

Systemprogrammierung

Grundlagen von Betriebssystemen

Teil C – XII.2 Speicherverwaltung: Zuteilungsverfahren

Wolfgang Schröder-Preikschat

21. Dezember 2021



Gliederung

Einführung
Rekapitulation

Platzierungsstrategie
Freispeicherorganisation
Verfahrensweisen

Speicherverschnitt
Fragmentierung
Verschmelzung
Kompaktifizierung

Zusammenfassung



Agenda

Einführung
Rekapitulation

Platzierungsstrategie
Freispeicherorganisation
Verfahrensweisen

Speicherverschnitt
Fragmentierung
Verschmelzung
Kompaktifizierung

Zusammenfassung



Lehrstoff

- Grundlagen der **Speicherzuteilungsstrategie** eines Betriebssystems für Mehrprogrammbetrieb thematisieren und punktuell vertiefen
 - verschiedene Formen der Organisation freien Speichers darstellen
 - Abspeicherung von Verwaltungsstrukturen beleuchten
 - Freispeicher, sog. **Löcher**, als speziell gefüllte **Hohlräume** auffassen
- klassische **Verfahrensweisen** besprechen und dadurch verschiedene Aspekte einer Zuteilungsstrategie herausarbeiten
 - Löcher nach Größe verwalten: *best-fit*, *worst-fit*, *buddy*
 - Löcher nach Adresse verwalten: *first-fit*, *next-fit*
- auf **Speicherverschnitt** eingehen, ein grundsätzliches Problem jeder Zuteilungsvariante, das ihre Effizienz bestimmt
 - **intern** ■ der, wenn er auftritt, unvermeidbar ist
 - **extern** ■ der aufwendig auflösbar ist
- **Verschmelzung** und **Kompaktifizierung** erklären, zwei Maßnahmen, um Speicherverschnitt zu minimieren oder aufzulösen



- zentrale Aufgabe ist es, über die **Speicherzuteilung** an einen Prozess Buch zu führen und seine Adressraumgröße dazu passend auszulegen **Platzierungsstrategie** (*placement policy*)
 - wo im Hauptspeicher ist noch Platz?
- zusätzliche Aufgabe kann die **Speichervirtualisierung** sein, um trotz knappem Hauptspeicher Mehrprogrammbetrieb zu maximieren
 - Ladestrategie** (*fetch policy*)
 - wann muss ein Datum im Hauptspeicher liegen?
 - Ersetzungsstrategie** (*replacement policy*)
 - welches Datum im Hauptspeicher ist ersetzbar?
- die zur Durchführung dieser Aufgaben typischerweise zu verfolgenden Strategien profitieren voneinander — oder bedingen einander
 - ein Datum kann ggf. erst platziert werden, wenn Platz freigemacht wurde
 - etwa indem das Datum den Inhalt eines belegten Speicherplatzes ersetzt
 - ggf. aber ist das so ersetzte Datum später erneut zu laden
 - bevor ein Datum geladen werden kann, ist Platz dafür bereitzustellen



Gliederung

Einführung
Rekapitulation

Platzierungsstrategie
Freispeicherorganisation
Verfahrensweisen

Speicherverschnitt
Fragmentierung
Verschmelzung
Kompaktifizierung

Zusammenfassung



Verwaltung der freien Speicherbereiche

Ein freier Bereich erscheint als **Hohlraum** im Innern des Haupts- oder Arbeitsspeichers eines Rechensystems.

- ein solcher Hohlraum wird als **Loch** (*hole*) bezeichnet, wobei mehrere davon und getrennt voneinander im realen Adressraum liegen
 - die Struktur dieser Hohlräume ist von fester oder variabler Größe
 - entsprechend motiviert sie verschiedene Darstellungen des Freispeichers
- Bitkarte**
 - für Hohlräume fester Größe \sim bit map
 - eignet sich für **seitennummelierte Adressräume**
 - grobkörnige Speichervergabe auf Seitenrahmenbasis
 - \hookrightarrow alle Hohlräume sind gleich gut bei der Löchersuche
- Lochliste**
 - für Hohlräume variabler Größe \sim hole list
 - ist typisch für **segmentierte Adressräume**
 - feinkörnige Speichervergabe auf Segmentbasis
 - \hookrightarrow nicht alle Hohlräume sind gleich gut bei der Löchersuche
- Anforderung an Verfahren zur Hohlräumzuteilung ist **Effizienz**, d.h., Sparsamkeit bezüglich Rechenzeit und Speicherplatz
 - in Hinsicht auf **Vergeudung** und **Zerstückelung** freien Speichers

