strategien eingesetzt, d.h. wie weit ist eine Strategie vom optimalen Ergebnis noch entfernt

Zeit	1	2	3	4	5	6	7	8	9	10	11	12
Referenz	0	2	4	3	2	3	0	1	0	3	0	2
Kachel 1	0	0	0	0	0	0	0	0	0	0	0	**2**
Kachel 2		2	2	2	2	2	2	**1**	1	1	1	1
Kachel 3			4	**3**	3	3	3	3	3	3	3	3

Ergebnis: 3 Fehler (*ohne Initialseitenfehler*)

Realisierbare Strategien

- Vorhersage zukünftiger aufgrund vergangener Referenzen
- Lokalitätsprinzip
  - *Zugriffsverhalten in unmittelbarer Vergangenheit ist eine gute Schätzung für das Verhalten in nächster Zukunft*
- Auswahl basiert auf
  - Häufigkeit der Zugriffe auf eine Seite
  - Zeitpunkt der Zugriffe
- Strategien: Verdrängung der Seite, die
  - Am längsten im Speicher war (*First In - First Out*, FIFO)
  - Am längsten nicht benutzt wurde (*Least Recently Used*, LRU)
  - Am wenigsten häufig benutzt wurde (*Least Frequently Used,* LFU)
  - Innerhalb eines vorgegebenen Zeitraums nicht mehr referenziert wurde (*Recently Not Used,* RNU)

FIFO-Strategie

- FIFO: 4 Seitenfehler für die betrachtete Referenzfolge
- Anomalie der FIFO-Strategie
  - Bei steigender Anzahl von Kacheln steigt die Anzahl von Seitenfehlern
  - Beispiel: Bei 4 Kacheln ⇒ 7 Fehler, bei 5 Kacheln ⇒ 8 Fehler
- Erwünscht
  - Weniger Seitenfehler, wenn mehr Speicher zur Verfügung steht
  - Monoton fallende Seitenfehlerrate bei steigender Kachelrate

Zeit	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Referenz	0	1	2	3	4	0	1	5	6	0	1	2	3	5	6
Kachel 1	0	0	0	0	**4**	4	4	4	**6**	6	6	6	6	6	6
Kachel 2		1	1	1	1	**0**	0	0	0	0	0	**2**	2	2	2
Kachel 3			2	2	2	2	**1**	1	1	1	1	1	**3**	3	3
Kachel 4				3	3	3	3	**5**	5	5	5	5	5	5	5

LFU-Strategie

- Verdränge Seite, die am wenigsten häufig benutzt wurde
  - Eine kaum verwendete Seite wird auch zukünftig selten benötigt
  - Ein Zähler pro Seite ⇒ Inkrementierung bei jedem Zugriff
  - Bei gleicher Frequenz: 2. Strategie, z.B. FIFO

Zeit		1	2	3	4	5	6	7	8	9	10	11	12
Referenz		0	2	4	3	2	3	0	1	0	3	0	2
Kachel 1		0	0	0	**3**	3	3	3	3	3	3	3	3
Kachel 2			2	2	2	2	2	2	**1**	1	1	1	**2**
Kachel 3				4	4	4	4	**0**	0	0	0	0	0
	0	1	1	1	(1)	(1)	(1)	2	2	3	3	4	4
	1	-	-	-	-	-	-	-	1	1	1	1	(1)
Zähler	2	-	1	1	1	2	2	2	(2)	(2)	(2)	(2)	3
	3	-	-	-	1	1	2	2	2	2	3	3	3
	4	-	-	1	1	1	1	(1)	(1)	(1)	(1)	(1)	(1)

LRU-Strategie

- Verdränge Seite, die am längsten nicht mehr referenziert wurde
  - „Modifizierter" Stapel: die Seite, auf die zuletzt zugegriffen wurde, wird auf oberste Stapelposition gelegt ⇒ Oberste k Seiten im Speicher
  - Falls Seite schon im Speicher, wird sie von alter Stapelposition gelöscht
  - Bei Auslagerung: k-te Seite austauschen, neu referenzierte Seite kommt auf oberste Position ⇒ Vorhandene Seiten „rutschen" nach unten

Zeit	1	2	3	4	5	6	7	8	9	10	11	12
Referenz	0	2	4	3	2	3	0	1	0	3	0	2
Kachel 1	0	0	0	**3**	3	3	3	3	3	3	3	3
Kachel 2		2	2	2	2	2	2	**1**	1	1	1	**2**
Kachel 3			4	4	4	4	**0**	0	0	0	0	0
top	0	2	4	3	2	3	0	1	0	3	0	2
Stapel 1	-	0	2	4	3	2	3	0	1	0	3	0
2	-	-	0	2	4	4	2	3	3	1	1	3

RNU-Strategie

- Verdränge Seite, die innerhalb eines Zeitraums nicht mehr referenziert wurde
  - Definition des Zeitraums über ein Fenster, das k zuletzt referenzierte Elemente umfasst
  - Für eine Verdrängung kommen alle Seiten in Frage, die nicht innerhalb des Fensters referenziert wurden
  - Kritische Größe: Fensterbreite k>0, aber k klein
  - Beispielreferenzfolge: 4 Seitenfehler, bei k=2

Zeit	1	2	3	4	5	6	7	8	9	10	11	12
Referenz	0	2	4	3	2	3	0	1	0	3	0	2
Kachel 1	0	0	0	**3**	3	3	3	3	3	3	3	3
Kachel 2		2	2	2	2	2	2	**1**	1	1	1	**2**
Kachel 3			4	4	4	4	**0**	0	0	0	0	0

Vergleich der Strategien

- Von den diskutierten Strategien zeigt LRU im Durchschnitt die beste Leistung, d.h. geringste Seitenfehlerrate
  - ⇒ Häufige Anwendung in realer Systemsoftware

- Interessant
  - LRU ist symmetrisch zu der optimalen Strategie
  - Vom aktuellen Zeitpunkt betrachtet LRU die Vergangenheit und die optimale Strategie die Zukunft

- Probleme bei der Realisierung
  - Alle realisierbaren Strategien erfordern bei jedem Zugriff gewisse Datenoperationen (Stapeloperationen, Zählerinkrementierung, …)
  - Durchführung dieser Operationen (vollständig in Software, mit Unterstützung von Hardware) ist zu aufwendig
  - Daher werden hauptsächlich *Annäherungsverfahren* realisiert

Angenäherte LRU/NRU-Strategie

- Hardwareunterstützung beim Seitenzugriff
  - Jeder Seite wird ein Referenzbit zugeordnet
  - Die Referenzbits werden bei jedem Zugriff aktualisiert
  - Keine Information über dem *Zeitpunkt eines Zugriffs*
- Fehlende Zeitinformation wird durch eine periodische Rücksetzung der Referenzbits simuliert
  - Variante 1: Bei jeder Rücksetzung wird ein Zähler inkrementiert, wenn die Seite in der Zeit nicht referenziert wurde ⇒ Verdrängung der Seite mit dem höchsten Zähler
  - Variante 2: Die alten Werte der Referenzbits werden einer in einer Zusatztabelle gespeicherten Bitfolge vorangestellt. Interpretiert man die Bitfolge als vorzeichenlose Zahl, so bestimmt die kleinste Zahl diejenige Seite, auf die am längsten nicht mehr zugegriffen wurde
  - Beispiel: Aktuelles Referenzbit: 1; Bisherige Bitfolge 001 ⇒ Neuer Wert 1001
- Beispiel: Second-Chance-Algorithmus (Clock-Algorithmus)

Second-Chance-Algorithmus (*Clock-Algorithmus*)

