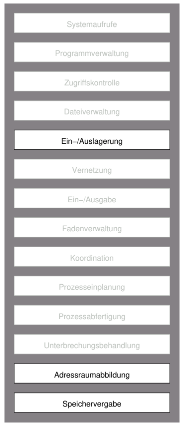
Speicherverwaltung



- Speicher als $\left\{\begin{array}{c}\text{physikalisches}\\ \text{logisches}\\ \text{virtuelles}\end{array}\right\}$ **Betriebsmittel**
  - gekapselt in dazu korrespondierenden Adressräumen
- Vergabe zur *Ladezeit* und ggf. zur *Laufzeit*
  - statisch/dynamisch aus Programmsicht („von oben“)
  - dynamisch aus Betriebssystemsicht („von unten“)
- in den meisten Fällen ein „knappes Betriebsmittel“

Adressräume (☞ Kap. 4)

physikalischer Adressraum definiert einen nicht-linear adressierbaren, von Lücken durchzogenen Bereich von E/A-Schnittstellen und Speicher ($*$RAM, $*$ROM), dessen Größe der Adressbreite der CPU entspricht

☞ 2^N Bytes, bei einer Adressbreite von N Bits

logischer Adressraum definiert einen linear adressierbaren Speicherbereich, dessen Größe selten der Adressbreite der CPU entspricht[52]

☞ 2^M Bytes, $M < N$

virtueller Adressraum ein logischer Adressraum, der 2^N Bytes umfasst

[52] Bei einer *Harvard-Architektur* (getrennter Programm- und Datenspeicher) kann $M = N$ gelten.

Physikalischer Adressraum

Adressbereich	Größe (KB)	Verwendung
`00000000–0009ffff`	640	RAM (System)
`000a0000–000bffff`	128	Video RAM
`000c0000–000c7fff`	32	BIOS Video RAM
`000c8000–000dffff`	96	keine
`000e0000–000effff`	64	BIOS Video RAM (*shadow*)
`000f0000–000fffff`	64	BIOS RAM (*shadow*)
`00100000–090fffff`	147456	RAM (Erweiterung)
`09100000–fffdffff`	4045696	keine
`fffe0000–fffeffff`	64	SM-RAM (*system management*)
`ffff0000–ffffffff`	64	BIOS ROM

Toshiba Tecra 730CDT, 1996

Adressen, die ins Leere verweisen . . . „*Bus Error*“

Adressbereich	Größe (KB)	Verwendung
`00000000–0009ffff`	640	RAM (System)
`000a0000–000bffff`	128	Video RAM
`000c0000–000c7fff`	32	BIOS Video RAM
`000c8000–000dffff`	96	keine
`000e0000–000effff`	64	BIOS Video RAM (*shadow*)
`000f0000–000fffff`	64	BIOS RAM (*shadow*)
`00100000–090fffff`	147456	RAM (Erweiterung)
`09100000–fffdffff`	4045696	keine
`fffe0000–fffeffff`	64	SM-RAM (*system management*)
`ffff0000–ffffffff`	64	BIOS ROM

Toshiba Tecra 730CDT, 1996

Adressen, die reserviert sind . . . *„Protection Fault“*

Adressbereich	Größe (KB)	Verwendung
`00000000–0009ffff`	640	RAM (System)
`000a0000–000bffff`	128	Video RAM
`000c0000–000c7fff`	32	BIOS Video RAM
`000c8000–000dffff`	96	keine
`000e0000–000effff`	64	BIOS Video RAM (*shadow*)
`000f0000–000fffff`	64	BIOS RAM (*shadow*)
`00100000–090fffff`	147456	RAM (Erweiterung)
`09100000–fffdffff`	4045696	keine
`fffe0000–fffeffff`	64	SM-RAM (*system management*)
`ffff0000–ffffffff`	64	BIOS ROM

Toshiba Tecra 730CDT, 1996

Adressen, die Programmen zugewiesen werden können . . .

Adressbereich	Größe (KB)	Verwendung
`00000000–0009ffff`	640	RAM (System)
`000a0000–000bffff`	128	Video RAM
`000c0000–000c7fff`	32	BIOS Video RAM
`000c8000–000dffff`	96	keine
`000e0000–000effff`	64	BIOS Video RAM (*shadow*)
`000f0000–000fffff`	64	BIOS RAM (*shadow*)
`00100000–090fffff`	147456	RAM (Erweiterung)
`09100000–fffdffff`	4045696	keine
`fffe0000–fffeffff`	64	SM-RAM (*system management*)
`ffff0000–ffffffff`	64	BIOS ROM

Toshiba Tecra 730CDT, 1996

Logischer Adressraum

- ein *Programmadressraum* wird in (mind.) drei Abschnitte logisch aufgeteilt:

Maschinenanweisungen, Programmkonstanten	**Text**	**-segment**
initialisierte Daten globale Variablen, Halde	**Daten**	
lokale Variablen, Hilfsvariablen, aktuelle Parameter	**Stapel**	

- die *Memory Management Unit* setzt logische in physikalische Adressen um
  - je nach Rechnerarchitektur ist die MMU ein Teil oder Koprozessor der CPU
  - eine MMU verwendet Segmente oder/und Seiten als Verwaltungseinheiten
- das Betriebssystem bildet logische auf gültige physikalische Adressen ab

Adressraumausprägungen

eindimensional ☞ in *Seiten* aufgeteilt . . . . (*paged*)

- der Prozessor interpretiert eine Programmadresse A_P als Tupel (p, o)

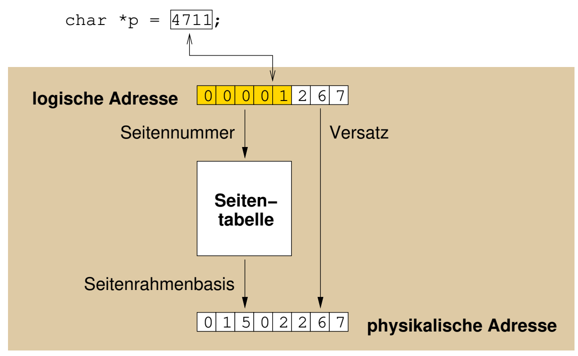
Seitennummer (*page*)	$p = A_P \ \mathtt{div}\ 2^N$	2^N ist Seitengröße in Bytes
Versatz (*offset*)	$o = A_P \ \mathtt{mod}\ 2^N$	

