

# Systemprogrammierung

*Grundlagen von Betriebssystemen*

## Teil B – VI.2 Betriebssystemkonzepte: Speicher

Wolfgang Schröder-Preikschat

23. Juni 2022



## Agenda

---

Einführung

Grundlagen

Speicherorganisation

Adressraum

Speicherverwaltung

Einleitung

Speicherzuteilung

Speichervirtualisierung

Zusammenfassung



# Gliederung

---

## Einführung

### Grundlagen

- Speicherorganisation
- Adressraum

### Speicherverwaltung

- Einleitung
- Speicherzuteilung
- Speichervirtualisierung

### Zusammenfassung



# Lehrstoff

---

- behandelt werden grundlegende Aspekte der **Speicherorganisation**
  - Dauerhaftigkeit von Datenhaltung und die Speicherpyramide
  - von Prozessen generierte Referenzfolge innerhalb ihrer Adressräume
  - Unterschied zwischen realen, logischen und virtuellen Adressraum
- die Grundkonzepte der **Speicherverwaltung** kennenlernen
  - Speicherzuteilung und -virtualisierung differenzieren
  - Auflösung gekachelter/segmentierter Speicherverwaltung erklären
  - wesentliche Merkmale von virtuellem Speicher im Ansatz vorstellen
- Bedeutung der sogenannten **Platzierungsstrategie** vermitteln
  - kurz segmentierte und gekachelte Verfahren vorstellen
  - Suchaufwand und Verschnitt gegenüberstellen
  - wichtige Verfahren (*best-*, *worst-*, *first-*, *next-fit*) benennen
- Vertiefung „dynamischer Speicher“, die **Halde**
  - Freispeicherverwaltung auf Maschinenprogrammebene
  - Interaktion zwischen Maschinenprogramm und Betriebssystem zeigen
  - zwischen lokaler und globaler Speicherverwaltung unterscheiden