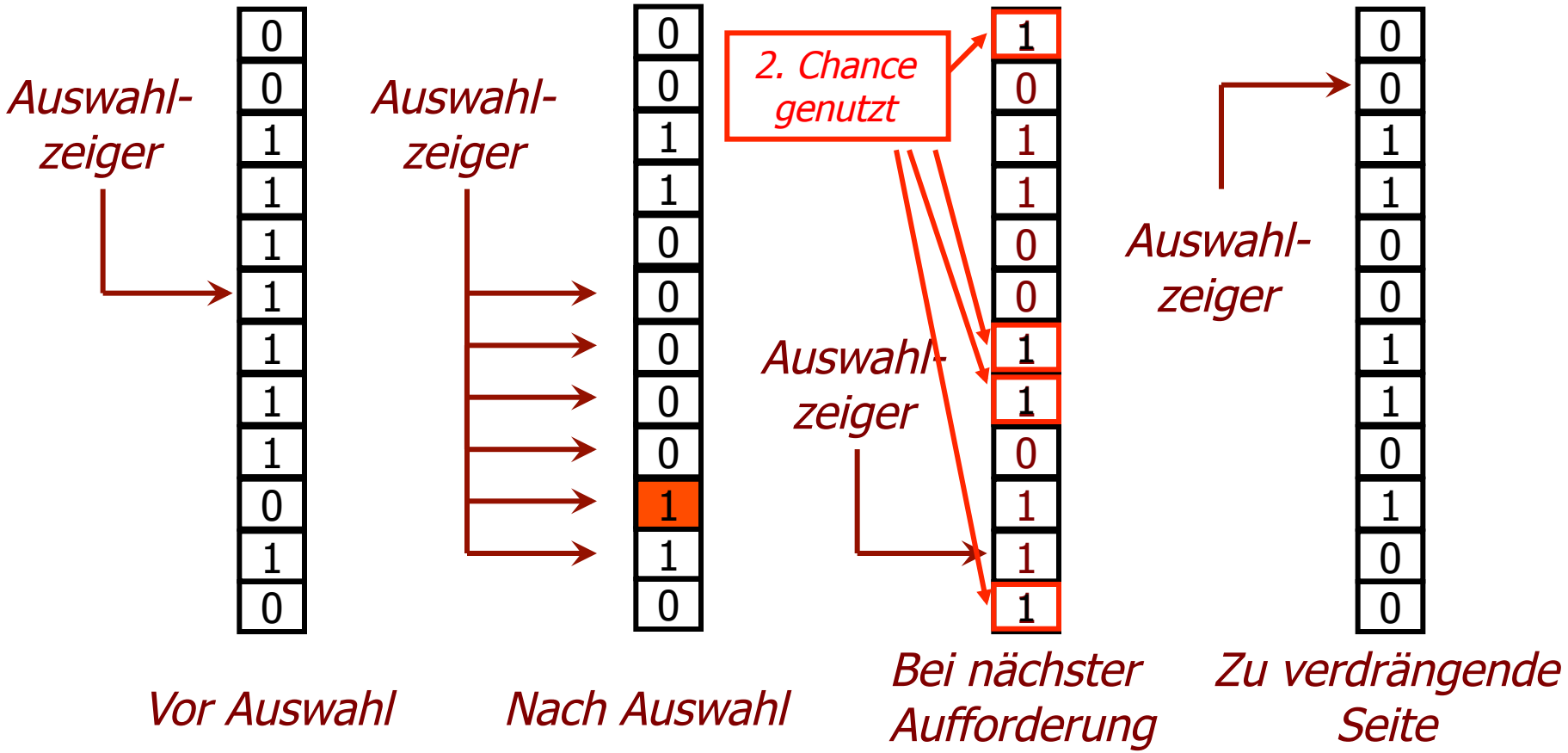
- FIFO-Modifikation durch Berücksichtigung von Referenzen
  - Sortiere die Seiten/Referenzbits gemäß Einlagerungsdauer
  - Durchlaufe zyklisch den Vektor mit Referenzbits
    - Falls das Referenzbit der aktuellen Seite =1
      - Setze das Referenzbit auf 0
      - Betrachte die nächste Seite
    - Falls das Referenzbit der aktuellen Seite =0
      - Verdränge die Seite
      - Setze die Suche mit der nächsten Seite fort

- Rücksetzung von Teilmengen von Referenzbits (*Alte* bis *Neue Position*). Alle anderen Seiten mit Referenzbit 0 haben eine zweite Chance, referenziert zu werden

- Clock-Algorithmus ist eine Implementierungsvariante

Second-Chance-Algorithmus

- Beispielsituation (Bitfolgen = Referenzindikatoren)



Lokalitätsprinzip (*Principle of Locality*)

- Ein Prozess greift im kurzen Zeitraum Δt nur auf einen kleinen Teil seines Adressraums $\Delta A \subset A$ zu

  Räumliche Lokalität: Nach einem Zugriff auf Adresse *a* ist ein Zugriff auf eine Adresse in der Nähe von *a* sehr wahrscheinlich

  Zeitliche Lokalität: Nach einem Zugriff auf Adresse *a* ist ein erneuter Zugriff (in Kürze) auf *a* sehr wahrscheinlich

- Warum?
  - Meist sequentielle Ausführung von Anweisungen
  - Oft vorkommende Schleifen (zeitliche Lokalität)
  - Ausführung bestimmter Programmteile nur in Ausnahmefällen

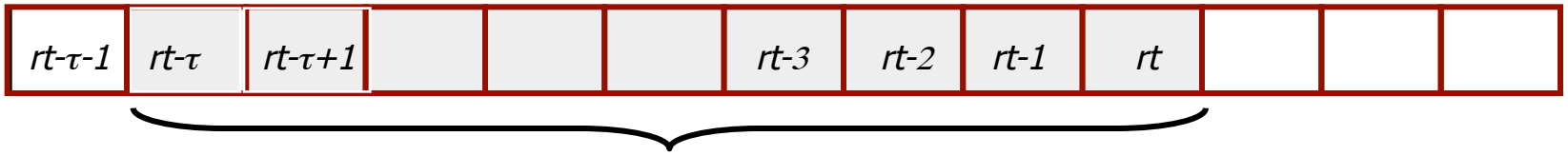
  $\Rightarrow$ 90/10-Regel: Prozesse verbringen 90% der Zeit in 10% des Adressraums

- Konsequenz
  - Nur kleine Teile des Adressraums werden simultan benötigt

  $\Rightarrow$ Auslagerung des Rests auf unteren Ebenen der Speicherhierarchie

Working Set

- Working-Set (*Arbeitsbereich*)
  Menge von Seiten $W_i(t,\tau)$, auf die ein Prozess i innerhalb der letzten τ Einheiten ab dem Zeitpunkt t zugegriffen hat



- Exakte Verfahren z.B. mit einem Schieberegister, in dem die letzten k Seiten gespeichert werden, zu aufwendig
- Annäherung
  - Verwendung der Ausführungszeit des Prozesses ⇒ Working Set = Seiten, auf die innerhalb der τ Einheiten zugegriffen wurde
  - Wichtig: nur die eigene Ausführungszeit eines Prozesses zählt (virtuelle Zeit, current virtual time)

Realisierung des Working-Sets

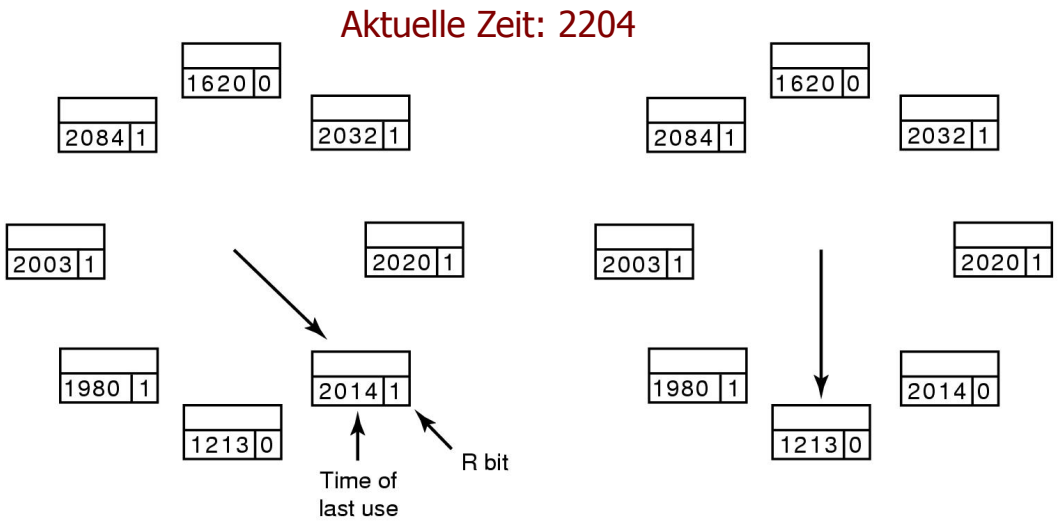
- Jeder Eintrag in der Seitentabelle enthält mindestens
  - Referenzbit (R)
  - Modifikationsbit (M)
  - Ungefähre Zeit des letzten Zugriffs
- Basierend auf einer Timerunterbrechung werden die R-Bits in regelmäßigen Abschnitten auf 0 (kein Zugriff) zurückgesetzt
- Bei jedem Zugriff wird das R-Bit untersucht und falls gesetzt – Seite im Speicher – wird die Zugriffszeit aktualisiert
- Bei Auslagerung wird die Seite mit Referenzbit 0 und ältester Zugriffszeit ausgewählt
- Realisierung des Verfahrens umständlich, da bei jedem Seitenfehler die gesamte Seitentabelle durchsucht werden muss