- Seiten werden abgebildet auf (physikalische) *Seitenrahmen*, auch *Kacheln*

zweidimensional ☞ in *Segmente* aufgeteilt . . . . (*segmented*)

- der Prozessor definiert eine Programmadresse A_S als Paar (S, A)
  - sind die Segmente gekachelt, dann wird A als A_P interpretiert
- Segmente werden abgebildet auf einen (phys.) eindimensionalen Adressraum

Adressumsetzung — Gekachelter Adressraum

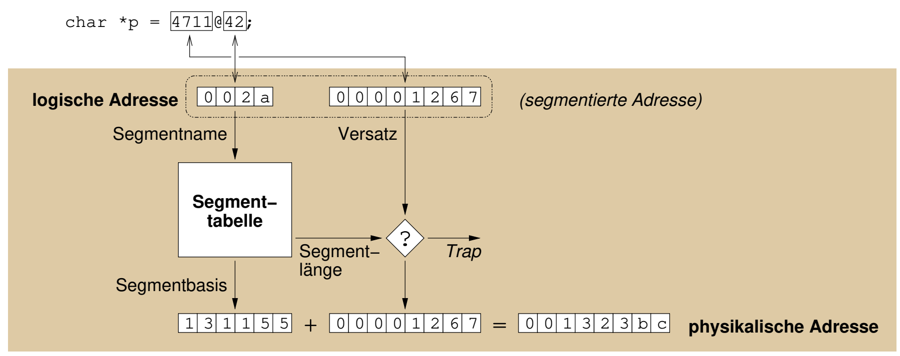


4711_{10} = 0x1267 (in C)

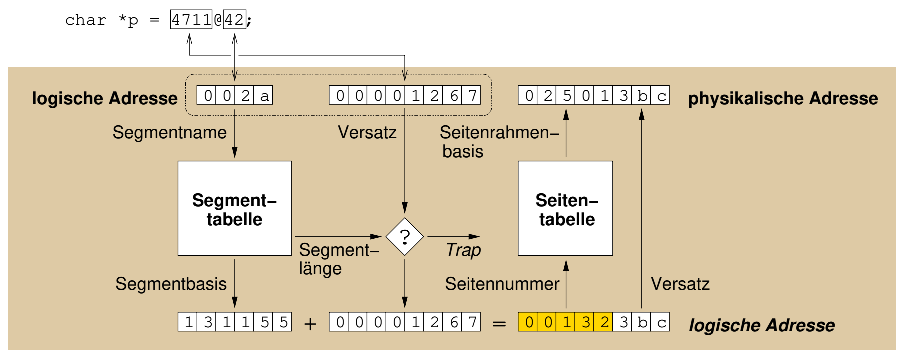
Eine *Seitennummer* ist ein Index in eine (pro Prozess vorhandende) *Seitentabelle*. Der dem Index entsprechende **Seitendeskriptor** enthält die Basisadresse des Seitenrahmens im physikalischen Adressraum.

☞ **Relokation**

Adressumsetzung — Segmentierter Adressraum



Adressumsetzung — Segmentiert-gekachelter Adressraum



Attribute von Seiten/Segmenten (1)

Für jede Seite bzw. jedes Segment existiert ein (Seiten-/Segment) **Deskriptor**, der Relokations- und Zugriffsdaten prozessbezogen verwaltet:

- die *physikalische Basisadresse* des Seitenrahmens/Segments im Hauptspeicher
- die *Zugriffsrechte* des Prozesses: lesen (*read*), schreiben (*write*)

Segmentdeskriptor Segmente sind (im Gegensatz zu Seiten) von variabler, dynamischer Größe; als zusätzliche Verwaltungsdaten fallen an:

- die *Segmentlänge*, um Segmentverletzungen abfangen zu können
- die *Expansionsrichtung*: Halde „*bottom-up*“, Stapel „*top-down*“

Lage und Ausdehnung von Seiten-/Segmenttabellen

***base*/*limit*-Registerpaar** definiert die Anfangsadresse (*base*) einer Tabelle und die Anzahl (*limit*) der Tabelleneinträge

- bei der Adressumsetzung wird eine *Indexprüfung* wie folgt durchgeführt:

  $descriptor = (index \leq limit)\ ?\ \&base[index] : \textbf{\textit{trap this process}}$

- wobei *index* eine Seitennummer oder einen Segmentnamen repräsentiert

Die Inhalte dieser **Prozessorregister** gehören zum *Prozesszustand*:

- initial bestimmt zur Ladezeit von Programmen ☞ `exec(2)`
- aktualisiert zur Laufzeit der Programme ☞ `brk(2)`

Virtueller Adressraum

- Abstraktion von der Größe <u>und</u> Örtlichkeit des verfügbaren Arbeitsspeichers
  - vom Prozess nicht benötigte Programmteile können ausgelagert sein
    ☞ sie liegen im *Hintergrundspeicher*, z. B. auf der Festplatte
  - der Prozessadressraum könnte über ein Rechnernetz verteilt sein
    ☞ Programmteile sind über die Arbeitsspeicher anderer Rechner verstreut
- Zugriffe auf nicht eingelagerte Programmteile fängt der Prozessor ab: *Trap*
  - sie werden stattdessen *partiell interpretiert* vom Betriebssystem
  - das Betriebssystem zwingt den unterbrochenen Prozess in einen E/A-Stoß
  - die Wiederaufnahme des CPU-Stoßes führt zur Wiederholung des Zugriffs

Attribute von Seiten/Segmenten (2)

- je nach Konzept sind Segmente und/oder Seiten von der Einlagerung betroffen

  ☞ $\left\{ \begin{array}{r} \text{segmentierte} \\ \text{segmentiert-gekachelte} \\ \text{gekachelte} \end{array} \right\}$ virtuelle Adressräume

- das „*present bit*“ im (Segment/Seiten) **Deskriptor** regelt die Zugriffsart:

  1 → eingelagert; Instruktion lesen, Operanden lesen/schreiben
  0 → ausgelagert, *Trap*; partielle Interpretation des Zugriffs, Einlagerung

Seiten-/Segmentfehler

present bit $= 0$ je nach Befehlssatz und Adressierungsarten der CPU kann der **Behandlungsaufwand** und **Leistungsverlust** beträchtlich sein