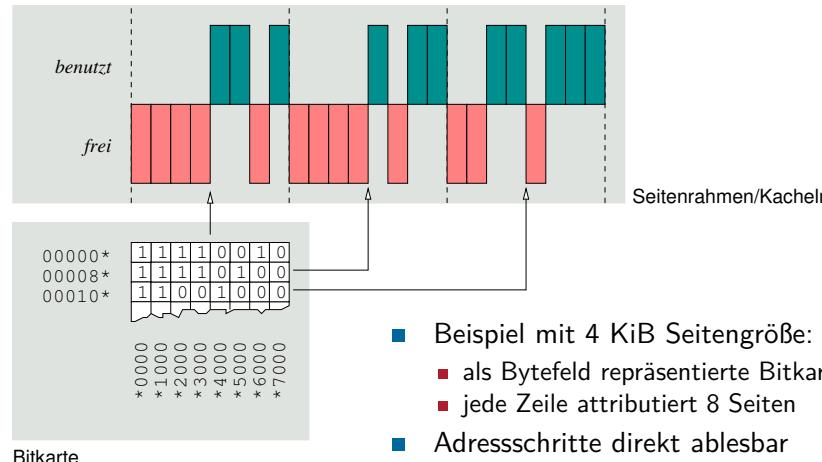


Bitkarte

- der Speicher ist aufgeteilt in **gleichgroße Stücke**, die jeweils Platz für n Bytes bieten, mit n typischerweise eine **Zweierpotenz**
 - d.h., n ist Vielfaches der **Seitengröße** eines logischen Adressraums
 - jedes solcher Stücke hat einen zweiwertigen logischen Zustand, der eine Aussage zur freien **Verfügbarkeit** macht
 - frei**
 - das Stück ist ein Hohlraum, keinem Prozess zugeordnet
 - benutzt**
 - das Stück ist kein Hohlraum, einem Prozess zugeordnet
 - je nach Konvention mit den Werten **1** und **0** kodiert, oder umgekehrt
 - der Speicherbedarf der Karte für den gesamten Hauptspeicher eines Rechners hängt damit maßgeblich von der Stückgröße ab
 - angenommen 8 GiB Hauptspeicher und 4 KiB Stück (Seitengröße):
- 8 GiB = 2097152 Seiten \times 4096 Bytes = 2097152 Bits
= 262144 Bytes = 256 KiB
- d.h., die Unkosten zur Abspeicherung der Bitkarte betragen 0.003 %
 - Aktionen zur Suche, zum Erwerben und zur Abgabe eines Hohlräums operieren auf ein byteweise gespeichertes zweidimensionales Bitfeld
 - manche Prozessoren (x86) bieten hierfür spezielle Maschinenbefehle



Freispeicherverwaltung mit Bitkarte



- Beispiel mit 4 KiB Seitengröße:
 - als Bytefeld repräsentierte Bitkarte
 - jede Zeile attributiert 8 Seiten
- Adressschritte direkt ablesbar
 - pro Zeile 0x8000 ($8 \times 4KiB$)
 - pro Spalte 0x1000 (4KiB)
- in Kombination die Kacheladresse

Lochliste

- der Speicher ist aufgeteilt in eventuell verschiedengroße Stücke, die jeweils Platz für mindestens n Bytes bieten
 - wobei n typischerweise Vielfaches der Größe von einem **Listenelement** ist
 - d.h., 4, 8 oder 16 Bytes bei einer 16-, 32- bzw. 64-Bit Maschine
- 1
2
3
4

```
typedef struct piece {
    chain_t *next; /* single-linked list assumed */
    size_t size;   /* # of bytes claimed by this piece */
} piece_t;
```

- der Speicherbedarf einer Liste für den gesamten Hauptspeicher eines Rechners hängt damit von Anzahl und Größe der Hohlräume ab
 - gleiche Annahme wie zuvor, jedoch Seite gleich Segment und 64-Bit:

$$8 \text{ GiB} < 2097152 \text{ Listenelemente (piece_t) à 16 Bytes}$$

$$< 33554432 \text{ Bytes} \equiv 32 \text{ MiB}$$
 - d.h., die Unkosten zur Abspeicherung der Lochliste liegen unter 0.390 %
 - sie fallen an, falls Hohlräume selbst unbrauchbar zur Abspeicherung sind
- Aktionen zur Suche, zum Erwerben und zur Abgabe eines Hohlräums beziehen sich auf eine **dynamische Datenstruktur**