# Gliederung

Einführung

Grundlagen

Speicherorganisation  
Adressraum

Speicherverwaltung

Einleitung  
Speicherzuteilung  
Speichervirtualisierung

Zusammenfassung



## Speichersysteme

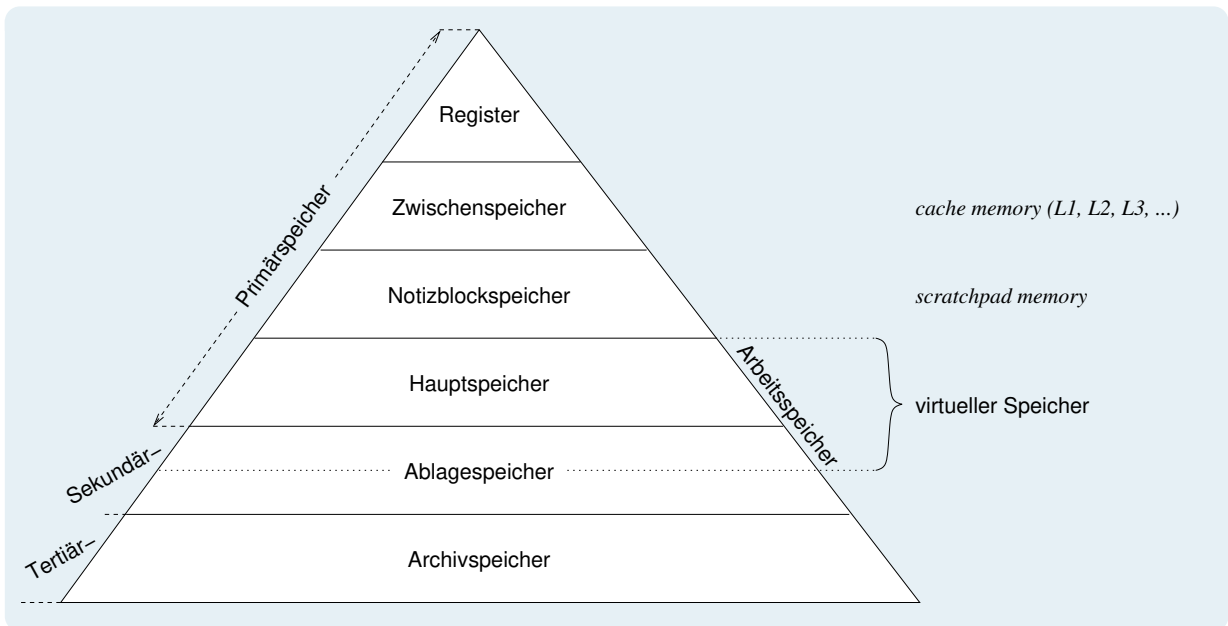
Dauerhaftigkeit von Datenhaltung

### Definition (Speicher)

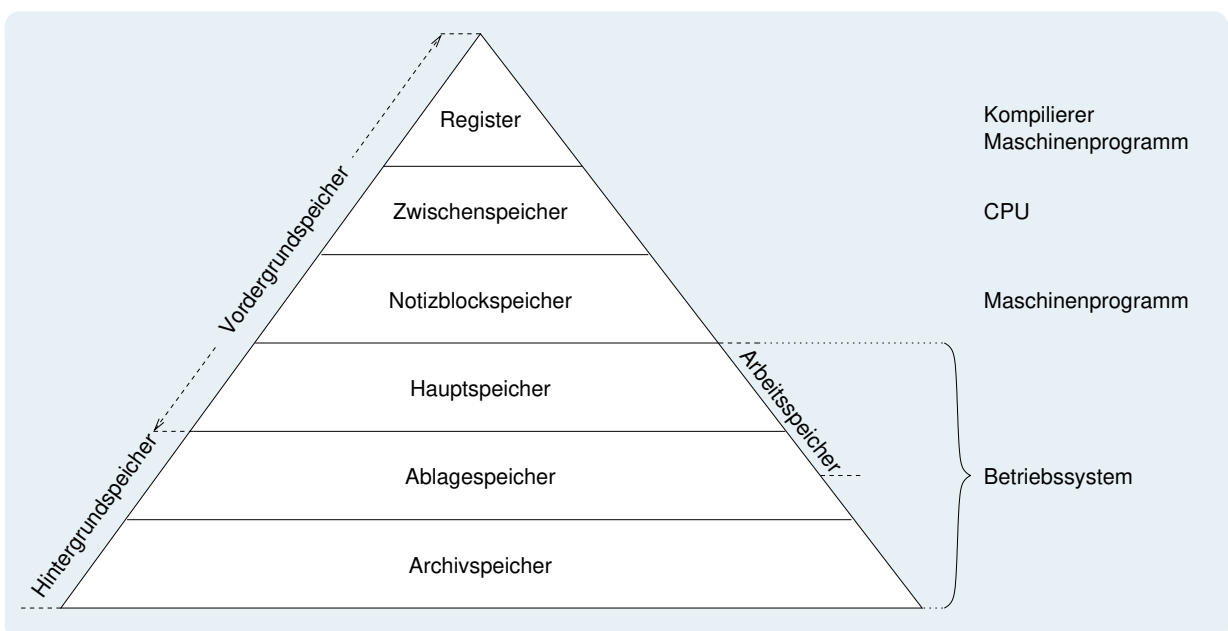
von (lat.): *spicarium* Getreidespeicher, ein Ort oder eine Einrichtung zum Einlagern von materiellen oder immateriellen Objekten.

- Vorrichtung an elektronischen Rechenanlagen, die eine kurz-, mittel- oder langfristige Speicherung von Informationen ermöglicht
  - kurz** ■ hunderte von ns  $\leq$  *Ladungshaltung*(RAM)  $\leq$  dutzende von s
    - **Primärspeicher**: Register-, Zwischen-, Haupt-/Arbeitsspeicher
  - mittel** ■ *Flash*-/Festplattenspeicher 2–10, im Mittel 5 Jahre
    - **Sekundärspeicher**: Arbeitsspeicher, Ablagesystem (Dateien)
  - lang** ■ Festplattenarchive  $\leq$  30 Jahre  $\leq$  Magnetbandarchive
    - optische Speicher (DVD) vermutlich 100 Jahre
    - **Tertiärspeicher**: Archiv
  - je größer die Zeitspanne, desto größer Kapazität und Zugriffszeit
- dabei sind Haupt-/Arbeitsspeicher und bedingt auch die Ablage in den Maschinenprogrammen direkt adressierbar
  - Multics [2] bildete Dateien auf Segmente im virtuellen Adressraum ab





- falls **virtueller Speicher** Merkmal der Maschinenprogrammebene ist, erstreckt sich der Arbeitspeicher auf Hauptspeicher und Ablage
  - nur die **Arbeitsmenge** an Text/Daten ist im Hauptspeicher eingelagert
  - alle anderen Bereiche sind in der Ablage (*swap area*) ausgelagert



- bei **Mehrprogrammbetrieb**  $\rightarrow$  **Adressraumisolation** verwaltet jedes Maschinenprogramm seinen eigenen **Haldenspeicher**
  - Verwaltungsfunktionen (`malloc/free`) stellt das Laufzeitsystem (`libc`)
  - diese interagieren mit der Speicherverwaltung des Betriebssystems



- ein **laufender Prozess** (vgl. [3, S. 20]) generiert Folgen von Adressen auf den Haupt-/Arbeitsspeicher, und zwar:
  - i nach Vorschrift des Programms, das diesen Prozess spezifiziert, wie auch
  - ii in Abhängigkeit von den Eingabedaten für den Programmablauf
- der zur Bildung dieser Adressen gegebene **Wertevorrat** hat anfangs eine feste Größe, dehnt sich gemeinhin dann aber weiter aus
  - dieser Vorrat ist initial statisch und gibt die zur Programmausführung mindestens erforderliche Menge an Haupt-/Arbeitsspeicher vor
  - zur Laufzeit ist diese **Menge dynamisch**, nimmt zu und kann dabei aber den „einem Prozess zugebilligten“ Wertevorrat nicht überschreiten
    - letzteres sichert entweder der Kompilierer oder das Betriebssystem zu
    - d.h., durch eine „typsichere Programmiersprache“ oder im Zusammenspiel mit der MMU der Befehlssatzebene
- der einem Prozess zugebilligte Adressvorrat gibt den **Adressraum** vor, in dem dieser Prozess (logisch/physisch) eingeschlossen ist
  - der Prozess kann aus seinem Adressraum normalerweise nicht ausbrechen und folglich nicht in fremde Adressräume eindringen
  - der **Prozessadressraum** hat eine durch (HW/BS) beschränkte Größe



## Gedankenspiel I

## Referenzfolge „symbolischer Adressen“

- gegeben sei folgendes Programm in C:

```

1 #include <stdlib.h>
2
3 int main() {
4     while (1)
5         ((void(*)())random())();
6 }
```

- zu bestimmen sei die Referenzfolge eines diesbezüglichen Prozesses, unter folgender Annahme:

```

7 static void vain() { }
8
9 void (*(random)())() {
10     return vain;
11 }
```

- C: ..., 5, 10, 7
- ASM<sub>x86</sub>: ..., 13, 15, 23, 24, 16, 19, 20, 17

- bzw. in ASM<sub>x86</sub>:

```

12 main:
13     subl $12, %esp
14     .L2:
15     call random
16     call *%eax
17     jmp .L2
18 vain:
19     rep
20     ret
21
22 random:
23     movl $vain, %eax
24     ret
```