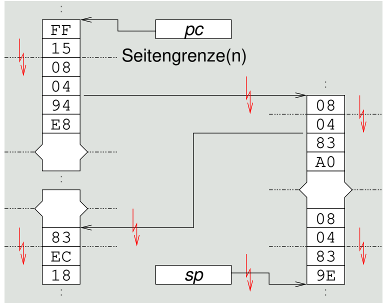
```
void hello () {
    printf("Hi!\n");
}

void (*moin)() = &hello;

main () {
    (*moin)();
}
```

```
main:
    pushl %ebp
    movl  %esp,%ebp
    pushl %eax
    pushl %eax
    andl  $-16,%esp
    call  *moin
    leave
    ret
    :
```

```
:
FF15080494E8
:
```

Fallstudie: Seitenfehler (1) „GAU“

`call *moin` (x86) Prozeduraufruf, indirekte Adressierung des Unterprogramms über einen Zeiger („*pointer to function returning void*“)



Bis zu sieben Seitenfehler können bei der Ausführung dieses einen Befehls auftreten:

1. Operandenadresse holen (08 04 94 E8)
2. Funktionszeiger lesen (08)
3. Funktionszeiger weiterlesen (04 83 A0)
4. Rücksprungadresse stapeln (08 04)
5. Rücksprungadresse weiterstapeln (83 9E)
6. Operationskode holen (83)
7. Operanden holen (EC 18)

[Welche Maßnahmen beugen potentiellen Seitenfehlern im Fall von 3., 5. und 7. vor?]

Fallstudie: Seitenfehler (2) Aufwandsabschätzung

effektive Zugriffszeit (*effective access time*, eat) auf ein Speicherdatum, ist stark abhängig von der *Seitenfehler<u>wahrscheinlichkeit</u>* (p) und verhält sich direkt proportional zur *Seitenfehlerrate*: $eat = (1-p) \cdot pat + p \cdot pft, 0 \leq p \leq 1$

Angenommen, folgende Systemparameter sind gegeben:

- 50 ns Zugriffszeit auf den RAM (*physical access time*, pat)
- 10 ms mittlere Zugriffszeit auf eine Festplatte (*page fault time*, pft)
- 1 % Wahrscheinlichkeit eines Seitenfehlers ($p = 0,01$)

Dann ergibt sich: $eat = 0,99 \cdot 50\,\text{ns} + 0,01 \cdot 10\,\text{ms} = 49,5\,\text{ns} + 10^5\,\text{ns} \approx 0,1\,\text{ms}$

- ein Einzelzugriff wäre im Seitenfehlerfall um den Faktor 2000 langsamer !!!

Fallstudie: Seitenfehler (3) „Herzklopfen kostenlos“

mittlere Zugriffszeit (*mean access time*, mat) auf den Arbeitsspeicher (RAM) hängt sehr stark ab von der *effektiven Zugriffszeit* auf eine Seite und der *Seitengröße*: $mat = (eat + (sizeof(page) - 1) \cdot pat)/pat$

Angenommen, folgende Systemparameter sind gegeben:

- Seitengröße von 4 096 Bytes (4 KB)
- 50 ns Zugriffszeit (pat) auf ein Byte im RAM
- effektive Zugriffszeit (eat) wie eben berechnet bzw. abgeschätzt

Dann ergibt sich: $mat = (eat + 4\,095 \cdot 50\,\text{ns})/50\,\text{ns} = 6\,095,99\,\text{ns} \approx 6\,\mu\text{s}$

- Folgezugriffe wären im Seitenfehlerfall im Ø um den Faktor 122 langsamer !!!

Pro und Contra

Virtuelle Adressräume sind . . .

vorteilhaft wenn „übergroße“ bzw. gleichzeitig/nebenläufig viele Programme in Anbetracht zu knappen Arbeitsspeichers auszuführen sind

ernüchternd wenn der eben durch die Virtualisierung bedingte Mehraufwand zu berücksichtigen ist und sich für ein gegebenes Anwendungsszenario als problematisch bis unakzeptabel erweisen sollte

☞ jedem sollte klar sein, was etwas kostet . . .

Speicherzuteilung

statisch Benutzerprogrammen <u>und</u> Betriebssystem sind Gebiete maximaler, fester Größe zugewiesen

- innerhalb eines Gebiets kann Speicher jedoch dynamisch vergeben werden
- Probleme: Brachliegen von Betriebsmitteln, Leistungsbegrenzung/-verluste
  - ☞ ungenutzter Speicher eines Gebiets ist in anderen Gebieten nicht nutzbar
  - ☞ begrenzte E/A-Bandbreite mangels Puffer im Betriebssystemgebiet
  - ☞ erhöhte Wartezeit von Prozessen wegen zu kleinen Puffern

dynamisch das Betriebssystem ermittelt „Segmente" angeforderter Größe im Arbeitsspeicher und teilt diese den Benutzerprogrammen bzw. sich selbst zu

Verschnitt . . .

Mischungen von Wein, Most oder Trauben zu unterschiedlichen Zwecken.

Abfall der etwa beim Zuschneiden kleinerer Flächen (z. B. Fensterleibungen) von Mustertapeten entsteht.

Hohlräume die beim Verpacken kleinerer Einheiten (Kisten) in größere Einheiten (Container) entstehen.

Rest der bei der Speicherzuteilung anfällt, wenn zur Erfüllung einer Anforderung gegebener Größe ein zu großes Segment Verwendung findet.

☞ interne vs. externe Fragmentierung

Politiken bei der Speicherzuteilung

Platzierungsstrategie (*placement policy*) <u>wohin</u> die Information ablegen?

- wo der Verschnitt am kleinsten, am größten bzw. zweitrangig ist?

Ladestrategie (*fetch policy*) <u>wann</u> ist Information zu laden?

- auf Anforderung oder im Voraus?

Ersetzungsstrategie (*replacement policy*) <u>welche</u> Information ist zu verdrängen?

- die älteste, am seltensten genutzte oder am längsten ungenutzte?