Effizientere Ausführung: WSClock-Algorithmus

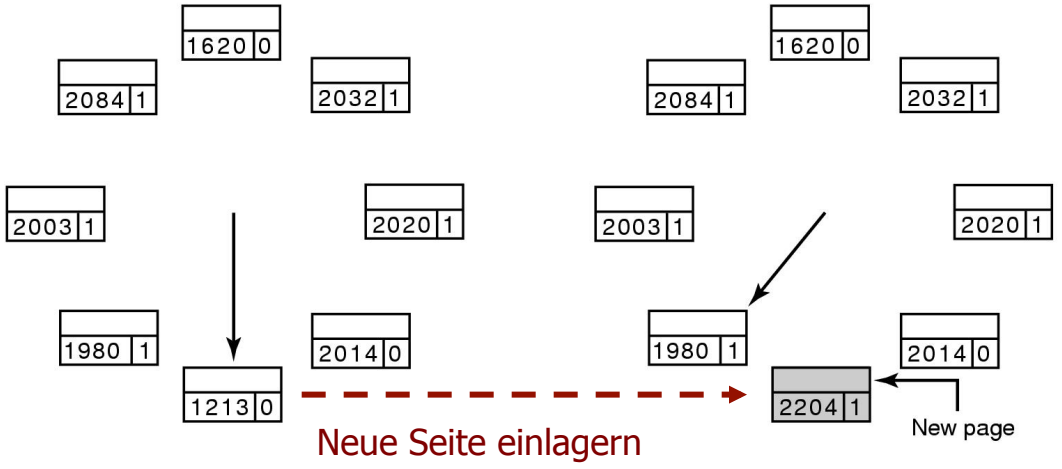
- WSClock-Algorithmus als Erweiterung des Clock-Algorithmus
  - Ringförmige Liste der Kacheln, jeder Eintrag mit mindestens R-Bit, M-Bit und letzter Zugriffszeit
  - Beim Seitenfehler: Untersuche zuerst die mit dem Zeiger markierte Seite
  - Falls R=1 ⇒ Setze R=0, verschiebe den Zeiger zur nächsten Kachel
  - Falls R=0 ⇒ Auslagerungskandidat
    - Falls Zugriff älter als t ⇒ untersuche Modifikationsbit
      - Falls M=0 (kein Schreibzugriff) ⇒ Verdränge Seite
      - Falls M=1 ⇒ Inhalt muss auf Festplatte geschrieben werden
        - Noch kein Austausch, um Prozessblockierung zu vermeiden, lediglich Markieren der Seite als Kandidat
      - Setze Zeiger eine Position fort und untersuche das R-Bit
  - Wenn die gesamte Liste durchlaufen wurde, sind zwei Fälle möglich
    - Mindestens eine Seite wurde zum Auslagern vorgemerkt
    - Keine Seite wurde vorgemerkt

WSClock-Algorithmus (2)

- Fall 1: Mindestens eine Seite wurde vorgemerkt
  - (Einige) vorgemerkte Seiten werden in der Zwischenzeit auf Festplatte ausgelagert, M-Bit gelöscht ⇒ Seite wird ausgetauscht



- Fall 2: Keine vorgemerkte Seite
  - Alle Seiten werden gebraucht ⇒ aktuelle Seite wird ausgelagert und die Kachel für die einzulagernde Seite verwendet



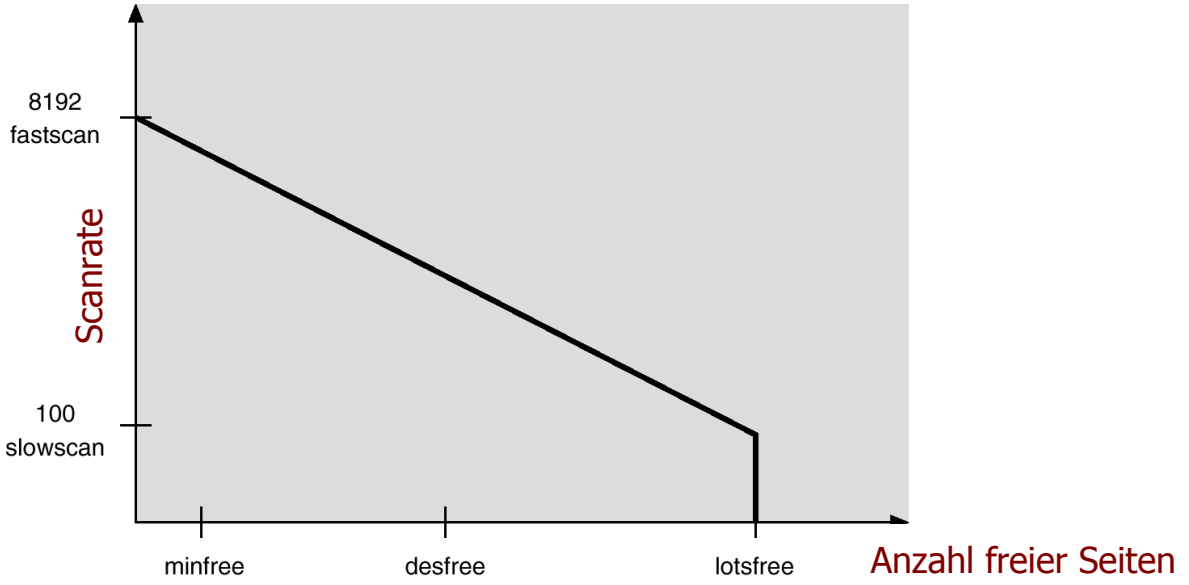
Paging-Daemon

- Technik zur Sicherung eines ausreichenden Vorrats an freien Kacheln für schnelle Reaktion auf weitere Speicheranforderungen
- Grundidee
  - Trennung von Seitenverdrängung und Seiteneinlagerung
  - Speicherverwaltung hält eine vorab festgelegte Anzahl an Kacheln frei

  ⇒ Seitenfehler können ohne Verdrängung beseitigt und neue Adressräume direkt angelegt werden
- Realisierung
  - Paging-Daemon wird periodisch aktiviert (Üblicher Wert: alle 250 ms)
  - Überprüfung, ob die a-priori vorgegebene Anzahl freier Kachel unterschritten wurde (Üblicher Wert: ca. 25% der Kachel frei)
    - Falls ja ⇒ Auslagerung von Seiten auf die Festplatte gemäß der eingesetzten Verdrängungsstrategie
    - Falls nein ⇒ Blockierung des Paging-Daemons bis zum nächsten Sollzeitpunkt

Beispiel: Speicherverwaltung in Solaris 2

- Realisierung eines Paging Daemons
- Drei wichtige Parameter
  - Lotsfree: mindestens 1/4 des Speichers frei, Überprüfung alle 0.25 Sek
  - Desfree: Fällt die Anzahl freier Seite unter diesen Schwellwert (Durchschnitt über 30 Sek) ⇒ Start von Swapping, Überprüfung alle 0.1 Sek
  - Minfree: Nur noch minimale Menge freier Seiten vorhanden ⇒ Aufruf der Seitenersetzung bei jeder Speicheranforderung

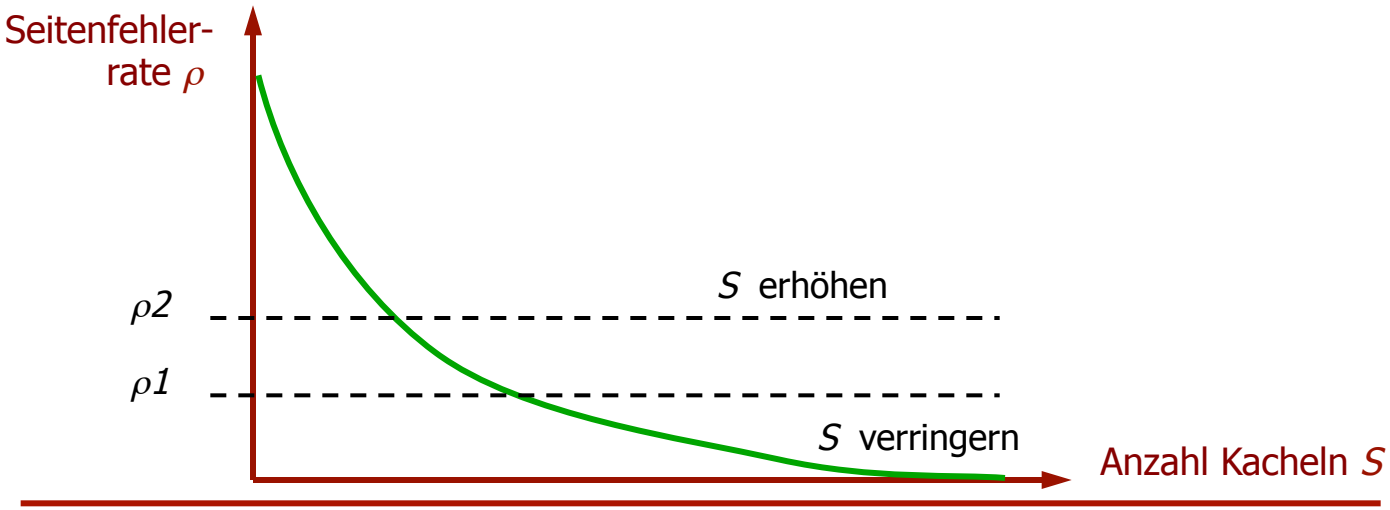


Solaris 2: Algorithmus

- Speicherverwaltung mit dem Prozess pageout
  - Variante des Clockalgorithmus (Zweizeigeralgorithmus, two-handed-clock)
  - Erster Zeiger scannt die Seiten und setzt die Referenzbits auf 0
  - Später läuft der zweite Zeiger nach und gibt alle Seiten mit R immer noch 0 wieder frei
- Wichtige Parameter
  - Scanrate: Scangeschwindigkeit (Seiten/Sekunde)
    - Anpassung an aktuellen Systemzustand (slowscan = 100 Seiten/s bis fastscan = Gesamtanzahl von Seiten/2 aber max = 8192)
  - Handspread: Statischer Abstand in Seiten zwischen den Zeigern
    - Tatsächlicher Abstand ergibt sich durch Kombination von scanrate und handspread, z.B. scanrate = 100 und handspread = 1024 $\Rightarrow$ 10 Sek zwischen den beiden Zeigern
    - Bei höher Belastung Abstände von einigen Tausendstel nicht ungewöhnlich
- Erweiterung: Überspringe Seiten von Shared Libraries