- mit gdb einen Blick in den Prozessadressraum werfen, um so Einblick in den **statischen Wertevorrat** an Programmadressen zu erhalten:

```

1 (gdb) info line main
2 No line number information available for address 0x80481c0 <main>
3 (gdb) disas 0x80481c0
4 Dump of assembler code for function main:
5   0x080481c0 <+0>: push   %ebp
6   0x080481c1 <+1>: mov    %esp,%ebp
7   0x080481c3 <+3>: and   $0xffffffff,%esp
8   0x080481c6 <+6>: xchg  %ax,%ax
9   0x080481c8 <+8>: call  0x8048300 <random>
10  0x080481cd <+13>: call  *%eax
11  0x080481cf <+15>: jmp   0x80481c8 <main+8>
12 End of assembler dump.
13 (gdb) disas 0x8048300
14 Dump of assembler code for function random:
15  0x08048300 <+0>: mov    $0x80482f0,%eax
16  0x08048305 <+5>: ret
17 End of assembler dump.
18 (gdb) disas 0x80482f0
19 Dump of assembler code for function vain:
20  0x080482f0 <+0>: repz ret
21 End of assembler dump.
22 (gdb)

```

Hier ist nur die aus dem Befehlsabruf resultierende, statisch leicht bestimmbare Referenzfolge gezeigt. Es fehlen Referenzen, die die Stapeloperationen (*push*, *call* und *ret*) zur Folge haben. Diese sind mangels Kenntnis der Vorgeschichte des Prozesses allein durch „Programmlektüre“ nicht ableitbar.

- für den **Befehlsabruf** ergibt sich als **Referenzfolge** des Prozesses:

■ ..., 0x08048[1c0, 1c1, 1c3, 1c6, 1c8, 300, 305, 1cd, 2f0, 1cf]



## Adressraumlehre I

totale Abbildung  $f : A_l \rightarrow A_r$

- Befehlssatzebene (Ebene<sub>2</sub>)

### Definition (realer Adressraum)

Der durch einen Prozessor definierte Wertevorrat  $A_r = [0, 2^n - 1]$  von Adressen, mit  $e \leq n \leq 64$  und (norm.)  $e \geq 16$ . Nicht jede Adresse in  $A_r$  ist jedoch gültig, d.h.,  $A_r$  kann Lücken aufweisen.

- der **Hauptspeicher** ist adressierbar durch einen oder mehrere Bereiche in  $A_r$ , je nach Hardwarekonfiguration
- Maschinenprogrammebene (Ebene<sub>3</sub>)

### Definition (logischer Adressraum)

Der in Programm  $P$  definierte Wertevorrat  $A_l = [n, m]$  von Adressen, mit  $A_l \subset A_r$ , der einem Prozess von  $P$  zugewilligt wird. Jede Adresse in  $A_l$  ist gültig, d.h.,  $A_l$  enthält *konzeptionell* keine Lücken.

- führt **Arbeitsspeicher** ein, der linear adressierbar ausgelegt ist und durch das Betriebssystem vollständig auf den Hauptspeicher abgebildet wird



## ■ Maschinenprogrammzebene (Ebene<sub>3</sub>)

### Definition (virtueller Adressraum)

$A_v = A_l$ :  $A_v$  übernimmt alle Eigenschaften von  $A_l$ . Jedoch nicht jede Adresse in  $A_v$  bildet ab auf ein im Hauptspeicher liegendes Datum.

- Benutzung einer solchen nicht abgebildeten Adresse in  $A_v$  verursacht in dem betreffenden Prozess einen **Zugriffsfehler**
  - der Prozess erfährt eine **synchrone Programmunterbrechung** (*trap*), die vom Betriebssystem behandelt wird
  - das Betriebssystem sorgt für die **Einlagerung** des adressierten Datums in den Hauptspeicher und
  - der Prozess wird zur **Wiederholung** der gescheiterten Aktion gebracht
- der durch  $A_v$  für den jeweiligen Prozess benötigte Hauptspeicher ist „nicht in Wirklichkeit vorhanden, aber echt erscheinend“
- jedoch steht jederzeit genügend Arbeitsspeicher für  $A_v$  zur Verfügung
    - einesteils im Hauptspeicher, anderenteils in der Ablage (*swap area*)
  - der Arbeitsspeicher ist eine virtuelle, der Hauptspeicher eine reale Größe



# Gliederung

Einführung

Grundlagen

Speicherorganisation

Adressraum

Speicherverwaltung

Einleitung

Speicherzuteilung

Speichervirtualisierung

Zusammenfassung



- 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



## „Reviere“ einer Speicherverwaltung

- normalerweise sind die **Verantwortlichkeiten** auf mehrere Ebenen innerhalb eines Rechensystems verteilt  
**Speicherzuteilung**
  - Maschinenprogramm und Betriebssystem
  - Haldenspeicher, Hauptspeicher**Speichervirtualisierung**
  - ist allein Aufgabe des Betriebssystems
  - Haupt-/Arbeitsspeicher, Ablage
- das Maschinenprogramm verwaltet den seinem Prozess (-adressraum) jeweils zugeteilten Speicher **lokal** eigenständig
  - stellt dabei **sprachenorientierte Kriterien** in den Vordergrund
    - typisch für den Haldenspeicher  $\leadsto$  malloc/free
- das Betriebssystem verwaltet den gesamten Haupt-/Arbeitsspeicher **global** für alle Prozessexemplare bzw. -adressräume
  - stellt dabei **systemorientierte Kriterien** in den Vordergrund
    - hilft, einen Haldenspeicher zu verwalten  $\leadsto$  z.B. mmap/sbrk
- Maschinenprogramm und Betriebssystem gehen somit eine **Symbiose** ein, sie nehmen eine **Arbeitsteilung** vor
  - genauer gesagt: das Laufzeitsystem (*libc*) im Maschinenprogramm