Platzierungsstrategie

- verwaltet nicht-zugeteilten Speicher, definiert die **Freispeicherorganisation**:

  Bitkarte (*bit map*) freier Bereiche <u>fester</u> Größe
  - eignet sich für die Verwaltung gekachelter Adressräume
  - grobkörnige Vergabe freien Speichers auf Seitenrahmenbasis

  verkettete Liste (*free list*) freier Bereiche <u>variabler</u> Größe
  - ist typisch für die Verwaltung segmentierter Adressräume
  - feinkörnige Vergabe freien Speichers auf Segmentbasis

  ☞ freie Bereiche bilden „Löcher", die mit Programmen/Daten auffüllbar sind

- segmentierter Speicher ist aufwändiger zu verwalten als gekachelter
  - Löcher auf Rahmenbasis sind alle gleich gut, auf Segmentbasis nicht

Freispeicherkarte

- ein *Bit* repräsentiert den Verfügbarkeitszustand einer jeden Verwaltungseinheit:

1	$\rightarrow$	frei	☞ verfügbar
0	$\rightarrow$	belegt	

Einheit $\equiv N$ Bytes in Folge, fest

- bei gekachelten Adressräumen entspricht die Einheit einem Seitenrahmen
  - die Zuteilung erfolgt *rahmenweise*, jeder freie Seitenrahmen passt
  - die MMU linearisiert die Rahmen zu einem eindimensionalen Adressraum

Verschnitt (*interne Fragmentierung*): Anforderungsgröße $<$ Zuteilungsgröße N

Umfang von Bitkarten

- die Erfassung freien Speichers beansprucht mehr oder weniger viel Speicher
  - z. B. ein System mit 1 GB Hauptspeicher und 4 KB Seitengröße
  - die dazu passende Bitkarte hat eine Größe von 32 KB:

$$\begin{aligned} sizeof(bit\ map) &= 1\,\text{GB} \div 4\,\text{KB} \div 8\,\text{Bits} \\ &= 2^{30} \div 2^{12} \div 2^{3} = 2^{15}\,\text{Bytes} \end{aligned}$$

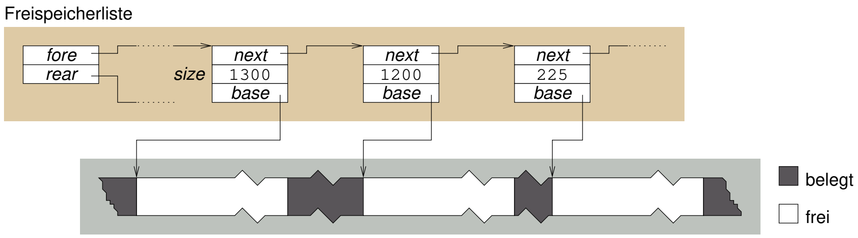
  - die Kartengröße variiert (bei gleich großem Speicher) mit der Seitengröße
  - manche MMUs erlauben die Programmierung/Einstellung dieses Parameters
- je feinkörniger die Speicherzuteilung, desto speicherintensiver die Bitkarte

Freispeicherliste

hole list („Löcherliste“) führt Buch über freie, zuteilbare Speicherbereiche

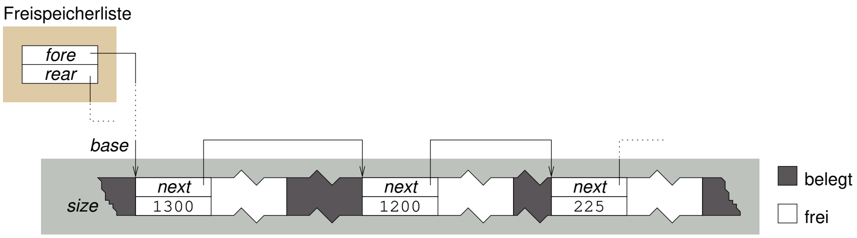
- jedes Listenelement enthält *Anfangsadresse* und *Länge* des freien Bereichs
  - wodurch jeweils ein *Speichersegment* beschrieben wird
- für die Liste ergeben sich zwei grundsätzliche Repräsentationsformen:
  1. Liste und Löcher sind voneinander getrennt
     - die Listenelemente sind Löcherdeskriptoren, sie belegen Betriebsmittel
     - Löcher haben eine beliebige Größe N, $N > 0$
  2. Liste wird in den Löchern geführt
     - die Listenelemente sind die Löcher, sie belegen keine Betriebsmittel
     - Löcher haben eine Mindestgröße N, $N \geq sizeof(list\ element)$
- *strategische Entscheidungen* bestimmen die Art der Listenverwaltung

Listenelemente als Löcherdeskriptoren



- die Liste und der Listenkopf liegen im Betriebssystemadressraum
  - *fore*, *rear* und *next* sind logische/virtuelle Adressen
  - *base* ist eine Bytenummer, die für eine physikalische Adresse steht
- Listenmanipulationen laufen wie gewohnt innerhalb eines Adressraums ab

Listenelemente als Löcher



- die Liste liegt im physikalischen, der Listenkopf im Betriebssystemadressraum
  - *fore*, *rear*, *next* und *base* sind physikalische Adressen
  - die Operationen darauf laufen jedoch im Betriebssystemadressraum ab
- Listenmanipulationen müssen ggf. Adressraumgrenzen überschreiten [Warum ggf.?]

Zuteilungsverfahren (1)

best-fit verwaltet Löcher nach aufsteigenden Größen

- den *Verschnitt minimieren*, d. h., das kleinste passende Loch suchen
- erzeugt kleine Löcher am Listenanfang, erhält große Löcher am Listenende
- der Suchaufwand nimmt zu

worst-fit verwaltet Löcher nach absteigenden Größen

- den *Suchaufwand minimieren*, d. h., das vorderste Loch verwenden
- zerstört große Löcher am Listenanfang, erzeugt kleine Löcher am Listenende
- hinterlässt eher große Löcher

Ist die angeforderte Größe kleiner als das gefundene Loch, fällt Verschnitt an, der als verbleibendes Loch in die Liste neu einsortiert werden muss.