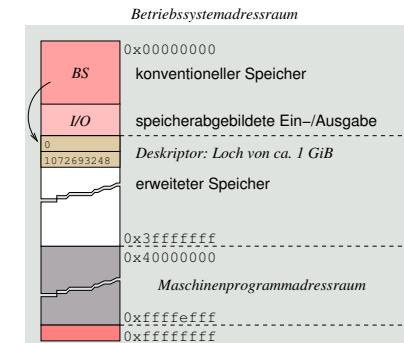
Abspeicherung der Lochliste

- jedes Listenelement beschreibt ein Stück freien Speicher, d.h., einen leeren oder mit etwas angefüllten Hohlräum im Speicherinnern
 - z.B. angefüllt mit eben dem Listenelement, das den Hohlräum beschreibt
- somit ergeben sich zwei grundlegende **Speicherausprägungen** für die Lochliste, mit Konsequenzen in verschiedener Hinsicht
 - die Hohlräume sind wirklich leer, adressräumlich von der Liste getrennt
 - jeder Hohlräum ist freier Speicherplatz im realen Adressraum, das durch ihn repräsentierte Loch kann beliebig klein sein: $\text{sizeof}(hole) > 0$
 - jedes Listenelement belegt Speicher im Adressraum des Betriebssystems und die Listenoperationen wirken in derselben Schutzdomäne
 - die Hohlräume sind scheinbar leer, adressräumlich mit der Liste vereint
 - jeder Hohlräum ist freier Speicher und zugleich ein Listenelement im realen Adressraum, er hat eine Mindestgröße: $\text{sizeof}(hole) \geq \text{sizeof}(piece_t)$
 - kein Listenelement belegt Speicher im Adressraum des Betriebssystems, aber die Listenoperationen wirken in einer anderen Schutzdomäne
- bei spezieller Auslegung des Betriebssystemadressraums kann von den positiven Eigenschaften beider Ausprägungen profitiert werden

Adressraumbelegungsplan Betriebssystem

32-Bit

- angenommen, der Hauptspeicher von $\approx 1 \text{ GiB}$ liegt partitioniert im realen Adressraum wie folgt:
 - 640 KiB **konventioneller Speicher** ab Adresse 0x00000000
 - 1 GiB – 640 KiB **erweiterter Speicher** ab Adresse 0x00100000
- weiter sei angenommen, dass für das Betriebssystem eine **identische Abbildung** (*identity mapping*) von logischen zu realen Adressen gilt
 - die **Adressraumpartition** für das Betriebssystem macht die unteren 1 GiB aus (vgl. [2, S. 29–31]):
 - der konventionelle Speicher ist für das Betriebssystem bestimmt
 - der erweiterte Speicher ist für die Maschinenprogramme bestimmt
 - die **Lochliste** liegt dann ebenfalls im erweiterten Speicher
 - initial besteht die Lochliste aus nur einem Listenelement



Identische Abbildung

identity mapping

Die **Identität** von realem und logischem/virtuellem Adressraum des Betriebssystems, in dem eine logische/virtuelle Adresse identisch zu einer realen Adresse ist.

- im gegebenen Beispiel bedeutet dies, dass die logische Adresse eines Elements der Lochliste der realen Adresse des Lochs gleicht
 - d.h., die Elemente der Lochliste liegen im Betriebssystemadressraum und
 - jedes Element füllt dabei jeweils auch einen Hohlräume im Hauptspeicher
- die Listenoperationen wirken in der Domäne des Betriebssystems
- zur Verwaltung freien Speichers fällt kein zusätzlicher Speicherbedarf an
- bei dieser **Hilfskonstruktion** (*workaround*) sind nicht nur Hohlräume, sondern alle Stücke dem Betriebssystem direkt zugänglich
 - Adressierungsfehler im Betriebssystem können daher leicht Stücke treffen, die Maschinenprogramme oder einige ihrer Bestandteile speichern
 - diese Stücke sind daher im Betriebssystemadressraum auszublenden
- sowohl segmentierter als auch seitennummerierter Adressraum helfen, die **Gebrauchsstücke** vor direkten Zugriffen zu schützen

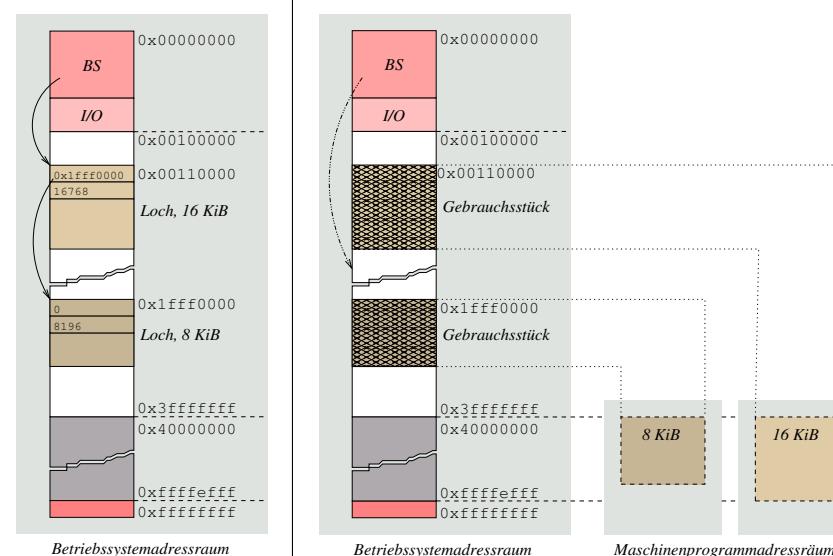
©wosch

SP (WS 2021/22, C – XII.2)

2.1 Platzierungsstrategie – Freispeicherorganisation

XII.2/13

Ein- und Ausblendung von Speicherstücken



©wosch

SP (WS 2021/22, C – XII.2)

2.1 Platzierungsstrategie – Freispeicherorganisation

XII.2/15

Gebrauchsstück

functional piece

Neben den Stücken, die Hohlräume darstellen, solche, die allgemein zur Ablage von Programmtext, -daten und Stapeln im Hauptspeicher von Prozessexemplaren in Gebrauch sind.