## Granularität/Auflösung der Speicherzuteilung

- je nach Haupt-/Arbeitsspeicher (inkl. Halbenspeicher) ergibt sich für die Verwaltung ein **problemspezifischer Zerlegungsgrad** wie folgt:
  - Wort** ■ Platzierungseinheit für Hauptspeicher
    - abhängig von CPU: Vielfaches von **Byte**<sup>1</sup>
  - Zelle** ■ Platzierungseinheit für Halden- und Hauptspeicher
    - abhängig von Laufzeit-/Betriebssystem: 8 B, 16 B, 16/32 B
  - Kachel** ■ Platzierungs-, Lade- und Ersetzungseinheit für Arbeitsspeicher
    - abhängig von MMU: 512 B bis 1 GiB (typisch 4 KiB)
- diese **Granulate** werden zusammengefasst in **uniforme Segmente** mit Adresswerten, die einem Vielfachen der Granulatgröße entsprechen
  - wobei die Segmente die **Schutzseinheit** im logischen Adressraum bilden
  - bei entsprechender **Ausrichtung** (*alignment*) im Haupt-/Arbeitsspeicher

### Platzierungsstrategie

Die Verfahren dazu sind je nach Speicherauflösung grob kategorisiert in **segmentierte** und **gekachelte Speicherverwaltung**.

<sup>1</sup>Das kleinste adressierbare Speicherelement: historisch 5–36 Bits breit.

## Aspekte der Speicherzuteilung

### segmentierte ↔ gekachelte Speicherverwaltung

Segmente können verschieden groß sein, ihre jeweiligen Attribute sind Granularität, Adresse und Länge. Kacheln dagegen sind alle gleich groß, sie unterscheiden sich lediglich durch ihre Adressen.

- um die „Granulate“ im Hauptspeicher platzieren zu können, ist Buch über nicht zugeteilte Speicherbereiche zu führen
  - freie Speicherbereiche werden in einer **Löcherliste** (*hole list*) geführt
    - ein freier Platz ist Loch, Lücke, Hohlraum (*hole*) zwischen belegten Plätzen
    - wobei der für ein Listenelement benötigte Speicher das Loch selbst sein kann
      - ↳ Halbenspeicher: freie und belegte Plätze teilen denselben Adressraum ☺
  - die Liste ist sortiert nach Größe oder Adresse der vorhandenen Löcher
    - womit verschiedene Ziele bei der Zuteilungsstrategie verbunden sind (S. 19)
    - jedoch macht Sortierung nach Größe bei gekacheltem Speicher wenig Sinn
- wenn möglich, ist **Verschnitt** (*waste*) klein zu halten: **Zielkonflikt!**
  - i einen Platz fester Größe zuteilen ~> Segmente kacheln
    - vermeidet *externen* Verschnitt auf Kosten *internen* Verschnitts
  - ii bei Freigabe zwei in  $A_r$  angrenzende Löcher zu einem Loch verschmelzen

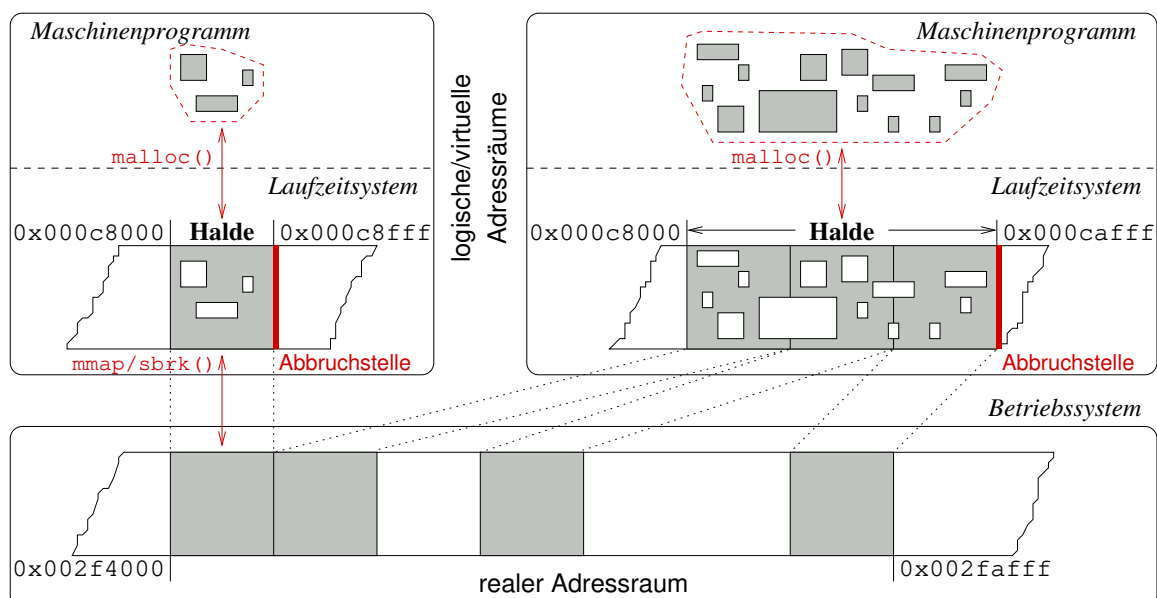
- **Sortierkriterien** der Freispeicherliste und damit verbundene **Ziele**:
  - best-fit* ■ zunehmende Größen, von vorne: **Verschnitt minimieren**
  - worst-fit* ■ abnehmende Größen, von vorne: **Suchaufwand minimieren**
  - beide Strategien machen die Einsortierung eines anfallenden Rests und somit einen zweiten Listendurchlauf notwendig
  - aus gleichem Grund ist Verschmelzung angrenzender Löcher zu einem großen Loch in beiden Fällen aufwendig
  - first-fit* ■ aufsteigende Adressen, von vorne: **Laufaufwand minimieren**
  - next-fit* ■ wie zuvor, jedoch ab letzter Fundstelle: **Verschnitt nivellieren**
  - Verschmelzung angrenzender Löcher zu einem großen Loch ist in beiden Fällen unaufwendig
  - beide Strategien tendieren dazu, große Löcher zu zerschlagen und dadurch mehr Verschnitt zu generieren (FF mehr als NF)