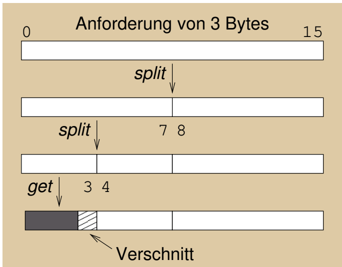
Zuteilungsverfahren (2)

buddy („Kamerad“, „Kumpel“) verwaltet Löcher nach aufsteigenden Größen von 2er-Potenzen

- das kleinste passende Loch (☞ *buddy*$_i$) der Größe 2^i suchen
  - i ist Index in eine Tabelle von Adressen freier *buddies* der Größe 2^i
  - *buddy*$_i$ wird durch (sukzessive) Splittung von *buddy*$_{j,\, j>i}$ gewonnen:
    - ☞ $2^n = 2 \times 2^{n-1}$
    - ☞ zwei gleichgroße Blöcke, die *buddy* des jeweils anderen sind
- der Verschnitt kann beträchtlich sein: bei $2^i + 1$ angeforderten Bytes, $2^i - 1$
  - jeder Verschnitt ist jedoch als Summe freier *buddies* darstellbar, wie auch jede Dezimalzahl als Summe von 2er-Potenzen
- ein Kompromiss zwischen *best-fit* und *worst-fit*

Zuteilungsverfahren (2)

buddy (Forts.) Zuteilung (☞ Splittung) aber auch Rückgabe (☞ Verschmelzung) sind effizient realisierbar



- zwei freie Blöcke können verschmolzen werden, wenn sie *buddies* sind
  - die Adressen von *buddies* unterscheiden sich nur in einer Bitposition
- zwei Blöcke 2^i sind *buddies*, wenn sich ihre Adressen in Bitposition i unterscheiden

Je nach MMU kann der Verschnitt als freier *buddy* entsprechend seiner Größe in die Tabelle/Liste eingetragen werden oder er ist als Verlust zu verbuchen.

Zuteilungsverfahren (3)

first-fit verwaltet Löcher nach aufsteigenden Adressen

- den *Verwaltungsaufwand minimieren*, d. h., die Liste nicht umsortieren
- erzeugt kleine Löcher am Listenanfang, erhält große Löcher am Listenende
- der Suchaufwand nimmt zu

next-fit die *round-robin* Variante von *first-fit*

- den *Suchaufwand minimieren*, d. h., Start beim zuletzt zugeteiltem Loch
- nähert sich einer Liste/Verteilung von „gleichgroßen Löchern“
- der Suchaufwand nimmt ab

Da Adressen und nicht Größen das Sortierkriterium darstellen, muss die Liste (im Gegensatz zu *best-fit* und *worst-fit*) nur einmal durchlaufen werden.

Verschmelzung

Speicherfreigabe bedeutet nicht nur die Rückgabe eines Betriebsmittels, sondern auch die Vereinigung kleiner Löcher zu einem großen Loch

- für einen zur Freigabe bestimmten Bereich ergeben sich vier Lagen:
  1. zwischen zwei zugeteilten Bereichen
  2. direkt nach einem Loch ☞ Vereinigung mit Vorgänger
  3. direkt vor einem Loch ☞ Vereinigung mit Nachfolger
  4. zwischen zwei Löchern ☞ Kombination von 2. und 3.
- der Vereinigungsaufwand variiert mit dem Zuteilungsverfahren
  - klein bei *buddy*, mittel bei *first/next-fit*, groß bei *best/worst-fit*
- Löchervereinigung verringert die *externe Fragmentierung* des Speichers

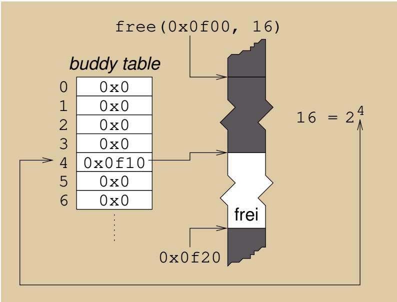
Verschmelzung vs. Zuteilungsverfahren

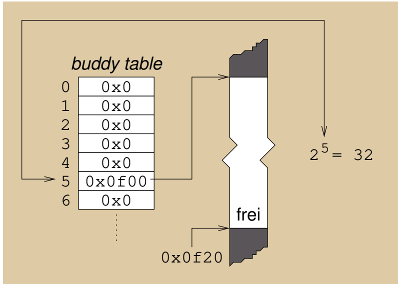
buddy anhand eines Bits der Adresse des freigegebenen Bereichs lässt sich leicht feststellen, ob sein *buddy* als Loch in der Tabelle verzeichnet ist

first/next-fit beim Durchlaufen der Freispeicherliste wird geprüft, ob Adresse plus Größe eines Lochs der Adresse des freigegebenen Bereichs bzw. ob Adresse plus Größe des freigegebenen Bereichs der Adresse eines Lochs entspricht

best/worst-fit ähnlich wie bei *first/next-fit*, jedoch kann im Gegensatz dazu nicht davon ausgegangen werden, dass bei einem angrenzenden Loch das Vorgänger- bzw. Nachfolgerelement in der Liste das ggf. andere angrenzende Loch sein muss ☞ es muss weitergesucht werden

Verschmelzung (1) ***buddy***

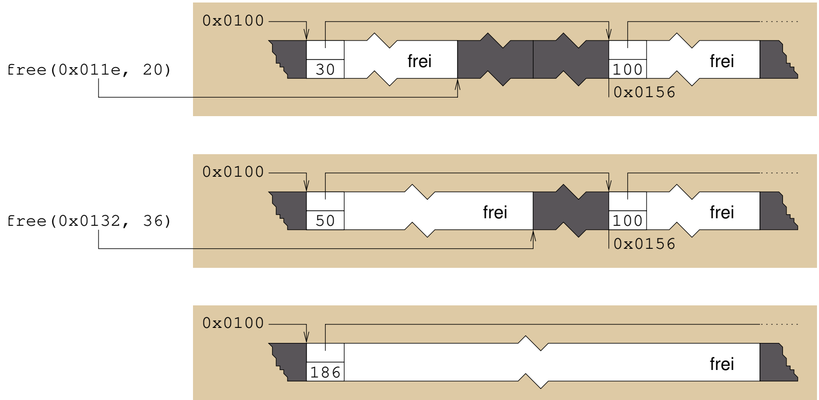




$$\begin{aligned} 0f00_{16} &= 0000\,1111\,0000\,0000_2 \\ 0f10_{16} &= 0000\,1111\,0001\,0000_2 \\ 16_{10} &= 0000\,0000\,0001\,0000_2 \end{aligned}$$

$$\begin{aligned} 0f00_{16} &= 0000\,1111\,0000\,0000_2 \\ 0f20_{16} &= 0000\,1111\,0010\,0000_2 \\ 32_{10} &= 0000\,0000\,0010\,0000_2 \end{aligned}$$