- ein solches Stück bildet entweder ein **Segment** oder ein **Vielfaches von Seiten**, je nach Adressraumkonzept (vgl. [2])
 - geschützt durch einen Segmentdeskriptor bzw. $n \geq 1$ Seitendeskriptoren
 - zugeteilt dem Adressraum des Prozesses, der das Stück gebraucht
- im Moment der **Zuteilung** zum Prozessadressraum, wird es aus dem Betriebssystemadressraum ausgeblendet
 - der mit dem Stück darin abgedeckte Adressbereich bleibt jedoch gültig
 - allerdings ist dieser Bereich nicht mehr durch **Adresszugriffe** zugänglich
- bei **Zurücknahme** der Stücke bzw. **Zerstörung** des Prozessexemplars werden sie wieder in den Betriebssystemadressraum eingeblendet
 - die Stücke werden wieder zu Hohlräumen, kommen auf die Lochliste
 - sie erscheinen wieder an ihren alten Stellen im Betriebssystemadressraum

Die Lösung ist immer einfach, man muss sie nur finden. (Alexander Solschenizyn)

©wosch

SP (WS 2021/22, C – XII.2)

2.1 Platzierungsstrategie – Freispeicherorganisation

XII.2/14

Lineare Lochliste I

Hinweis (Verschnitt vs. Suchaufwand)

Ist die angeforderte Größe kleiner als die Größe des gefundenen Loch, die Differenz jedoch größer als ein Listenelement ist, fällt **Verschnitt** an, der als verbleibendes Loch in die Liste neu eingesortiert werden muss.

- die Lochliste ist der Größe nach auf- oder absteigend sortiert:
 - best-fit**
 - aufsteigende Lochgrößen, das kleinste passende Loch suchen
 - beste Zuteilung, minimaler Verschnitt, aber eher langsam
 - erzeugt kleine Löcher von vorn, erhält große Löcher hinten
 - hinterlässt eher kleine Löcher, bei steigendem Suchaufwand
 - worst-fit**
 - absteigende Lochgrößen, das größte passende Loch suchen
 - sehr schnelle Zuteilungsentscheidung, begünstigt Zerstückelung
 - zerstört große Löcher von vorn, macht kleine Löcher hinten
 - hinterlässt eher große Löcher, bei konstantem Suchaufwand
- fällt ein Restloch an, muss dieses in die Liste eingesortiert werden, aber nur, wenn es eine bestimmte **Mindestgröße** nicht unterschreitet
 - typischerweise die Größe (in Bytes) eines Listenelements

©wosch

SP (WS 2021/22, C – XII.2)

2.2 Platzierungsstrategie – Verfahrensweisen

XII.2/16

Hinweis (Suchaufwand vs. Zuteilung)

Ist die angeforderte Größe kleiner als die Größe des gefundenen Loch, die Differenz jedoch größer als ein Listenelement ist, fällt Verschnitt an, der jedoch nicht in die Liste eingesortiert werden muss.

- die Lochliste ist der Größe des Adresswerts nach aufsteigend sortiert:
 - **first-fit** ■ schnelle Zuteilung, begünstigt aber Verschwendungen
 - erzeugt kleine Löcher von vorn, erhält große Löcher hinten
 - hinterlässt eher kleine Löcher, bei steigendem Suchaufwand
 - **next-fit** ■ reihum (*round-robin*) Variante von *first-fit*
 - die Suche beginnt immer beim zuletzt zugeteiltem Loch
 - hinterlässt eher gleichgroße Löcher (Gleichverteilung)
 - Konsequenz ist ein im Mittel eher abnehmender Suchaufwand
- keine dieser Verfahren erzeugt ein Restloch, das im nachgeschalteten zweiten Listendurchlauf eingesortiert werden müsste
 - sie machen eine effiziente Hohlraumverwaltung (vgl. S. 14) möglich



Gliederung

Einführung

Rekapitulation

Platzierungsstrategie

Freispeicherorganisation
Verfahrensweisen

Speicherverschnitt

Fragmentierung
Verschmelzung
Kompaktifizierung

Zusammenfassung



Halbierungsverfahren

Hinweis (Verschnitt vs. Suchaufwand)

Das zur Speicheranfrage gegebener Größe am besten passende Stück durch fortgesetzte Halbierung eines großen Stücks gewinnen.