*Auch eine schwere Tür hat nur einen kleinen Schlüssel nötig. (Dickens)*

- die „Einfachheit“ macht *first-* und *next-fit* zu guten Kompromissen...
  - vgl. Anhang, S. 32–36



## Synergie bei der Speicherzuteilung



- das **Laufzeitsystem** verwaltet Speicher, der dem Adressraum eines einzelnen Maschinenprogramms vom Betriebssystem zugeteilt wurde
- das **Betriebssystem** verwaltet Speicher, der den Adressräumen aller Maschinenprogramme zugeteilt worden ist oder werden kann



*Speicherrückgabe durch free an das Betriebssystem ist nicht nötig, da bei virtuellem Speicher ungenutzter Speicher schon rückgewonnen wird — kolportiert die Informatikfolklore.*

- jede Implementierung virtuellen Speichers basiert auf **Schätzungen**
  - Rückgewinnung ungenutzten Speichers leistet die Ersetzungsstrategie
  - alle dazu bekannten Verfahren greifen auf **Heuristiken** zurück
  - ob Speicher endgültig ungenutzt ist, weil er frei ist, bleibt daher ungewiss
  - nur das Maschinenprogramm selbst kann darüber Gewissheit haben
- eine Folge daraus ist, dass **Programmierfehler** unentdeckt bleiben
  - Adressen zu ungenutztem Hauptspeicher sind im Adressraum noch gültig
  - nur Rückgabe ans Betriebssystem kann diese Adressen ungültig machen
- zudem verursacht diese Heuristik **nichtdeterministische Prozesse**
  - eine kontraproduktive Eigenschaft für (festen/harten) Echtzeitbetrieb
  - deshalb implementiert nicht jedes Betriebssystem virtuellen Speicher...

## Verschmelzung und Kompaktifizierung

Ist notwendig bzw. wünschenswert für große, rückgabefähige Löcher.

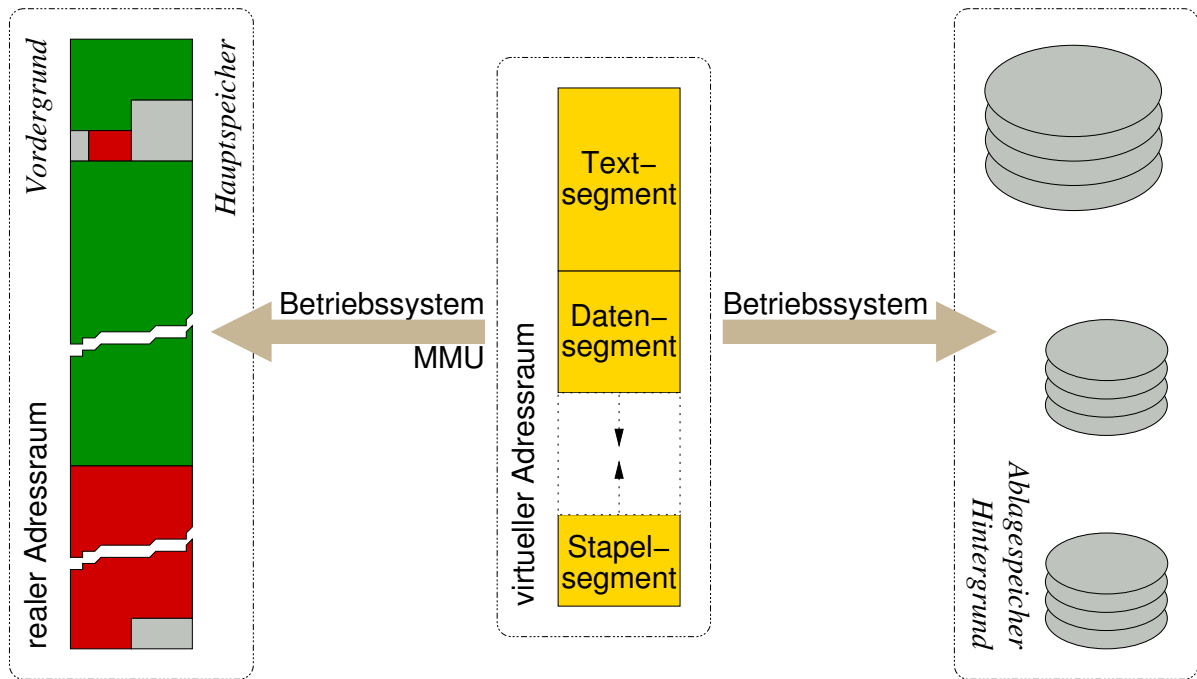


# Virtueller Speicher

*Eine **Betriebssystemtechnik**, die laufende (ggf. nichtsequentielle) Prozesse ermöglicht, obwohl ihre Maschinenprogramme samt Daten nicht komplett im Hauptspeicher liegen.*