Verschmelzung (2) *first/next-fit*



Fragmentierung

(lat.) *Bruchstückbildung*; Zerstückelung des Speichers in immer kleinere, verstreut vorliegende Bruchstücke:

interne Fragmentierung *Verschwendung*, Problem gekachelten Speichers

- Speicher wird in Einheiten gleicher, fester Größe (☞ Seiten) vergeben
  - eine angeforderte Größe muss kein Vielfaches der Seitengröße sein
  - als Folge kann am Seitenende ein Bruchstück (Verschnitt) entstehen
- der „lokale Verschnitt“ kann, dürfte aber nicht genutzt werden [Weshalb?]

externe Fragmentierung *Verlust*, Problem segmentierten Speichers

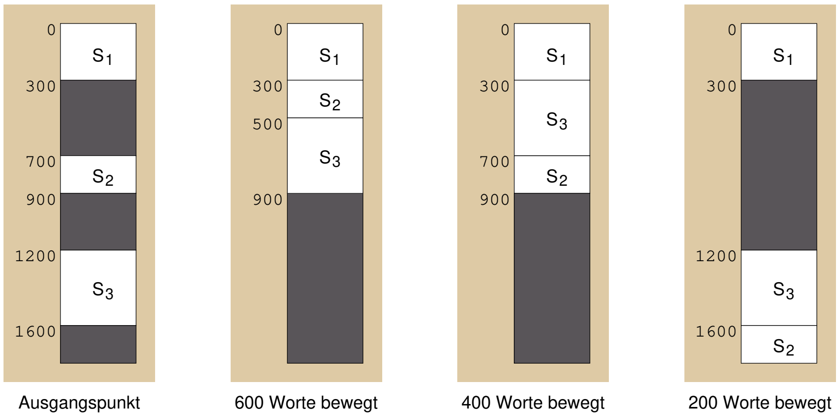
- Speicher wird in Einheiten angeforderter Größe (☞ Segmente) vergeben
- anhaltender Betrieb durchzieht den gesamten Speicher mit Bruchstücken
- der „globale Verschnitt“ ist wegen ggf. zu kleiner Bruchstücke unbrauchbar

Kompaktifizierung (1)

- Auflösung externer Fragmentierung durch Vereinigung globalen Verschnitts
  - Segmente von (Bytes oder Seitenrahmen) werden so verschoben, dass am Ende ein einziges großes Loch übrigbleibt
  - alle in der Freispeicherliste erfassten Löcher werden sukzessive verschmolzen, so dass schließlich nur noch ein Listenelement existiert
  - Aus-/Einlagerung (*swapping*) von Segmenten unterstützt den Kopiervorgang
- **Relokation** der verschobenen Segmente ist erforderlich

  ☞ logische/virtuelle Adressräume, positionsunabhängiger Programmtext
- ein je nach Fragmentierungsgrad komplexes *Optimierungsproblem* . . .

Kompaktifizierung (2)



Ladestrategie

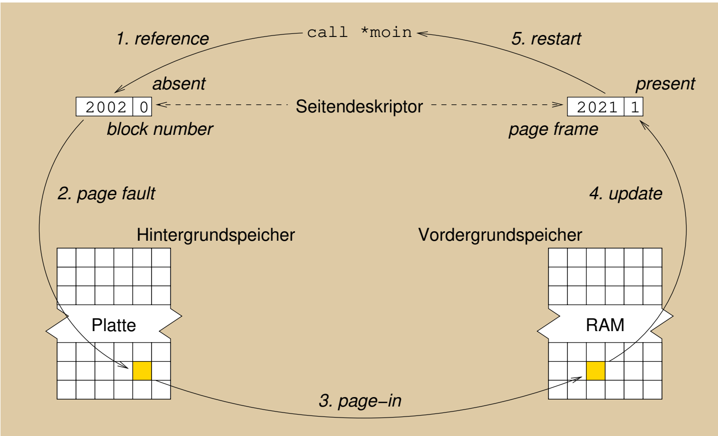
- entscheidet, wann und wie die **Einlagerung** von Seiten/Segmenten erfolgt:

  Einzelanforderung (*on demand*) ☞ *present bit* ✗ 11-15
  - Seiten-/Segmentfehler (*page/segment fault*) führt zur *Trap*-Behandlung
  - Behandlungsergebnis ist die Einlagerung der angeforderten Speichereinheit

  Vorausladen (*anticipatory*)
  - *Heuristiken* können Hinweise über zukünftige Zugriffsmuster liefern
    ☞ Programmlokalität, Arbeitsmenge (*working set*)
  - alternativ als **Vorabruf** (*prefetch*) im Zuge einer Einzelanforderung

- ggf. fällt die *Verdrängung* von Seiten/Segmenten an ☞ Ersetzungsstrategie

Einzelanforderung — „*on demand paging*“



Vorabruf

- *Vorbeugung nachfolgender Seitenfehler* — der „`call *moin`“-GAU (✗ 11-17):
  1. den unterbrochenen Befehl dekodieren, die Adressierungsart feststellen
  2. da der Operand die Adresse einer Zeigervariablen (`moin`) ist, den Adresswert auf Überschreitung einer Seitengrenze prüfen
  3. da der Befehl die Rücksprungadresse stapeln wird, die gleiche Überprüfung mit dem Stapelzeiger durchführen
  4. in der Seitentabelle die entsprechenden Deskriptoren lokalisieren und prüfen, ob die Seiten anwesend sind: jede abwesende Seite ist einzulagern
  5. da jetzt die Zeigervariable (`moin`) vorliegt, sie dereferenzieren und ihren Wert auf Überschreitung einer Seitengrenze prüfen
  6. wie 4.

- *partielle Interpretation* des unterbrochenen Befehls durch das Betriebssystem

Ersetzungsstrategie

- entscheidet über die **Verdrängung** von Seiten (aber auch Segmenten):

  FIFO (*first-in, first-out*) die zuerst eingelagerten
  - Seiten entsprechend ihres Einlagerungszeitpunkts linear verketten

  LFU (*least frequently used*) die am seltensten genutzten
  - jeden Seitenzugriff zählen, **MFU** (*most frequently used*) als Alternative

  LRU (*least recently used*) die kürzlich am wenigsten genutzten
  - auf „logische Uhrzeiten“, Stapeltechniken oder Referenzlisten setzen
  - bzw. weniger aufwändig mit Hilfe von Referenzbits approximieren

- Ursache ist *Speicherüberbelegung*: der Speicherbedarf aller Prozesse ist größer als der verfügbare physikalische Programm-/Datenspeicher (RAM)

Hauptziel von Ersetzungsverfahren

☞ *Seitenfehlerwahrscheinlichkeit* senken und *Seitenfehlerrate* verringern

optimales Verfahren (OPT) Verdrängung der Seite, die am längsten nicht mehr verwendet werden wird