- die Lochliste ist der **Zweierpotenzgröße** nach aufsteigend sortiert:
 - **buddy** ■ sucht das kleinste passende Loch *buddy*; der Größe 2^i
 - i ist Index in eine Tabelle von Adressen auf Löcher der Größe 2^i
 - wobei i so zu bestimmen ist, dass gilt $2^{i-1} < \text{size} \leq 2^i$, $i > 1$
 - mit *size* als Größe (in Bytes) des angeforderten Speicherstücks
 - *buddy*; entsteht durch sukzessive Splittung von *buddy*, $j > i$:
 - $2^n = 2 \times 2^{n-1}$
 - zwei gleichgroße Stücke, die „Kumpel“ des jeweils anderen sind
 - i wird fortgesetzt dekrementiert, solange $2^{i-1} > \text{size}$, $i > 1$
- mögl. Verschnitt durch eine **Auswahl von Stückgrößen** begegnen
 - vergleichsweise geringer Such- und Aufsplittungsaufwand, jedoch kann der anfallende Verschnitt dennoch beträchtlich sein
 - im Mittel sind die zugeteilten Stücke um 1/3 größer als angefordert und die belegten Stücke nur zu 3/4 genutzt [1, S. 32]



Bruchstückbildung

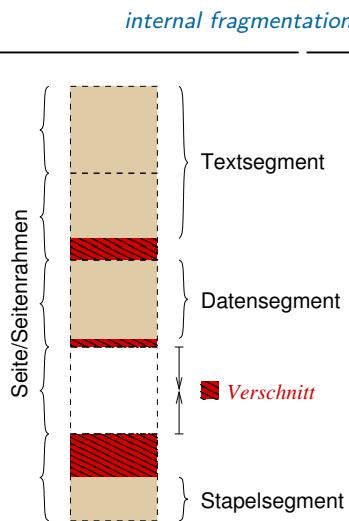
Verschnitt durch zuviel zugeteilte oder nicht nutzbare Bereiche, der als Abfall in Erscheinung tritt und Verschwendungen bedeutet.

- je nach **Adressraumkonzept** und **Zuteilungsverfahren** zeigen sich verschiedene Ausprägungen der **Fragmentierung**
 - **intern** ■ seitennummerierte Adressräume, Halbierungsverfahren (*buddy*)
 - die angeforderte Größe ist kleiner als das zugeteilte Stück
 - falls Seitennummerierung, ist die Größe auch kein Seitenvielfaches
 - der „lokale Verschnitt“ ist nutzbar, dürfte es aber nicht sein
 - **Verschwendungen**, ist (durch das Betriebssystem) unvermeidbar
 - **extern** ■ segmentierte Adressräume, Halbierungsverfahren (*buddy*)
 - die angeforderte Größe ist zu groß für jedes einzelne Loch
 - in Summe ihrer Größen genügen die Löcher der angeforderten Größe
 - allerdings sind sie im Hauptspeicher nicht linear angeordnet
 - der „globale Verschnitt“ ist ggf. nicht mehr zuteilbar
 - **Verlust**, ist (durch das Betriebssystem) aufwendig vermeidbar
 - externe Fragmentierung kann durch **Verschmelzung** verringert und **Kompaktifizierung** aufgelöst werden



Interne Fragmentierung

- **seitennummerierter Adressraum**
 - abzubildende Programmsegmente sind Vielfaches von Bytes
 - der (log./virt.) Prozessadressraum ist aber ein Vielfaches von Seiten
 - die jew. letzte Seite der Segmente ist ggf. nicht komplett belegt
- **seitenlokaler Verschnitt**
 - wird vom Programm *logisch* nicht beansprucht
 - ist vom Prozess *physisch* jedoch adressierbar
 - da eine seitennummelierte MMU Seiten schützt, keine Segmente
- das **Halbierungsverfahren (buddy)** liefert ein ähnliches Bild
 - immer dann, wenn die Anforderungsgröße keine Zweierpotenz ist
 - ein Verschnitt von $2^i - \text{size}$ (in Bytes) ergibt sich zum Stückende hin



©wosch

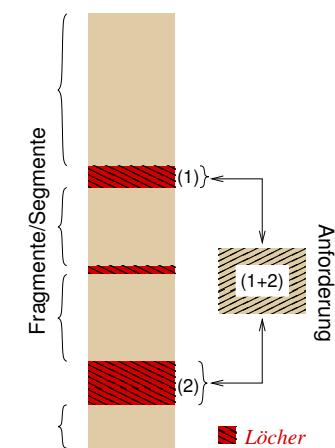
SP (WS 2021/22, C – XII.2)

3.1 Specherverschnitt – Fragmentierung

XII.2/21

Externe Fragmentierung

- **segmentierter Adressraum**
 - die zu platzierenden Fragmente sind Vielfaches von Bytes
 - sie werden 1:1 auf Segmente einer MMU abgebildet
 - die jew. eine lineare Bytefolge im realen Adressraum bedingen
- **globaler Verschnitt**
 - die Summe von Löchern ist groß genug für die Speicheranforderung
 - die Löcher liegen aber verstreut im realen Adressraum vor und
 - jedes einzelne Loch ist zu klein für die Speicheranforderung
- das **Halbierungsverfahren (buddy)** liefert ein ähnliches Bild
 - immer dann, wenn zwischen zu kleinen Löchern ein Gebrauchsstück liegt
 - jede Stückgröße ist eine Zweierpotenz, das größte Loch ist aber zu klein



©wosch

SP (WS 2021/22, C – XII.2)

3.1 Specherverschnitt – Fragmentierung

XII.2/22

Vereinigung eines Lochs mit angrenzenden Löchern

Eine wichtige Maßnahme, die bei der **Zurücknahme** eines Gebrauchsstücks oder **Zerstörung** eines Prozessexemplars greift.