- sobald und solange Spannung anliegt, erfordert die Befehlssatzebene (d.h., der reale Prozessor) einen kontinuierlichen Befehlsstrom
  - anderenfalls gelingt der Befehlsabruf- und -ausführungszyklus nicht
- aus welchen Programmabläufen sich dieser Befehlsstrom ergibt, ist für die Befehlssatzebene jedoch nicht von Bedeutung
  - Abläufe innerhalb eines realen/logischen Adressraums erfordern, dass die betreffenden Programme vollständig im Hauptspeicher vorliegen
    - jede Adresse muss auf ein im Hauptspeicher vorliegendes Datum abbilden
  - im Gegensatz zu Abläufen innerhalb eines virtuellen Adressraums, der die **partielle Abbildung** einer Adresse auf ein Datum ermöglicht (vgl. S. 13)
- die durch die **Lokalität** eines Prozesses definierte **Referenzfolge** gibt die Adressen vor, die auf den Hauptspeicher abzubilden sind
  - alle anderen Adressen bilden ab auf die Ablage (*swap area*)





- Abbildungseinheit ist eine **Seite** (*page*), die Strukturierungselement sowohl des logischen als auch des virtuellen Adressraums ist
  - sie passt exakt auf eine Kachel, auch **Seitenrahmen** (*page frame*)



## Abbildungstabelle

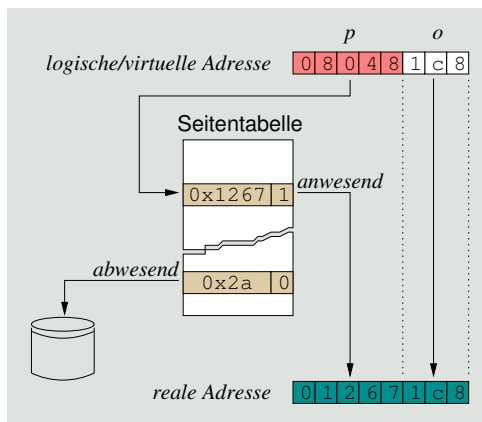
$f : \text{Seiten} \mapsto \text{Seitenrahmen d.h. Kacheln}$

- eine ein- oder mehrstufig organisierte **Seitentabelle** (*page table*)
  - ein Feld (*array*), Datentypenelement „**Seitendeskriptor**“ (*page descriptor*)
    - definiert durch das Betriebssystem, verarbeitet von der MMU
  - im Deutschen auch bezeichnet als „Seiten-Kachel-Tabelle“
- indiziert durch die **Seitennummer** (*page number*) einer Adresse
  - jede logische/virtuelle Adresse bildet ein Tupel  $A = (p, o)$
  - mit Seitennummer  $p$  und Versatz (*offset*)  $o$  innerhalb dieser Seite
    - Wertevorrat  $o = [0, 2^i - 1]$ , mit  $9 \leq i \leq 30$  (vgl. S. 17)
    - Wertevorrat  $p = [0, 2^{n-i} - 1]$ , mit  $32 \leq n \leq 64$
    - mit  $p$  als Indexwert liest die MMU den  $A$  abbildenden Seitendeskriptor
- der Seitendeskriptor enthält die für die Abbildung nötigen **Attribute**
  - Informationen über den gegenwärtigen Zustand der Seite
    - anwesend, referenziert, modifiziert, schreibbar, ausführbar, zugänglich, ...
  - Kachelnummer/-adresse im Hauptspeicher (falls anwesend, eingelagert) oder Blocknummer in der Ablage (falls abwesend, ausgelagert)
- jeder Zugriff auf eine abwesende, ausgelagerte Seite verursacht einen **Seitenfehler** (*page fault*)  $\rightsquigarrow$  Teilinterpretation des Zugriffs

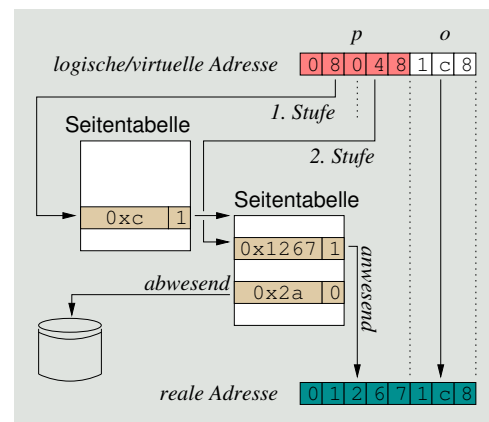


- angenommen, die CPU ruft folgenden Befehl zur Ausführung ab:  
0x080481c8 <+8>: call 0x8048300 <random> (vgl. S. 11)

- einstufige Abbildung



- zweistufige Abbildung (x86)



- *base/limit* Registerpaar (MMU) grenzt die Seitentabelle ein
- verschieden große Seitentabellen

- *base* Register (MMU) lokalisiert die Seitentabelle der 1. Stufe
- gleich große Seitentabellen

→ **Trap**, falls  $p \geq limit$  (li.) oder ungültiger/leerer Seitendeskriptor (beide)



## Gliederung

Einführung

Grundlagen

Speicherorganisation

Adressraum

Speicherverwaltung

Einleitung

Speicherzuteilung

Speichervirtualisierung

Zusammenfassung



- behandelt wurde die **Speicherorganisation** von Rechensystemen
  - Primär-, Sekundär- und Tertiärspeicher als die drei Hauptkategorien
  - Verfeinerungen sind Haupt-, Arbeitsspeicher und Ablage
  - Bausteine von Vorder- und Hintergrundspeicher zur Programmausführung
  - bilden Ebenen einer **Speicherpyramide** (auch: Speicherhierarchie)
- die für Speicherverwaltung relevante **Adressraumlehre** präsentiert
  - **Referenzfolgen** sind wichtig, um virtuellen Speicher zu begreifen
  - Bezugspunkt dabei ist die **Adressraumart**: real, logisch, virtuell
  - logischer und virtueller Adressraum sind zwei verschiedene Konzepte:
    - totale Abbildung  $f : A_l \rightarrow A_r$  von logischen auf realen Adressen
    - partielle Abbildung  $f : A_v \rightsquigarrow A_r$  von virtuellen auf realen Adressen
- Aufgaben der Speicherverwaltung sind in **Politiken** untergliedert
  - Platzierungs-, Lade- und Ersetzungsstrategie
  - erstere meint Speicherzuteilung, letztere beiden Speichervirtualisierung
  - virtueller Speicher ermöglicht Prozesse unvollständiger Programme
- das **Ausmaß** eines virtuellen Adressraums kann riesig sein
  - der Hauptspeicher im realen Adressraum ist indes verschwindend klein...