Page-Fault-Frequency-Modell

- Seitenfehlerrate = Anzahl Seitenfehler / Zeiteinheit
- Für jeden Prozess wird die Seitenfehlerrate ρ gemessen
  - Einführung von zwei Schwellwerten ρ_1 (obere Intervallgrenze) und ρ_2 (untere Intervallgrenze)
  - Einstellung der Kachelanzahl S in Abhängigkeit von Seitenfehlerrate durch
    - Verringere Anzahl Kacheln, falls $\rho < \rho_1$, d.h. $S = S - 1$
    - Erhöhe Anzahl Kacheln, falls $\rho > \rho_2$, d.h. $S = S + 1$

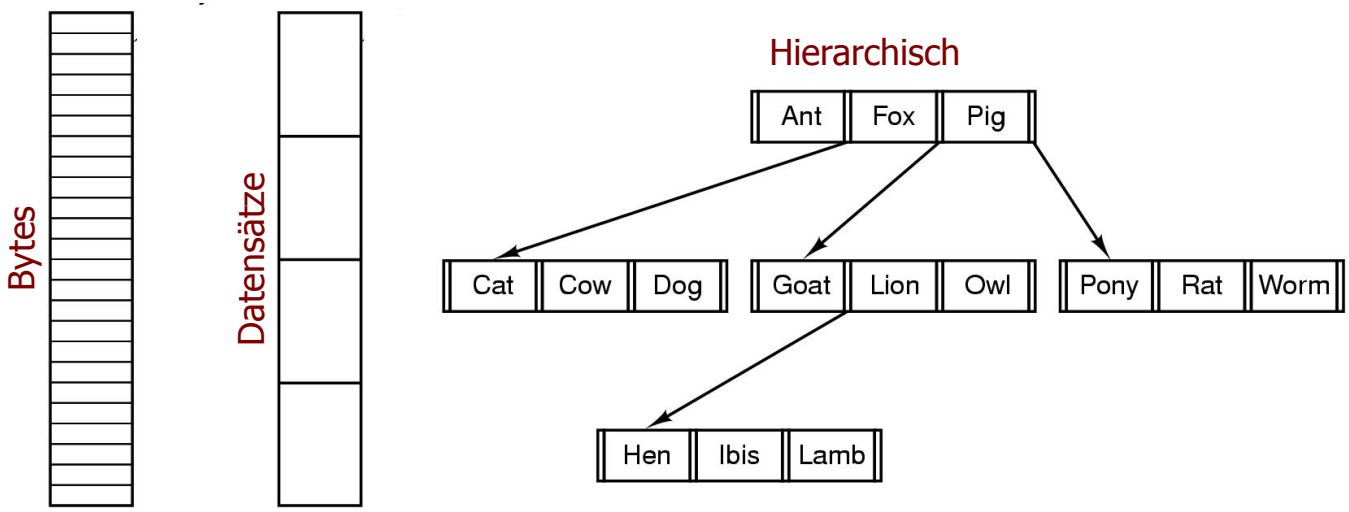


6.4 Dateisysteme

- Dateisystem ist das meist sichtbare Konzept eines Betriebssystems
  - Mechanismus für online Speicherung und Zugriff auf Daten, Programme, … für alle Benutzer
  - Allgemein bekannte Schlüsselwörter
    - Datei (*file*): Sammlung von Daten auf einem stabilen Speicher
    - Verzeichnisse (*directory, folder*): Beinhalten eine Sammlung von Dateien und sind selbst in Verzeichnisstrukturen organisiert
    - Partitionen: Physische oder logische Aufteilung der Verzeichnisstrukturen in kleinere Teilmengen

Organisationsform der Dateien

- Dateiform: Textdateien (Daten zeilenweise eingelesen) oder Objektdateien (Bytesequenz)
- Strukturierung
  - Lineare Bytefolge: Struktur der Datei wird durch die Anwendung bestimmt und beim Einlesen interpretiert
  - Sequenz von Linien/Datensätzen: Einlesen/Überschreiben von zusammengesetzten Strukturen fester Größe
  - Hierarchisch: Jeder Datensatz wird über einen Schlüssel identifiziert und anschließend eingelesen

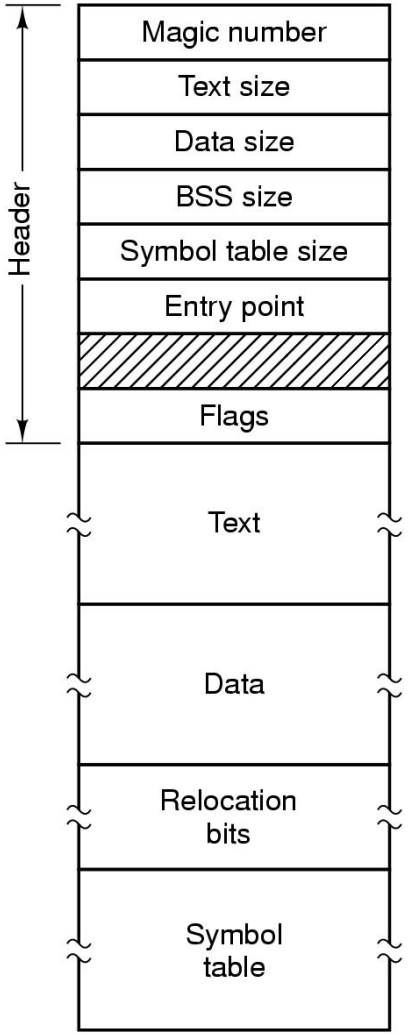


Dateiattribute (Auswahl)

- Name: Symbolischer Bezeichner, aussagekräftig für Benutzer und üblicherweise zweigeteilt
  1. Benutzervergebener Name der Datei
  2. Dateierweiterung: Hinweis zu Dateiinhalt und Zuordnung zu einem Verarbeitungsprogramm
- Bezeichner: Eindeutige – üblicherweise numerische – Beschreibung der Datei im Dateisystem
- Dateityp, z.B. bei UNIX reguläre Dateien (alphanumerisch oder binär), Verzeichnisse, spezielle Gerätedateien …
- Position: Zeiger auf den Speicherplatz der Datei (Gerät, Position)
- Größe: Aktuelle Dateigröße in Bytes, Wörtern oder Blöcken
- Benutzer- und Zugriffsinformationen: Wer darf lesen / schreiben / ausführen, …
- Zeit und Datum: Erstellung, letzter Zugriff, letzte Modifikation, …

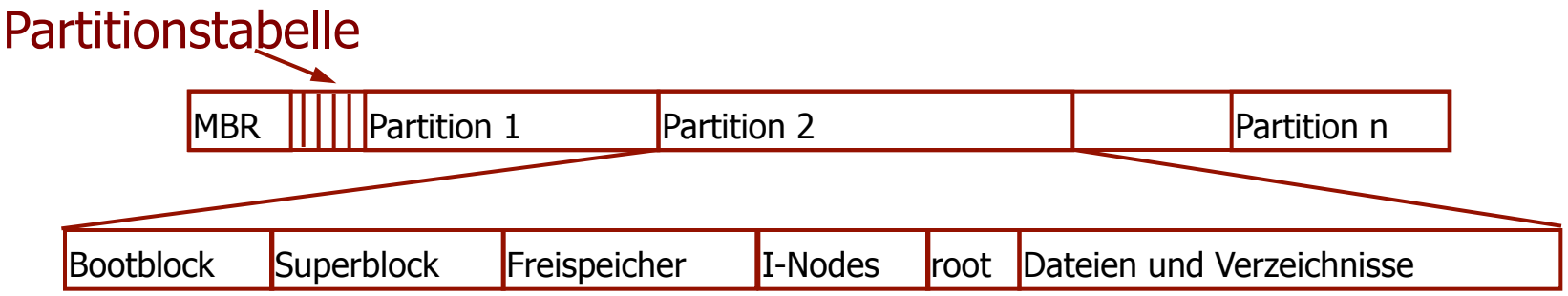
Dateitypen: Beispiel ausführbare Datei

- Binäre Dateien nur dann ausführbar, wenn sie das richtige Format haben
- Identifizierung durch Header (siehe /usr/share/magic)
  - Magic Number: Datei ist ausführbar
  - Größenangaben der unterschiedlichen Dateibereiche (Text, Daten, …)
  - Einstiegspunkt: An welcher Stelle soll die Ausführung des in der Datei enthaltenen Codes beginnen?
- Kern der Datei
  - Codeinformationen
  - Statische Daten
  - Relokationsbits zur Anordnung von Code und Daten im Hauptspeicher
  - Symboltabelle, falls Debugging notwendig