- **Verschmelzung** von Löchern erzeugt größere Hohlräume und bringt damit folgende positive (nichtfunktionale) Eigenschaften
 - weniger Löcher, dadurch geringere externe Fragmentierung
 - weniger Lochdeskriptoren, dadurch kürzere Listen und Suchzeiten
 - beides beschleunigt die Speicherzuteilung, gibt kürzere Antwortzeiten
- **Löchervereinigung** sieht sich mit vier Situationen konfrontiert, je nach dem, welche relative Lage ein Loch im Adressraum hat:
 1. zw. zwei Gebrauchsstücken
 2. direkt nach einem Loch
 3. direkt vor einem Loch
 4. zwischen zwei Löchern
 - keine Vereinigung möglich
 - Vereinigung mit Vorgänger
 - Vereinigung mit Nachfolger
 - Kombination von 2. und 3.



©wosch

SP (WS 2021/22, C – XII.2)

3.2 Specherverschnitt – Verschmelzung

XII.2/23

Bezug zum Zuteilungsverfahren

... KISS

- die **Verschmelzungsaufwände** variieren teils sehr stark mit der Art und Weise, wie die Lochliste vom Betriebssystem geführt ist:
 - **buddy**
 - das Stück wird mit seinem *Buddy*-Stück verschmolzen
 - Adressen zweier *Buddies* gleichen sich bis auf einem Bit
 - ein Stück ist *Buddy* eines anderen Stücks, wenn gilt:


```
1 bool buddy(void *this, unsigned size, void *that) {
2     return (size && !(size & (size - 1))) /* power of two!? */
3         && ((unsigned)this ^ (unsigned)that) == size;
4 }
```
 - ggf. mit jeweils nächst größerem *Buddy* verschmelzen
 - **first/next-fit**
 - beim Einsortieren in die Lochliste Nachbarschaft prüfen
 - ein Stück ist Nachbar eines anderen Stücks, wenn gilt:


```
5 bool neighbor(void *this, unsigned size, void *that) {
6     return ((unsigned)this + size) == (unsigned)that;
7 }
```
 - mit jeweils aktuellem und nächsten Listenelement prüfen
 - **best/worst-fit**
 - wie *first/next-fit*, beim Einsortieren prüfen
 - aber Listennachfolger müssen keine Nachbarn sein ☺
 - die ganze Lochliste durchlaufen: zwei Nachbarn finden ☺
 - erst dann ggf. verschmelzen und neu einsortieren ☺



©wosch

SP (WS 2021/22, C – XII.2)

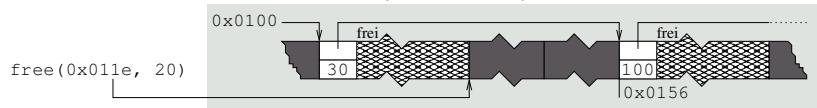
3.2 Specherverschnitt – Verschmelzung

XII.2/24

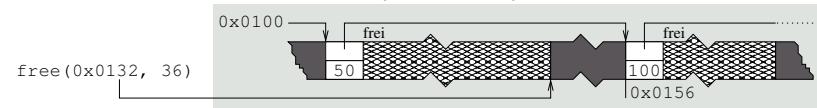
Vereinigung beliebiger Löcher

first/next-fit

- das Stück ist „rechter Nachbar“ (Nachfolger) des Lochs:

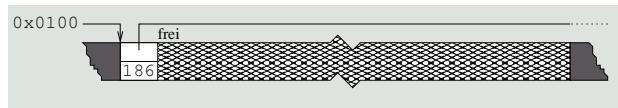


- das alte 30 Bytes große Loch kann um 20 Bytes vergrößert werden
- Verschmelzung; das nächste Stück ist „rechter Nachbar“ (Nachfolger) des Lochs und „linker Nachbar“ (Vorgänger) des Lochnachfolgers



- das alte 50 Bytes große Loch kann um 36 Bytes vergrößert werden
- das neue 86 Bytes große Loch kann um 100 Bytes vergrößert werden

- Verschmelzung:



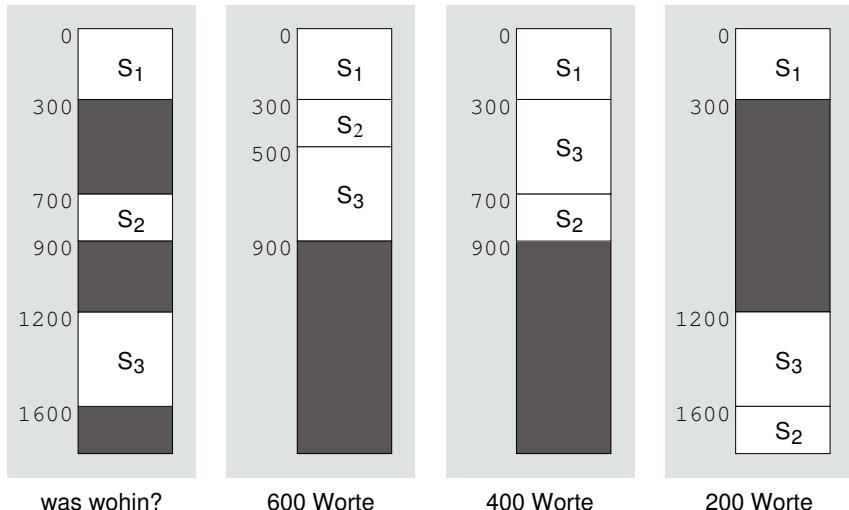
©wosch

SP (WS 2021/22, C – XII.2)

3.2 Speicherverschnitt – Verschmelzung

XII.2/25

Auflösung externer Fragmentierung: Optionen



©wosch

SP (WS 2021/22, C – XII.2)

3.3 Speicherverschnitt – Kompaktifizierung

XII.2/27

Vereinigung des globalen Verschnitts

Die Gebrauchsstücke im Hauptspeicher werden so verschoben, dass am Ende ein einziges großes Loch vorhanden ist.

- um **externe Fragmentierung** aufzulösen, sind Gebrauchsstücke im Hauptspeicher durch **Kopievorgänge** umzulagern
 - direkt im Hauptspeicher oder
 - indirekt über den Ablagespeicher \sim swapping
- so wird zunächst ein weiteres Loch geschaffen, das dann aber gleich wieder mit Nachbarlöchern verschmilzt
 - schrittweise wird die Lochliste verkürzt, bis nur noch ein Loch übrigbleibt
- Umlagerung zieht **Verlagerung** der betroffenen Segmente oder Seiten nach sich, wenn sie ihre neue Position im realen Adressraum haben
 - deren Lage ändert sich nur im realen Adressraum, nicht im logischen
 - nur die Basisadresse im Segment-/Seitendeskriptor ist zu aktualisieren
 - im logischen Adressraum behält jedes Segment/jede Seite seine Adresse
- zentraler Aspekt dabei ist, die Anzahl der Umlagerungsvorgänge zu minimieren, was ein komplexes **Optimierungsproblem** darstellt

©wosch

SP (WS 2021/22, C – XII.2)

3.3 Speicherverschnitt – Kompaktifizierung

XII.2/26

Gliederung

Einführung
Rekapitulation

Platzierungsstrategie
Freispeicherorganisation
Verfahrensweisen

Speicherverschnitt
Fragmentierung
Verschmelzung
Kompaktifizierung

Zusammenfassung

©wosch

SP (WS 2021/22, C – XII.2)

4. Zusammenfassung

XII.2/28

- Zuteilung von Arbeitsspeicher ist Aufgabe der **Platzierungsstrategie**
 - die Erfassung freier Speicherstücke hängt u.a. ab vom Adressraummodell
 - i Seiten bzw. Seitenrahmen \sim Bitkarte oder Lochliste
 - ii Segmente \sim Lochliste
 - weitere Folge davon ist **interne (i) oder externe (ii) Fragmentierung**
 - Speicherverschnitt durch zuviel zugeteilte bzw. nicht nutzbare Bereiche
- die **Zuteilungsverfahren** verwalten Löcher nach Größe oder Adresse
 - nach abnehmender Größe *worst-fit*
 - nach ansteigender Größe *best-fit, buddy*
 - Adresse *first-fit, next-fit*
- angefallener **Speicherverschnitt** ist zu reduzieren oder aufzulösen
 - i Verschmelzung von Löchern verringert externe Fragmentierung
 - beschleunigt die Speicherzuteilungsverfahren und
 - lässt die Speicherzuteilung im Mittel häufiger gelingen
 - ii Kompaktifizierung der Löcher löst externe Fragmentierung auf
 - hinterlässt (im Idealfall) ein großes Loch
 - erfordert aber positionsunabhängige Programme d.h. logische Adressräume



Halbierungsverfahren: Beispiel

binary buddy

- angenommen sei die Anforderung eines Speicherstücks von 42 Bytes:
 - $2^6 = 64$ ist kleinste Zweierpotenz ≥ 42 , d.h., zuzuteilen sind 64 Bytes
 - ein entsprechend großes Loch fehlt, es ist durch Splitting zu erzeugen
 - nächstes Loch in der Liste ist ein Stück von 1024 KiB
 1. $1024 = 2^{10} = 2 \times 2^9 = 512 + 512$, zu groß, eins davon wird halbiert
 2. $512 = 2^9 = 2 \times 2^8 = 256 + 256$, zu groß, eins davon wird halbiert
 3. $256 = 2^8 = 2 \times 2^7 = 128 + 128$, zu groß, eins davon wird halbiert
 4. $128 = 2^7 = 2 \times 2^6 = 64 + 64$, passt, eins davon wird zugeteilt