## Literaturverzeichnis I

- [1] CHASE, J. S. ; LEVY, H. M. ; FREELEY, M. J. ; LAZOWSKA, E. D.:  
Sharing and Protection in a Single-Address-Space Operating System.  
In: *Transaction on Computer Systems* 12 (1994), Nov., Nr. 4, S. 271–307
- [2] DALEY, R. C. ; DENNIS, J. B.:  
Virtual Memory, Processes, and Sharing in MULTICS.  
In: *Communications of the ACM* 11 (1968), Mai, Nr. 5, S. 306–312
- [3] KLEINÖDER, J. ; SCHRÖDER-PREIKSCHAT, W. :  
Prozesse.  
In: LEHRSTUHL INFORMATIK 4 (Hrsg.): *Systemprogrammierung*.  
FAU Erlangen-Nürnberg, 2015 (Vorlesungsfolien), Kapitel 6.1



## ■ Seitendeskriptor und Bildung einer „eingerahmten“ realen Adresse:<sup>2</sup>

```

1 #define PAGE_SIZE (1<<12) /* 4KB per page and page frame, resp. */
2
3 typedef struct page {
4     unsigned present:1; /* true iff in mainstore, false otherwise */
5     unsigned reserved:11; /* more page attributes (for further study in SP2) */
6     unsigned frame:20; /* page-aligned mainstore address: page frame number */
7 } page_t;
8
9 inline void *match(page_t *page, void *addr) {
10     return (void *)((page->frame * PAGE_SIZE) | ((size_t)addr & (PAGE_SIZE - 1)));
11 }

```

## ■ Seitentabelle und Ableitung des Seitendeskriptors aus einer Adresse:

```

12 typedef struct map {
13     page_t *table; /* page table base address */
14     size_t limit; /* last valid page table entry */
15 } map_t;
16
17 extern map_t map; /* page table base/limit register pair */
18
19 inline page_t *probe(map_t *map, void *addr) {
20     unsigned pano = ((unsigned)addr) / PAGE_SIZE;
21     if (pano <= map->limit) /* valid page number or address, resp. */
22         return &map->table[pano]; /* read page descriptor */
23     trap(SEGMENTATION_FAULT); /* invalid page number: raise exception to OS */
24 }

```

<sup>2</sup>Mit möglichem Blocknummernwertevorrat  $[0, 2^{31} - 1]$  für abwesende Seiten.



## Abbildungsfunktionen II

## ■ partielle Abbildung einer virtuellen Adresse, $f : A_V \rightsquigarrow A_R$ (vgl. S. 13)

```

1 void *v2r(void *addr) {
2     do {
3         page_t *page = probe(&map, addr);
4         if (page->present) /* placed in mainstore */
5             return match(page, addr);
6         trap(PAGE_FAULT); /* absent: raise exception to OS */
7     } while (1); /* retry mapping */
8 }

```

- gültige Seiten eines virtuellen Adressraums sind ein- oder ausgelagert
- sind sie ausgelagert, wird ein **Seitenfehler** (*page fault*) angezeigt
- Folge ist die **Seitenumlagerung** (*paging*) aus der Ablage (*swap area*) heraus und hinein in den Hauptspeicher

## ■ totale Abbildung einer logischen Adresse, $f : A_l \rightarrow A_r$ (vgl. S. 12)

```

9 void *l2r(void *addr) {
10     return match(probe(&map, addr), addr);
11 }

```

- im Unterschied zur partiellen Abbildung müssen die gültigen Seiten eines logischen Adressraums immer eingelagert sein
- Seitenfehler gibt es hier nicht, jedoch bleibt der **Speicherzugriffsfehler** (*segmentation fault*: vgl. S. 29, Zeile 23)





# Umfang eines virtuellen Adressraums

- mit  $N$  für die **Adressbreite** (einer virtuellen Adresse) in Bits:

$N$	Adressraumgröße ( $2^N$ Bytes)	Dimension			
16	65 536	64 kibi	$(2^{10})$	kilo	$(10^3)$
20	1 048 576	1 mebi	$(2^{20})$	mega	$(10^6)$
32	4 294 967 296	4 gibi	$(2^{30})$	giga	$(10^9)$
⋮			⋮		⋮
48	281 474 976 710 656	256 tebi	$(2^{40})$	tera	$(10^{12})$
64	18 446 744 073 709 551 616	16 384 pebi	$(2^{50})$	peta	$(10^{15})$

- ein einziger virtueller Adressraum kann so riesig sein, dass es schnell an Ablageplatz fehlt, um ausgelagerte Seiten zu speichern ☹️
- darüber hinaus können seine Adressen ewig gültig sein...