Layout eines Dateisystems

- Aufteilung der Festplatten in Partitionen
- Sektor 0 der Festplatte: MBR (*Master Boot Record*)
  - Lokalisierung der aktiven Partition beim Booten
  - Verwendung der Partitionstabelle (Start/End-Adresse aller Partitionen, aktive Partition)
- Bootblock: Erster Block einer jeden Partition
  - Bootblock der aktiven Partition wird ausgeführt und lädt das Dateisystem
  - Superblock: Schlüsselparameter des Dateisystems (Magische Nummer zur Identifizierung, Anzahl Blöcke, administrative Informationen)
  - Liste der freien Blöcke
  - Verwaltungsstrukturen für die existierenden Dateien und Verzeichnisse

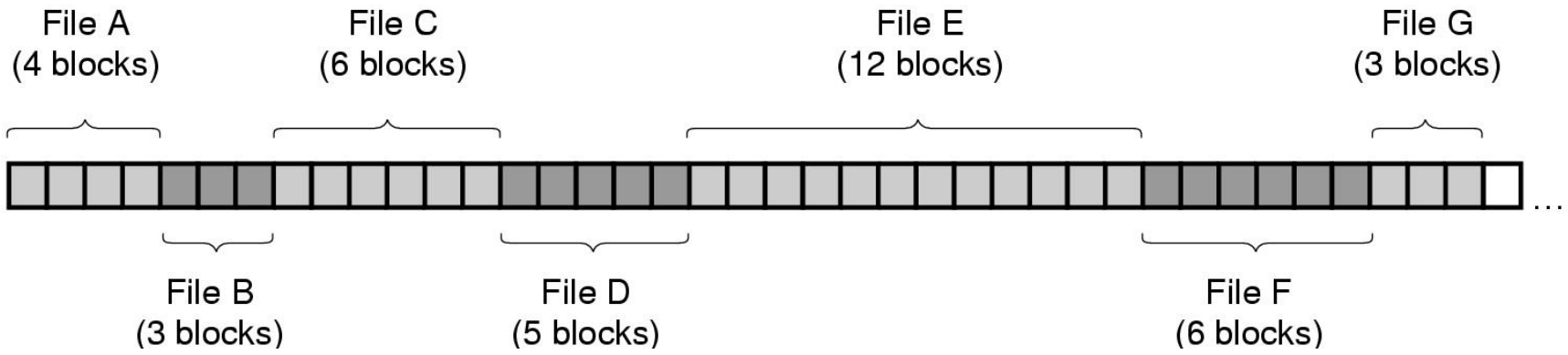


Wichtige Merkmale eines Dateisystems

- Physische Struktur
  - Abbildungsschema auf die Datenträgergeometrie
    - ⇒Abbildung der Dateien auf die physischen Objekte des Speichermediums, wie Sektoren, Zylinder und Plattenoberflächen
  - Zugriffs-/Modifikationsalgorithmen (Öffnen, Schreiben, Positionieren, ...)
- Logische Struktur
  - Darstellung des Dateisystems gegenüber einem Benutzer
  - Anordnung von Dateien in einer Verzeichnisstruktur
  - Dateibezeichner und Metainformationen, Zugriffsberechtigungen
- Zwei-Phasen-Realisierung
  - Abbildung Dateien → Fortlaufend nummerierte Blöcke fester Länge
  - Abbildung Datenblöcke → spezifische Geometrie des Speichermediums
    - Hohe Zugriffsgeschwindigkeit durch geeignete Blockpositionierung
    - Minimierung der Wahrscheinlichkeit von Datenverlusten im Fehlerfall

Zusammenhängende Zuordnung Dateien → Datenblöcke

- Einfachste Abbildung: Zusammenhängende Belegung
  - Dateiblöcke gleicher Größe werden direkt hintereinander abgelegt

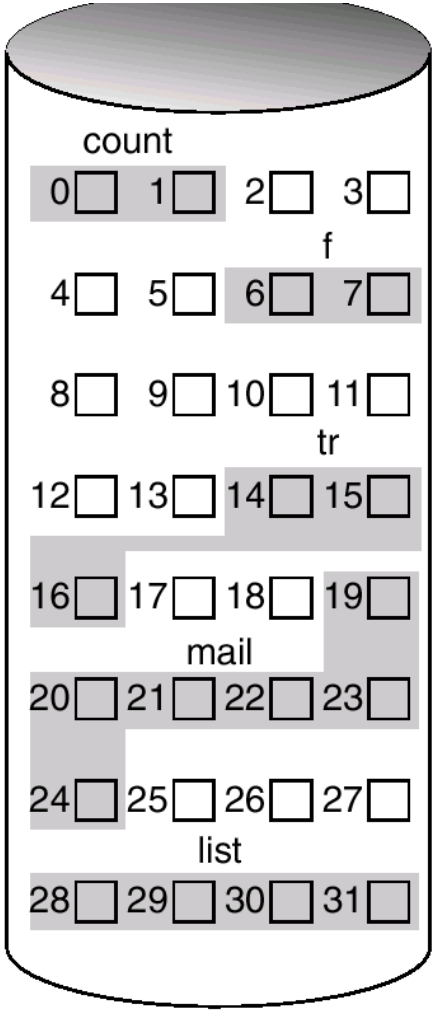


- Vorteile
  - Dateizugriff mit maximaler Effizienz ⇒ Lesen einer Datei mit einer kontinuierlichen Operation
  - Direkter Zugriff auf einzelne Blöcke möglich
  - Einfache Verwaltung, da lediglich die Länge und die Indexnummer des ersten Datenblocks bekannt sein müssen

Zusammenhängende Zuordnung Dateien → Datenblöcke (2)

- Nachteile
  - Fragmentierung des Datenträgers durch Löschen/Anlegen von Dateien unterschiedlicher Länge
  - Probleme, wenn mehrere Dateien zum Schreiben offen gehalten werden müssen

  ⇒ Nur in speziellen Fällen brauchbar
- Wiederentdeckung durch Beschreiben von CD-ROMs / DVDs / Blu-Rays