- dies erfordert jedoch Wissen über die im weiteren Verlauf der Ausführung von Prozessen generierten Speicheradressen
  - das Laufzeitverhalten von Prozessen müsste vorhergesagt werden
  - auch mit exaktem Wissen über Eingabewerte ist dies kaum/nicht möglich
    ☞ asynchrone Programmunterbrechungen (✗ 5-29)
- bestenfalls ist eine „gute" *Approximation* möglich und praktisch umsetzbar

LFU Zählverfahren MFU

- für jede Seite wird mitgezählt, wie häufig sie referenziert worden ist:
  - im Seitendeskriptor ist dazu ein *Referenzzähler* enthalten
  - der Zähler wird mit jeder Referenz zu der Seite inkrementiert
  - aufwändige Implementierung, bei eher schlechter Approximation von OPT

LFU ersetzt die Seite mit dem kleinsten Zählerwert

- „aktive Seiten" haben üblicherweise große Zählerwerte
- ebenso wie die aktiv gewesenen Seiten z. B. der Initialisierungsphase

MFU ersetzt die Seite mit dem größten Zählerwert

- „kürzlich aktive Seiten" haben eher kleine Zählerwerte

LRU Grundsätzliche Verfahren

LRU$_{time}$ verwendet einen Zähler („logische Uhr") in der CPU, der bei jedem Speicherzugriff erhöht und in den jeweiligen Seitendeskriptor geschrieben wird
☞ verdrängt die Seite mit dem kleinsten Zählerwert

LRU$_{stack}$ nutzt einen Stapel eingelagerter Seiten, aus dem bei jedem Seitenzugriff die betreffende Seite herausgezogen und wieder oben drauf gelegt wird
☞ verdrängt die Seite am Stapelboden

LRU$_{ref}$ führt Buch über alle zurückliegenden Seitenreferenzen (*reference string*)
☞ verdrängt die Seite mit dem größten *Rückwärtsabstand*

- entspricht OPT, wenn die Vergangenheit (nicht die Zukunft) betrachtet wird
  - gute Approximation von OPT, bei sehr aufwändiger Implementierung

LRU Approximation (1)

page aging verwendet pro Seitendeskriptor ein **Referenzbit** (*reference bit*), das bei Einlagerung bzw. jedem Seitenzugriff auf 1 gesetzt wird

- das Alter eingelagerter Seiten wird periodisch (☞ Zeitgeber) bestimmt:
  - für jede eingelagerte Seite wird ein N-Bit Zähler (*age counter*) geführt
  - der Zähler funktioniert als *Shiftregister* zur Aufnahme von Referenzbits
  - nach Aufnahme eines Referenzbits in den Zähler, wird es gelöscht
- „kürzlich am wenigsten genutzte" Seiten haben einen Zählerwert von 0
  - d. h., sie wurden seit N Perioden nicht mehr referenziert
- ein TLB[53] vermeidet die zusätzlichen Speicherreferenzen (*caching*)

[53] *Translation Lookaside Buffer*, eine Tabelle von Kreuzverweisen zwischen virtuellen und physikalischen Adressen kürzlich referenzierter Seiten bzw. Seitenrahmen, inkl. Referenzbit. Der TLB ist Bestandteil des Prozessors.

LRU — Approximation (2)

page aging (Forts.) mit Ablauf eines Zeitquantums (Tick), werden Referenzbits eingelagerter Seiten in die Shiftregister übernommen

- z. B. ein 8-Bit „*age counter*“: $age = (age >> 1) \,|\, (ref << 7)$

Referenzbit	Alter (*age*, initial 0)
1	10000000
1	11000000
0	01100000
1	10110000
⋮	⋮

Den Inhalt des 8-Bit Schiftregisters (*age*) als ganze Zahl intepretiert liefert ein Maß für die Aktivität bzw. Ersetzungspriorität einer Seite: mit abnehmender Wertigkeit der Aktivität steigt die Ersetzungspriorität.

- Aufwand steigt mit der Adressraumgröße des unterbr. Prozesses ☞ UNIX

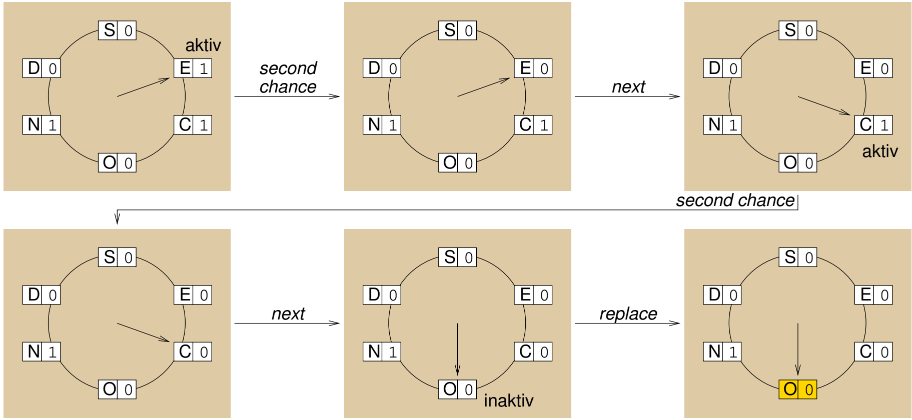
LRU — Approximation (3)

clock policy (*second chance*) ein Ersetzungsverfahren, das im Grunde nach FIFO arbeitet, jedoch zusätzlich noch die Referenzbits der jeweils in Betracht zu ziehenden Seiten berücksichtigt

Referenzbit	Aktion	Bedeutung
1	Referenzbit auf 0 setzen	Seite erhält zweite Chance
0	—	Seite ist Ersetzungskandidat

- im schlimmsten Fall erfolgt ein „Rundumschlag“ über alle Seiten ☞ FIFO
  - falls die Referenzbits aller betrachteten Seiten (auf 1) gesetzt waren
- allgemein gilt: zu ersetzende **beschriebene Seiten** („*dirty*“) auslagern

LRU — *Second Chance*



LRU — Approximation (4)

enhanced second chance prüft, ob eine Seite schreibend referenziert wurde

- Grundlage dafür ist ein **Modifikationsbit** (*modify/dirty bit*) pro Seite
  - wird bei Schreibzugriffen auf 1 gesetzt, behält sonst seinen Zustand bei
- zusammen mit dem Referenzbit ergeben sich vier Paarungen (R, D):