*A full 64-bit address space will last for 500 years if allocated at the rate of one gigabyte per second. [1, S. 272]*



# Haldenspeicher I

*first-fit*

```
1 #include <stddef.h>
2
3 typedef struct cell {
4     struct cell *next;
5     size_t size;
6 } cell_t;
7 extern cell_t *list;
8
9 extern cell_t *acquire(size_t);
10 extern size_t release(cell_t *);
```

- freien Speicher erwerben, einen passenden Bereich ausfindig machen:

```
11 cell_t *acquire(size_t want) {
12     size_t need = ((want + sizeof(cell_t) - 1) / sizeof(cell_t)) + 1;
13     cell_t **link = &list, *cell;
14
15     while ((cell = *link)) {
16         if (cell->size < need) /* one/another hole listed, check it */
17             link = &cell->next; /* bad luck, it's too small */
18         else { /* continue search... */
19             if (cell->size > need) { /* hole is large enough */
20                 cell->size -= need; /* shorten it */
21                 (cell + cell->size)->size = need; /* remember cell size */
22                 return (cell + cell->size + 1); /* deliver initial address */
23             }
24             *link = cell->next; /* hole fits exactly, remove from list */
25             return cell + 1; /* deliver initial address */
26         }
27     }
28     return 0;
29 }
30 }
```

*Next-fit führt einen last-Zeiger ein, der den zuletzt untersuchten Listeneintrag (*link*) vermerkt. Suchläufe beginnen bzw. enden bei last, mit Umgriff am Listenende auf den Listenanfang (*list*).*





- Speicher freigeben, wenn möglich mit Nachbarlöchern verschmelzen:

```

31 size_t release(cell_t *hole) {
32     size_t free = 0;
33
34     if (hole-- && hole->size) {           /* start at cell head (descriptor) */
35         cell_t **link = &list;
36
37         while (*link && ((*link)->next < hole)) /* find location on list */
38             link = &(*link)->next;          /* continue search... */
39
40         if (*link) {                       /* location found, try to merge */
41             if (((*link) + (*link)->size) == hole) /* preceding hole, merge */
42                 free = ((*link)->size += hole->size);
43             if ((hole + hole->size) == (*link)->next) /* succeeding hole, merge */
44                 free = ((*link)->size += (*link)->next->size);
45         }
46
47         if (!free) {                       /* merging was not possible, add hole to free list */
48             free = hole->size;               /* remember amount of freeing */
49             hole->next = *link;             /* link with successor hole */
50             *link = hole;                   /* enlist freed hole */
51         }
52     }
53     return free * sizeof(cell_t);          /* deliver number of bytes freed */
54 }

```

- 34 ■ neben der Zellengröße ist der Test auf ein **Kennzeichen** sinnvoll, um „falsche“ Zellen zumindest ansatzweise erkennen zu können
- als Platzhalter des Kennzeichens bietet sich das next-Attribut an



## Haldenspeicher III

- Nachbildung der funktionalen Eigenschaften von malloc und free:

```

1  #include "cell.h"
2
3  void *malloc(size_t want) {
4      cell_t *cell = acquire(want);        /* try request */
5      if (!cell) {                         /* failed, no hole found */
6          cell = enlarge(want);           /* ask for more memory */
7          if (cell)                       /* great, OS supplied */
8              cell = acquire(want);       /* retry request */
9      }
10     return cell;
11 }
12
13 void free(void *cell) {
14     size_t size = release((cell_t *)cell);
15     if (size)                             /* new free memory released */
16         reclaim(size);                    /* give operating system (OS) a hint */
17 }

```

- im Unterschied zum Original, teilt free dem Betriebssystem mit, dass Haldenspeicher endgültig rückgewonnen werden kann
- Informatikfolklore hält diesen Hinweis unnötig bei virtuellem Speicher



# Interaktion Maschinenprogramm/Betriebssystem I

## ■ UNIX-kompatible Betriebssystemschnittstelle nutzen:

```
1 #include <unistd.h>
```

## ■ dynamischen Speicher vergrößern, logischen Adressraum verlängern:

```
2 cell_t *enlarge(size_t want) {
3     size_t size = ((want + getpagesize() - 1) / getpagesize()) * getpagesize();
4     cell_t *cell = (cell_t *)sbrk(size);    /* new program break */
5
6     if ((int)cell != -1) {
7         cell->next = 0;                    /* only cell this section */
8         cell->size = size / sizeof(cell_t); /* make cell size */
9         release(cell + 1);                /* add section to free list */
10    }
11
12    return (int)cell == -1 ? 0 : cell;
13 }
```

- das Beispiel greift mit `sbrk` einen recht alten Ansatz auf,<sup>3</sup> der das Ende des Datensegments eines Maschinenprogramms verschiebt
  - nach hinten – um mehr Speicher vom Betriebssystem zu erhalten
  - nach vorne – um dem Betriebssystem Speicher zurück zu geben
- verändert wird die **Abbruchstelle** im Programm (*program break*), die Adresse der ersten Speicherstelle jenseits des gültigen Datensegments

<sup>3</sup>Nicht zuletzt, um Spielraum zum Selbststudium zu lassen. ☺



# Interaktion Maschinenprogramm/Betriebssystem II

## ■ Größenangabe auf Tauglichkeit zur Speicherrückgabe prüfen:

```
14 int adapted(size_t size) {
15     return ((size / getpagesize()) * getpagesize()) == size;
16 }
```

## ■ dynamischen Speicher zurückgeben, logischen Adressraum verkürzen:

```
17 void *reclaim(size_t size) {
18     if (adapted(size)) {
19         cell_t **link = &list;
20
21         while ((*link)->next != 0)    /* skip to end of data section */
22             link = &(*link)->next;    /* BTW: a tail pointer is a good idea... */
23
24         if (((*link)->size * sizeof(cell_t) == size)    /* hole size fits and */
25             && ((*link) + (*link)->size) == sbrk(0)) {    /* hole is before break */
26             void *hole = *link;    /* save return value */
27             sbrk(-((*link)->size * sizeof(cell_t)));    /* new program break */
28             *link = 0;    /* new list tail item */
29             return hole;
30         }
31     }
32     return 0;
33 }
```

- bei *first/next-fit* könnte die Adresse des letzten gelisteten Lochs eine neue Abbruchstelle im Maschinenprogramm markieren
- dazu muss die dem Loch folgende Adresse der aktuellen Abbruchstelle entsprechen und die Lochgröße muss Vielfaches der Kachelgröße sein