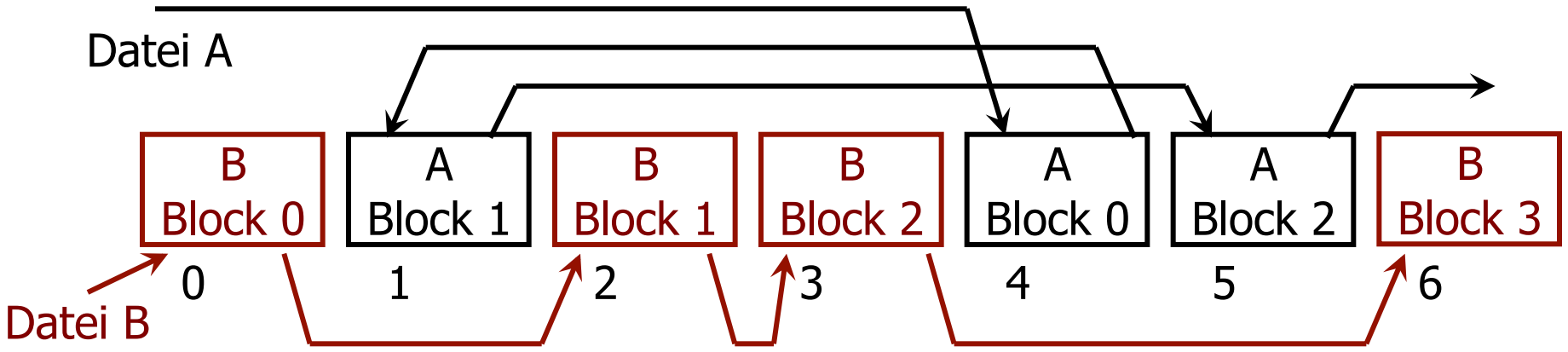


directory

file	start	length
count	0	2
tr	14	3
mail	19	6
list	28	4
f	6	2

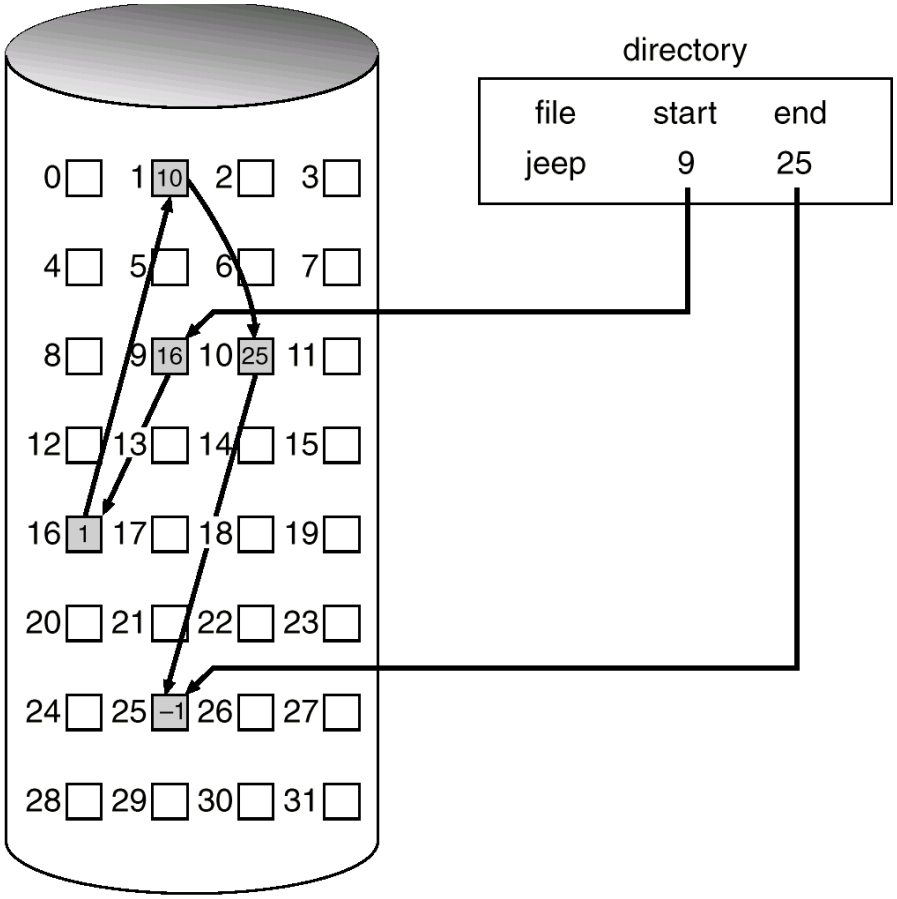
Belegung durch verkettete Listen

- Dateien können beliebig auf die Datenblöcke verteilt werden
- Zuordnung erfordert geeignete Datenstrukturen
  - Interne Verkettung
  - Externe Verkettung
  - Indexblöcke
- Interne Verkettung ohne Verwaltungsdatenblöcke
  - Startblock ist direkt im Dateiverzeichnis abgelegt
  - Jeder Block verweist auf seinen Nachfolger
  - Ausschließlich sequentielles Lesen möglich



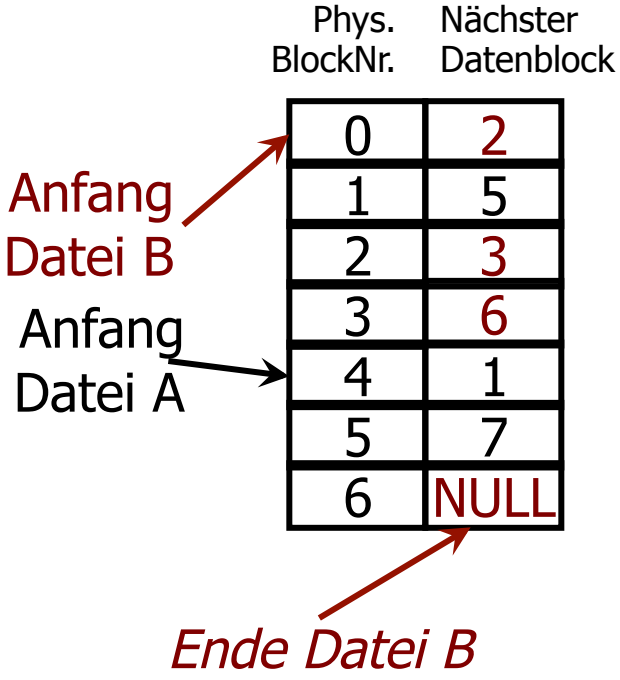
Belegung durch verkettete Listen (2)

- Vorteile
  - Keine externe Fragmentierung, Größenvorgabe nicht nötig
  - Einfaches Anlegen von Dateien (Ein Verzeichniseintrag)
- Nachteile
  - Teil des Blocks reserviert für Zeiger ⇒ interne Fragmentierung
  - Ineffizienter Zugriff: Auswertung vorhergehender Blöcke beim Zugriff notwendig
  - Zuverlässigkeit: Dateiverlust durch Zerstörung eines Blocks. Doppeltverkettung vergrößert die interne Fragmentierung
- Abhilfe: Gruppierung mehrerer Blöcke in Cluster



Externe Verkettung mit Hilfstabelle

- Einsatz bei MS-DOS: FAT = File Allocation Table
- Eigenschaften der Allokationstabelle
  - Je eine Zeile pro Block des externen Speichers mit Verweis auf Adresse des nächsten Blocks
  - FAT wird an fest reservierter Stelle einer Partition gehalten und laufend aktualisiert
  - Startblock der Datei wird im Dateiverzeichnis abgelegt
  - Dateiende durch NULL-Zeiger angezeigt

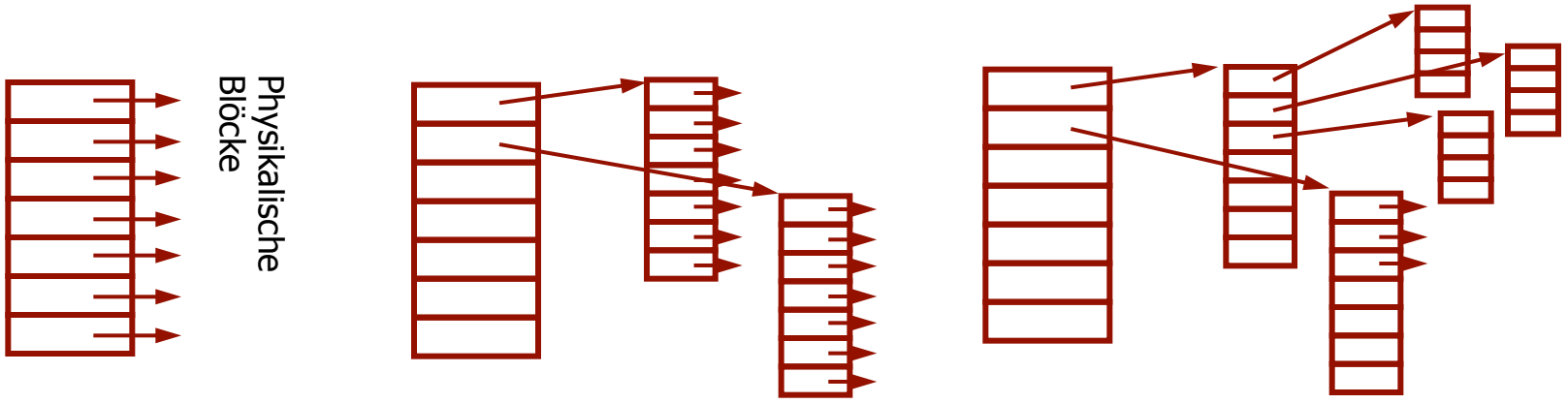


- Nachteile
  - Ineffizienter Zugriff, da bei jedem Zugriff ein Lesevorgang für die FAT notwendig ist ⇒ Sprung an Partitionsanfang
  - Redundante FAT-Sicherung, um Datenverluste zu vermeiden

Indexblöcke

- Dateispezifisches Feld von Indizes zur Identifikation der belegten Blöcke
  - Jeder Datei wird eine Datenstruktur (Indexblock) zugeordnet
  - Indexblock ist nur dann im Speicher, wenn die Datei geöffnet ist
- Ein Verzeichniseintrag verweist auf den Indexblock
  - Der Indexblock wird auf Festplatte gespeichert
  - Ein Indexblock reicht nicht aus, um alle Indizes einer langen Datei aufzunehmen ⇒ Erweiterung zur Verwendung mehrerer Indexblöcke
    - Indizes verweisen auf weitere Indexblöcke statt auf Datenblöcke