	Bedeutung	Entscheidung
(0, 0)	ungenutzt	beste Wahl
(0, 1)	beschrieben	keine schlechte Wahl
(1, 0)	kürzlich gelesen	keine gute Wahl
(1, 1)	kürzlich beschrieben	schlechteste Wahl

- kann zwei Durchläufe erwirken, Seiten eine weitere Chance geben ☞ MacOS

Freiseitenpuffer

- Alternative zu Ersetzungsstrategien, basiert auf **Vorabruf** (☞ *Ladestrategie*)
  - das System sorgt anhaltend für eine bestimmte Menge freier Seitenrahmen
  - über **Schwellwerte** („*water mark*") wird die Aus-/Einlagerung gesteuert:
    low Seitenrahmen als frei markieren, Seiten ggf. auslagern
    high Seiten ggf. einlagern, *Vorausladen*
- frei markierte Seiten wandern in einen **Zwischenpuffer** (*cache*) von Freiseiten
  - die Zuordnung von Seiten zu Seitenrahmen bleibt jedoch erhalten
  - vor ihrer Ersetzung doch noch benutzte Seiten werden „zurückgeholt"
  - sogenanntes „*reclaiming*" von Seiten durch Prozesse ☞ Solaris
- effizient: Ersetzungszeit entspricht weitestgehend der Ladezeit

Seitenanforderung

- wiederverwendbare Betriebsmittel „Seitenrahmen" (begrenzter Anzahl) sind mehreren Prozessen zuzuordnen
  - Rechnerausstattung und -architektur geben „harte" Begrenzungen vor:
    Obergrenze festgelegt durch die Größe des Vordergrundspeichers
    Untergrenze definiert durch den „komplexesten" Maschinenbefehl
  - „weiche" Begrenzungen: Systemlast, Grad an Mehrprogrammbetrieb
- innerhalb der durch die Hardware gegebenen Grenzen, ist die Zuordnung . . .

  gleichverteilt in Abhängigkeit von der Prozessanzahl und/oder
  größenabhängig bedingt durch den (statischen) Programmumfang

Einzugsbereich bei der Seitenersetzung

lokal nur Seitenrahmen des von der Seitenersetzung betroffenen Prozesses nutzen

- die *Seitenfehlerrate* ist von einem Prozess selbst kontrollierbar
  - Prozesse verdrängen niemals Seiten anderer Adressräume
  - fördert ein deterministisches Laufzeitverhalten von Prozessen
- *statische Zuordnung* von Seitenrahmen für den Prozessadressraum

global alle verfügbaren Seitenrahmen heranziehen; *dynamische Zuordnung*

- Verdrängung von Seitenrahmen/Ersetzung von Seiten ist unvorhersehbar
- adressraumübergreifende Beeinflussung des Laufzeitverhaltens von Prozessen

☞ Kombination beider Ansätze ist möglich (Prozess-/Adressraumklassen)

Seitenflattern

thrashing Flattern; Überlastung; die Dresche, Tracht Prügel; Niederlage

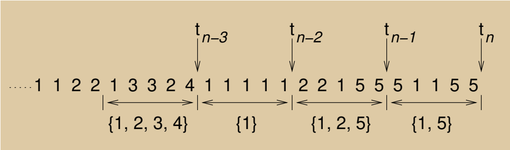
- Ein-/Auslagerung von Seiten bestimmt (phasenweise) die Systemaktivität
  - eben erst ausgelagerte Seiten werden sofort wieder eingelagert
  - Prozesse verbringen mehr Zeit beim „*paging*" als beim Rechnen
- ein die Systemleistung verringerndes Phänomen globaler Seitenersetzung
  - Prozesse bewegen sich zu nahe am Seiten(rahmen)minimum
  - zu hoher Grad an Mehrprogrammbetrieb
  - ungünstige Ersetzungsstrategie
- Lösungsansätze: „*swapping*" (✗ 10-4), (lokale/globale) Arbeitsmengen

☞ steht in Relation zu der Zeit, die Prozesse mit sinnvoller Arbeit verbringen

Arbeitsmengen

working set die Menge der Seiten, die ein Prozess (☞ lokal) bzw. das System (☞ global) in naher Zukunft aktiv in Benutzung haben wird

- die Berechnung der Arbeitsmenge ist nur annäherungsweise möglich:
  - Ausgangspunkt ist die **Seitenreferenzfolge** der jüngeren Vergangenheit



Die **Fensterbreite** deckt eine „feste Anzahl von Maschinenbefehlen“ ab, approximiert in Form von periodischen Unterbrechungen: sie ist bestimmt durch die **Periodenlänge**.

  - regelmäßig wird ein **Fenster** (*working set window*) darauf geöffnet
- kleine Fenster liefern wenig aktive Seiten, große zeigen Überlappungen

Approximation einer Arbeitsmenge

- Referenzbit, Seitenalter und periodische Unterbrechungen als Grundlage: bei Ablauf eines Zeitquantums die Referenzbits eingelagerter Seiten prüfen

	Aktion
1	Referenzbit und Alter auf 0 setzen
0	Alter der Seite um 1 erhöhen

Seiten von Arbeitsmengen haben Alterswerte kleiner als die Fensterbreite

- beim lokalen Ansatz altern nur Seiten des jeweils unterbrochenen Prozesses

  ☞ Problem: gemeinsam genutzte Seiten (z. B. *shared libraries*)

- Alternative ist der globale Ansatz, der jedoch mehr Seiten zu untersuchen hat

Zusammenfassung

- der logische Adressraum abstrahiert von den Widrigkeiten der Hardware
  - unbestückbare/reservierte Adressbereiche, Lücken in der Adressbelegung
  - die Illusion eines linear adressierbaren Speicherbereichs wird geschaffen
- der virtuelle Adressraum ermöglicht „ortstransparente Speicherzugriffe“
  - nicht benötigte Programmteile liegen im Hintergrundspeicher
  - bei Bedarf/im Voraus werden sie in den Vordergrundspeicher transferiert
- die Verwaltung des (virtuellen) Speichers verfolgt dazu mehrere Politiken
  - Platzierungs-, Lade- und Ersetzungsstrategie für Seiten bzw. Segmente
  - der Fragmentierung wird durch Verschmelzung/Kompaktifizierung begegnet