1024 KiB			
512		512 KiB	
256	256	512 KiB	
128	128	512 KiB	
64	64	128	256

- geeignet ist eine **zweidimensionale Repräsentation** der Lochliste:
 - i eine **Tabelle** von Buddy-Klassen, aufsteigend sortiert (Zweierpotenzen), lokal gespeichert im Betriebssystem
 - ii eine **lineare Liste** gleicher Buddies, gespeichert in den Hohlräumen



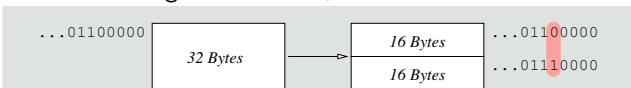
- [1] HEISS, H.-U. : Speicherverwaltung.
In: AG BETRIEBSYSTEME UND VERTEILTE SYSTEME (Hrsg.): *Konzepte und Methoden der Systemsoftware*. Universität-GH Paderborn, 2000 (Vorlesungsfolien), Kapitel 5
- [2] KLEINÖDER, J. ; SCHRÖDER-PREIKSCHAT, W. : Adressräume.
In: [4], Kapitel 12.1
- [3] KLEINÖDER, J. ; SCHRÖDER-PREIKSCHAT, W. : Speicher.
In: [4], Kapitel 6.2
- [4] KLEINÖDER, J. ; SCHRÖDER-PREIKSCHAT, W. ; LEHRSTUHL INFORMATIK 4 (Hrsg.): *Systemprogrammierung*. FAU Erlangen-Nürnberg, 2015 (Vorlesungsfolien)



Halbierungsverfahren: Randbedingungen

binary buddy

- bei dieser Technik bilden die Größen der Adressbereiche aller Löcher und Gebrauchsstücke eine Zweierpotenz
 - wird ein beliebiges dieser Stücke in zwei gleich große Hälften gesplittet, entstehen zwei *Buddies*, deren Größe wieder einer Zweierpotenz bildet
 - umgekehrt: werden diese beiden *Buddies* wieder kombiniert, entsteht ein einzelner *Buddy* doppelter Größe
- *Buddy*-Stücke der Größe 2^n werden im Adressraum so platziert, dass ihre jeweilige Anfangsadresse ein Vielfaches von 2^n ist
 - d.h., für $n > 1$ sind die niederwertigen n Bits dieser Adressen gleich 0
- aus dieser Nebenbedingung ergibt sich folgende Konsequenz, die in zweierlei Hinsicht von Bedeutung ist:
 - i bei Splitting eines *Buddies* der Größe 2^{n+1} unterscheiden sich die beiden Adressen der *Buddy*-Hälften nur in Bit 2^n
 - ii gegeben sei ein Stück der Größe 2^n an Adresse a , dann errechnet sich die Adresse b seines *Buddies* wie folgt: $b = a + 2^n$, mit $a \bmod 2^n = 0$
- Beispiel: *Buddy* der Größe 2^5
 - gibt 2×2^4



- die **Lochliste**, repräsentiert als Tabelle verketteter *Buddies*:

```
1 static chain_t *holelist[N SLOT];
```

- ein Loch als *Buddy* hervorbringen (*breed*):

```
2 void *breed(unsigned slot) {
3     chain_t *hole = 0;
4     if (slot < N SLOT) {
5         if (holelist[slot] != 0) {
6             hole = holelist[slot];
7             holelist[slot] = hole->link;
8         } else {
9             if ((hole = breed(slot + 1)) != 0) {
10                 chain_t *next = (chain_t *)((unsigned)hole ^ (1 << slot));
11                 next->link = 0;
12                 holelist[slot] = next;
13             }
14         }
15     }
16     return hole;
17 }
```

- einen Bedarf (*need*) an freien Speicher geltend machen:

```
18 void *need(unsigned size) {
19     unsigned slot;
20     for (slot = 0; (1 << slot) < size; slot++);
21     return breed(slot);
22 }
```



- vgl. auch die Belegung von S. 31 (v. li.): fünf Gebrauchsstücke A–E
 - free(D, 256), kein freier *Buddy*, verbleibt als Loch D von 256 Bytes
 - free(B, 64), kein freier *Buddy*, verbleibt als Loch B von 64 Bytes
 - free(A, 64), freier *Buddy* B, verschmilzt zu Loch AB von 128 Bytes
 - free(C, 128), freier *Buddy* AB, verschmilzt zu Loch ABC von 256 Bytes
 - freier *Buddy* D, verschmilzt zu Loch ABCD von 512 Bytes
 - free(E, 512), freier *Buddy* ABCD, verschmilzt zum Loch von 1024 KiB

	64	64	128	256	512 KiB
1	64	64	128	256	512 KiB
2	64	64	128	256	512 KiB
3	128	128	256		512 KiB
4	256		256		512 KiB
4		512			512 KiB
5				1024	KiB