    ⇒Baumstruktur analog zum Aufbau mehrstufiger Seitentabellen



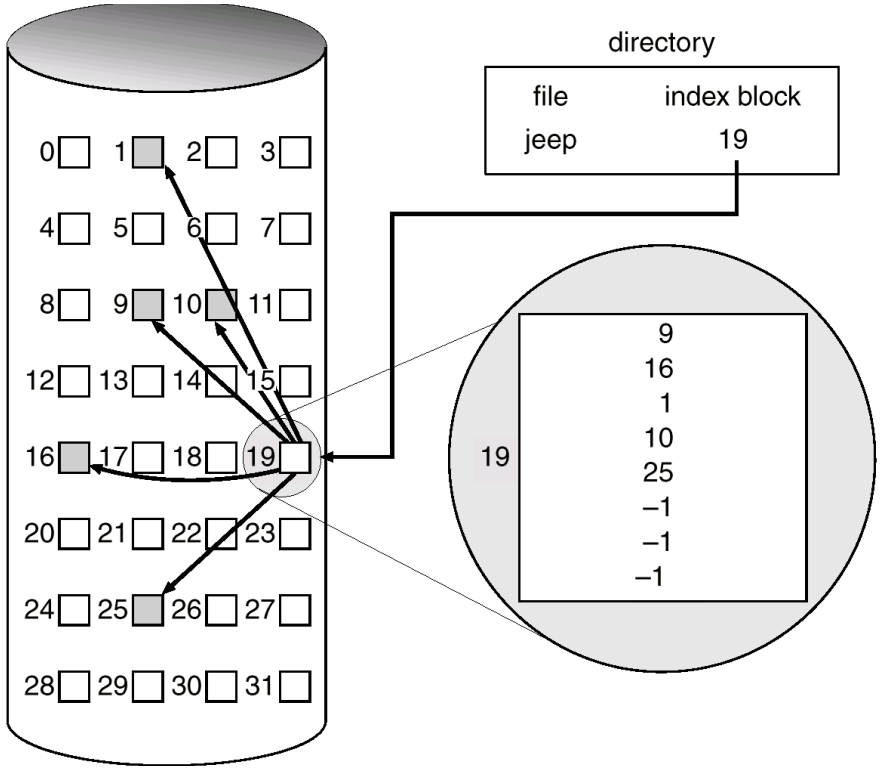
Indexblöcke (2)

- Zugriffsschema bei Indexblöcken
  - Lese den Indexblock
  - Lese die *i* -te Zeile für den Zugriff auf den *i* -ten Block aus
  - Folge den Zeiger zur Blockposition

- Vorteil: Speicherplatzersparnis
  - Tabelle zur Speicherung der verketteten Liste aller Blöcke wächst linear mit Plattengröße

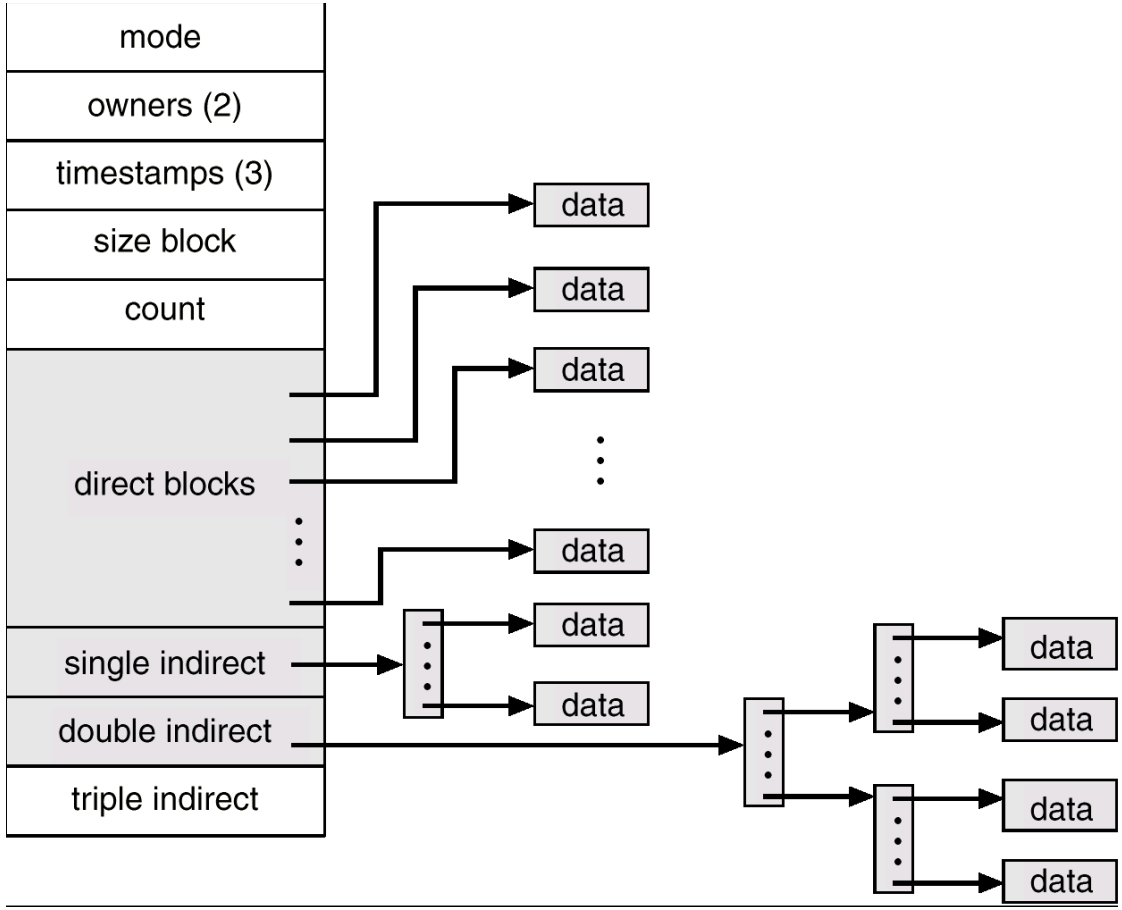
  ⇒ Festplatte mit *n* Blöcken benötigt maximal *n* Einträge

  - Bei Indexblöcken wächst die Größe des Arrays proportional zur maximalen Anzahl von gleichzeitig geöffneten Dateien, unabhängig von der Plattengröße



Indexblöcke (3)

- Indexblöcke heißen bei UNIX I-Nodes
- Kombiniertes Schema
  - Direct blocks: Zeiger auf Daten ⇒ Zugriff auf den Dateianfang ohne Umwege
  - Single indirect: Verweis auf Block mit Adressen der weiteren Blöcke
  - Double indirect: Zwei Zwischenstufen, …



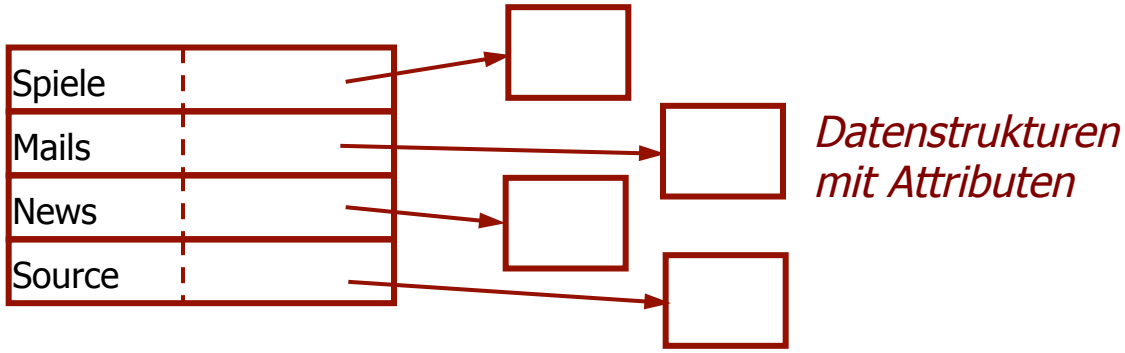
Realisierung von Verzeichnissen

- Verzeichniseinträge speichern die Position mindestens des ersten Dateiblocks und evtl. weitere Attribute (Eigentümer, Entstehungszeit, Zugriffsmodus, …)
  1. Einfache Organisation: Alle Informationen werden im Verzeichniseintrag gespeichert ⇒ Verzeichnis ist Liste von Einträgen (z.B. MS-DOS)
  2. I-Node-basierte Verzeichnisse haben kürzere Einträge, die den Dateinamen und die Nummer des I-Nodes aufnehmen (UNIX)

