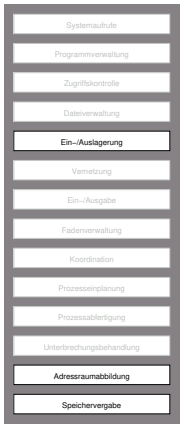
## Speicherverwaltung




- Speicher als { physikalisches / logisches / virtuelles } **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**
  initialisierte Daten globale Variablen, Halde **Daten**
  lokale Variablen, Hilfsvariablen, aktuelle Parameter **Stapel**
  } **-segment**

- 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 $(p, o)$

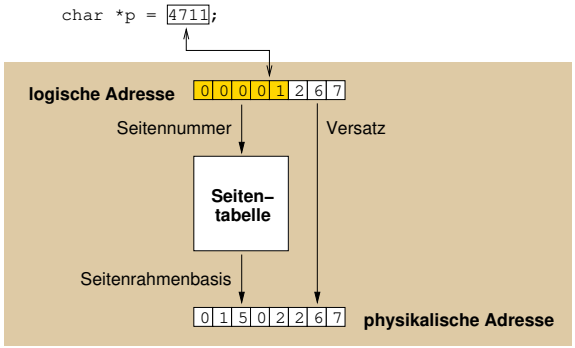
  Seitennummer (*page*) $p = A_P \texttt{ div } 2^N$
  Versatz (*offset*) $o = A_P \texttt{ mod } 2^N$

  $2^N$ ist Seitengröße in Bytes
- 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



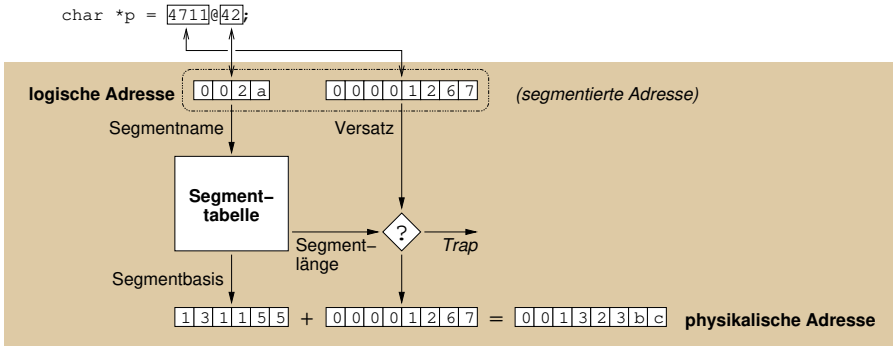

$4711_{10}$ = 0x1267 (in C)

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**

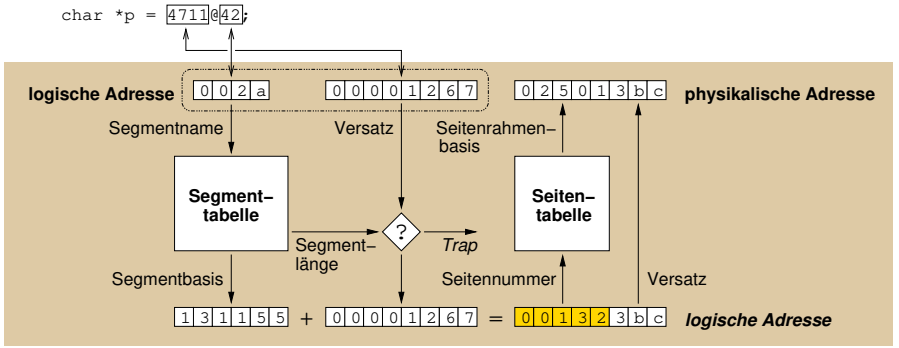


## Adressumsetzung — Segmentierter Adressraum






## 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 ☞ `sbreak(2)`



## Virtueller Adressraum

- Abstraktion von der Größe und 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

  ☞ { segmentierte / segmentiert-gekachelte / gekachelte } 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

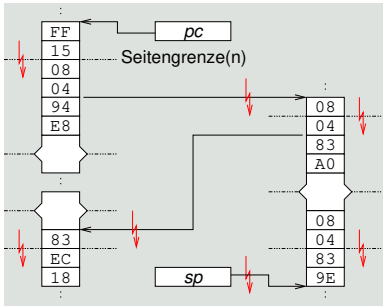
```
void hello () {
    printf("Hi!\n");
}

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 *Seitenfehlerwahrscheinlichkeit* ($p$) 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 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 und 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*) wohin die Information ablegen?

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

**Ladestrategie** (*fetch policy*) wann ist Information zu laden?

- auf Anforderung oder im Voraus?

**Ersetzungsstrategie** (*replacement policy*) welche 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 fester Größe
  - – eignet sich für die Verwaltung gekachelter Adressräume
  - – grobkörnige Vergabe freien Speichers auf Seitenrahmenbasis

  **verkettete Liste** (*free list*) freier Bereiche variabler 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:

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

  - 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



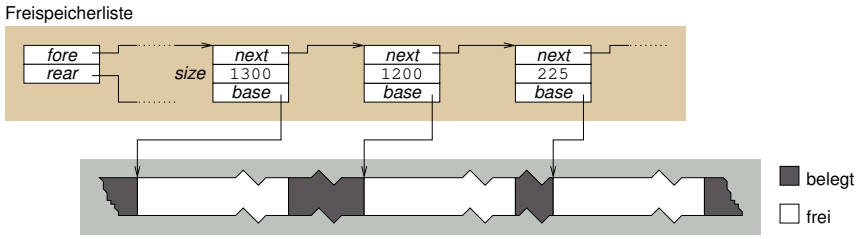
## Freispeicherliste

***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

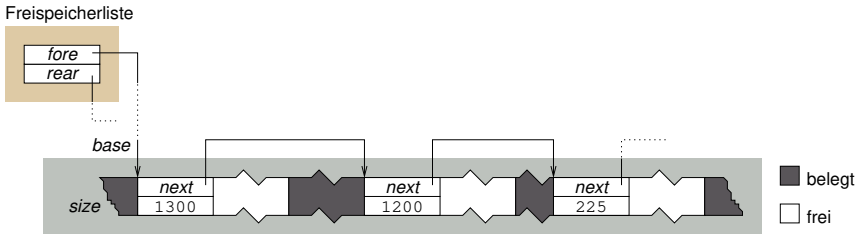


## Listenelemente als Löcherdeskriptoren




- 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)

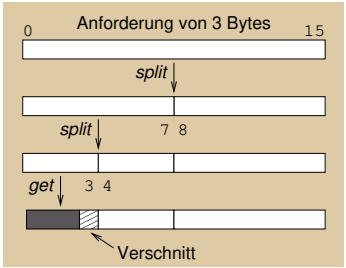
***buddy*** („Kamerad“, „Kumpel“) verwaltet Löcher nach aufsteigenden Größen von 2er-Potenzen

- 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.