1

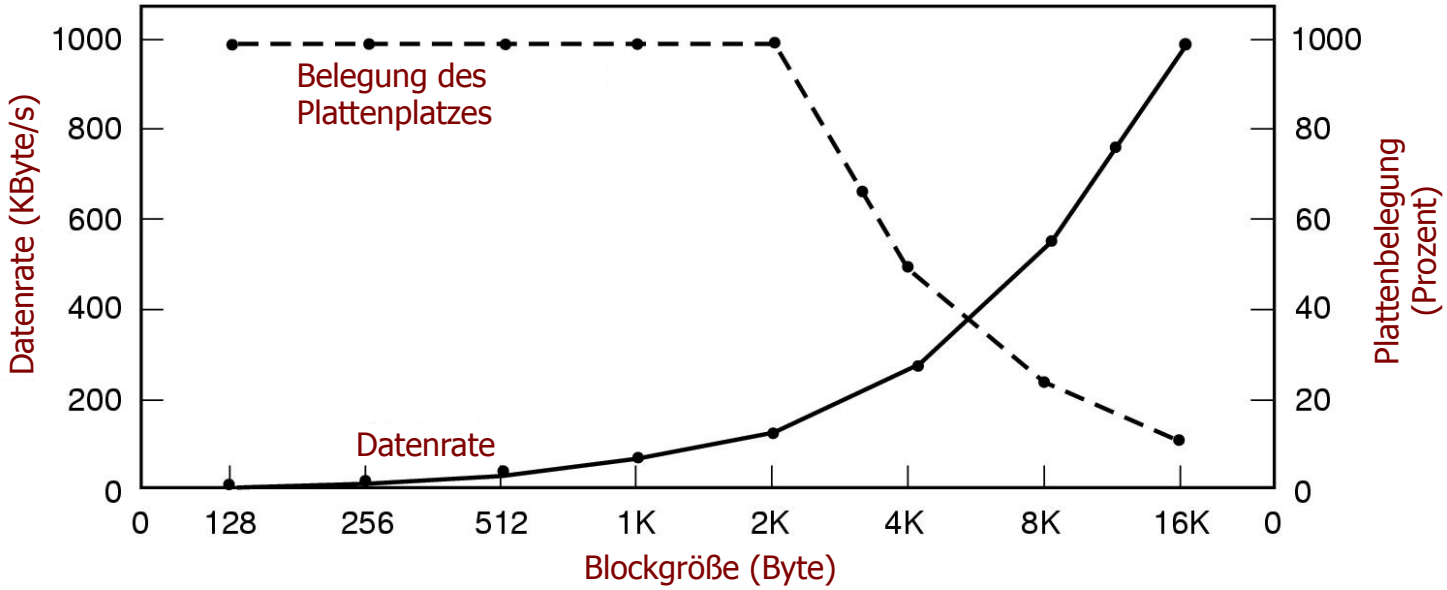
Spiele	Attribute
Mails	Attribute
News	Attribute
Source	Attribute

2



Verwaltung des Plattenspeichers

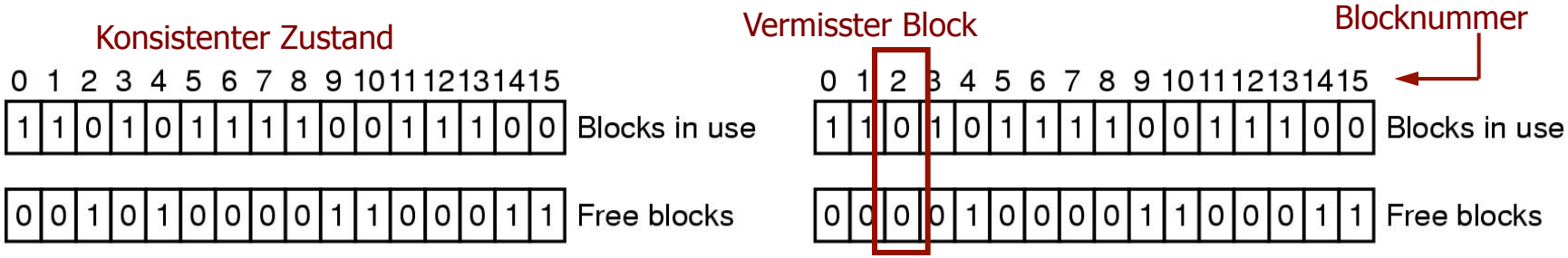
- Ausgangssituation: Fast alle Dateisysteme teilen die Dateien in Blöcke fester Größe auf
- Wie groß soll ein solcher Block sein?
  - Spuren, Sektoren, Zylinder ⇒ geräteabhängig, ungeeignet
  - Kompromiss zwischen Zugriffsgeschwindigkeit und Fragmentierung
  - Beispiel Datenrate/Platzbelegung als Funktion von der Blockgröße bei einer mittleren Dateigröße von 1680 Bytes



Zuverlässigkeit von Dateisystemen: Konsistenz

- Konsistenzüberprüfung
  - Vergleich der Daten in den Verzeichnissen mit den Datenblöcken auf der Festplatte und ggf. Behebung der Inkonsistenzen
  - Kritisch bei I-Nodes, Verzeichnisblöcken, Listen freier Segmente
  - Realisierung mit Hilfsprogrammen wie scandisk, fsck, …
  - Überprüfung von Dateien und Blöcken

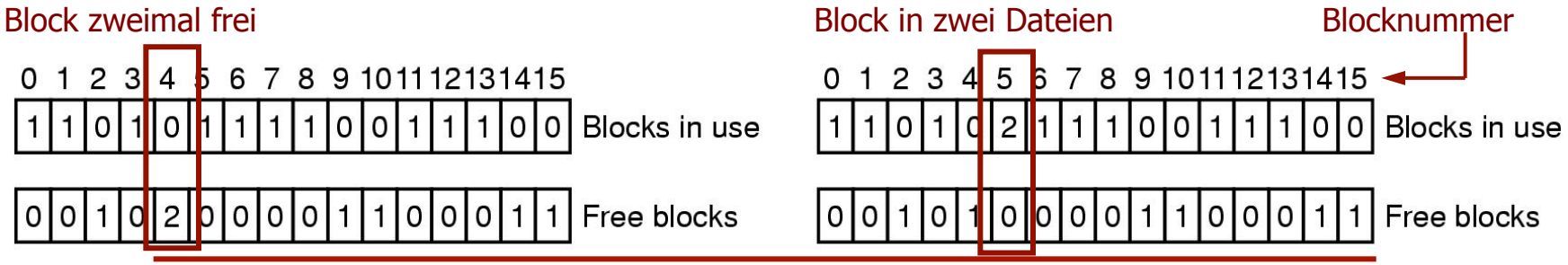
- Vorgehensweise bei Blöcken
  - Zwei Tabellen mit Zählern, wie häufig ein Block vorkommt in
    - Einer Datei
    - Der Liste mit freien Segmenten
  - Auslesen aller I-Nodes und Überprüfung aller Blöcke
  - Bei Konsistenz ist jeder Block in genau einer Tabelle



Konsistenzüberprüfung (2)

- Was kann passieren?
  - Vermisster Block: Speicherplatzverschwendung ⇒ Freiliste aktualisieren
  - Datenblock doppelt als frei ausgewiesen ⇒ Freiliste aktualisieren
  - Block kommt in zwei Dateien vor ⇒ kritischer Fall
    - Wird eine Datei gelöscht, so wird der Block als frei gekennzeichnet
    - Block wird kopiert ⇒ beide Dateien haben eigenen Block
    - Meldung an Benutzer, da eine Datei evtl. durcheinander gebracht wurde

- Verzeichnisüberprüfung
  - Analoger Zähler für Dateien statt Blocks
  - Rekursive Untersuchung des Verzeichnisbaums und der I-Nodes
  - Bei Konsistenz stimmen beide Zähler überein
  - Fehler: Überflüssige I-Nodes oder doppelte Einträge



Log-basierte Dateisysteme

- Aktuelle Situation
  - CPU-Geschwindigkeit steigt sehr schnell an
  - Vergrößerung der Kapazitäten von Speicher und Caches
  - Festplatten werden immer größer, Geschwindigkeitszuwachs kann jedoch mit dem CPU-Zuwachs nicht mithalten ⇒ Schreibzugriffe immer kürzer im Vergleich zum Verwaltungsaufwand

  ⇒ Bedeutung der Caches und der Konsistenzprobleme wird immer größer
- Entwicklung von log-basierten Dateisystemen (auch transaktions-basierte oder Journaling Dateisysteme)
  - Verwendung des Transaktionskonzepts und der Recoverymechanismen
- Grundidee
  - Gesamte Festplatte wird als ein Logbuch strukturiert
  - Schreibzugriffe werden verzögert und auf einmal – unter Ausnutzung der vollen Festplattenbandbreite – geschrieben

Log-basierte Dateisysteme (2)

- In periodischen Abständen/bei Erreichen einer Größe: Schreiben der Segmente (I-Nodes, Verzeichnis-, Dateiblöcke) ans Logbuchende
  - Am Segmentanfang: Zusammenfassung über den Segmentinhalt
  - I-Nodes: gleiche Funktion/Struktur, verstreut über das gesamte Logbuch
  - Nach Auffinden des I-Nodes: Auslesen der Blöcke wie gewohnt
  - Zur schnelleren Auffindung der I-Nodes ⇒ I-Node-Map
- Thread Cleaner durchsucht das Logbuch und räumt es auf
  - Liest das erste Segment des Logbuchs aus und vergleicht die I-Nodes mit der I-Node-Map
    - Noch aktive Elemente werden in den Speicher übertragen und bei der nächsten Aktualisierung auf die Platte geschrieben
    - Eintragungen bereits gesicherter Elemente werden gelöscht
    - Zirkulärer Puffer: Schreiber fügt vorne Segmente, Cleaner entfernt alte Segmente
- Buchführung keineswegs trivial ⇒ ständige Aktualisierung der I-Nodes und der I-Node-Map notwendig
- Untersuchungen: Bedeutende Verbesserung bei kleinen Schreibzugriffen