Betriebssystemabstraktionen

Adressraum ☞ physikalische, logische oder virtuelle Adressen

Speicher ☞ im Vorder- bzw. Hintergrund

Datei ☞ langfristige (permanente) Speicherung von Informationen

Prozess ☞ feder-, leicht- oder schwergewichtige Aktivitätsträger

Darüberhinaus sind oftmals noch **Koordinations**- und **Kommunikationsmittel** zur Unterstützung der Interaktion nebenläufiger Prozesse verfügbar.

Adressraum

physikalischer Adressraum ☞ Hardware . . . . (Ebene $_2$)

- die Größe entspricht der Adressbreite der CPU: N Bit $\Rightarrow 2^N$ Bytes
- nicht alle Adressen sind gültig und zur Programmspeicherung verwendbar

logischer Adressraum ☞ Kompilierer, Binder, Betriebssystem . . . . (Ebene $_{5/4/3}$)

- definiert einen zusammenhängenden, linear adressierbaren Programmbereich
- alle Adressen sind gültig und zur Programmspeicherung verwendbar
- ist typischerweise (sehr viel) kleiner als die Adressbreite der CPU hergibt

virtueller Adressraum ☞ Betriebssystem . . . . (Ebene $_3$)

- ein logischer Adressraum, dessen Größe der Adressbreite der CPU entspricht

Physikalischer Adressraum

Toshiba Tecra 730CDT, 1996:

Adressbereich	Belegung
00000000–0009ffff	RAM
000a0000–000c7fff	System
000c8000–000dffff	keine
000e0000–000fffff	System
00100000–090fffff	RAM
09100000–fffdffff	keine
fffe0000–ffffffff	System

Logischer Adressraum (1)

- jedes Programm wird in einem eigenen logischen Adressraum ausgeführt
  - die $\left\{ \begin{array}{r} \text{Anfangs-} \\ \text{End-} \end{array} \right\}$ Adressen aller logischen Adressräume sind (meist) gleich
  - logische Adressen sind auf die gültigen physikalischen Adressen abzubilden
- die erforderliche *Adressabbildung* erfolgt (typischerweise) mehrstufig:

Programmadresse	$\leadsto$	logische Adresse
logische Adresse	$\leadsto$	physikalische Adresse

- im Gegensatz zu physikalischen Adressen sind logische Adressen mehrdeutig

Logischer Adressraum (2)

- in der Adress(raum)abbildung sind verschiedene <u>Ebenen</u> (✗ Kap. 5) involviert:

Entwicklungszeit	☞ Programmierer	☞ Ebene $_6$
Übersetzungszeit	☞ Kompilierer, Assemblierer	☞ Ebene $_{5/4}$
Bindezeit	☞ Binder	☞ Ebene $_4$
Ladezeit	☞ Lader	☞ Ebene $_3$
Laufzeit	☞ bindender Lader, MMU	☞ Ebene $_{3/2}$

- je später diese Abbildung durchgeführt wird, desto. . .

  ☞ höher das Abstraktionsniveau und geringer die Hardwareabhängigkeit
  ☞ höher der Systemaufwand und geringer der Spezialisierungsgrad

- der Zeitpunkt unterliegt einem „*trade off*“ zwischen Flexibilität und Effizienz

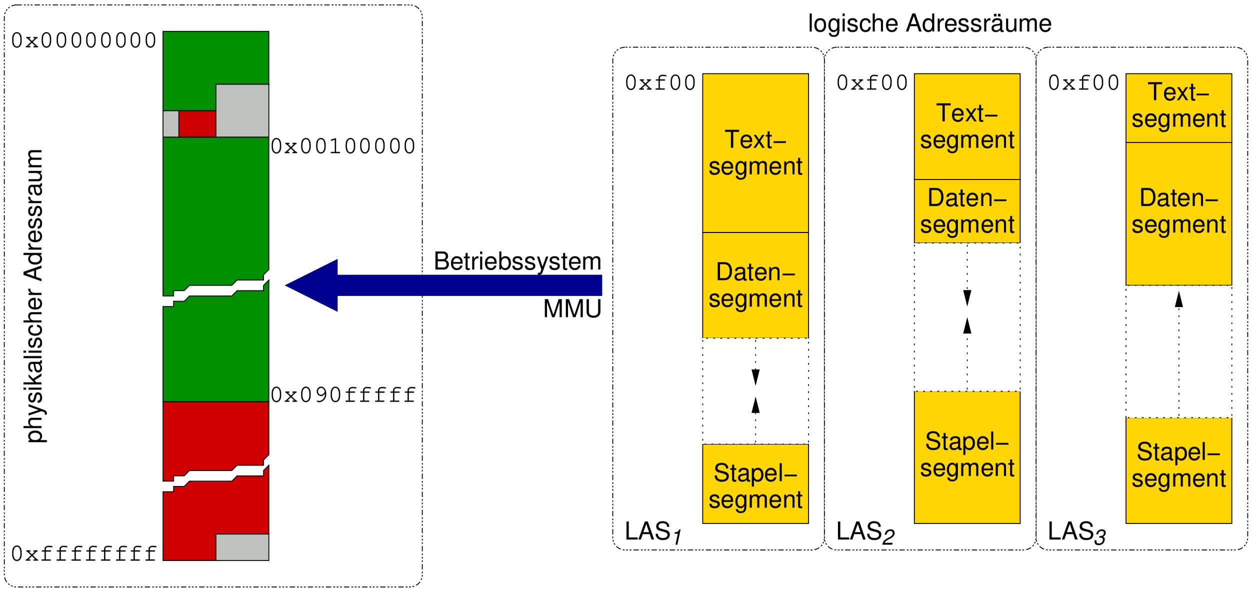
Adressraumsegmentierung

- gebräuchlich ist die logische Unterteilung des Adressraums in drei Segmente:

  Textsegment (*text segment*) enthält die Maschinenabweisungen der Ebene $_2$ (CPU) und andere Programmkonstanten ☞ statisch/dynamisch
  Datensegment (*data segment*) kapselt initialisierte Daten, globale Variablen und ggf. eine Halde (*heap*) ☞ statisch/dynamisch
  Stapelsegment (*stack segment*) beherbergt lokale Variablen, Hilfsvariablen und aktuelle Parameter ☞ dynamisch

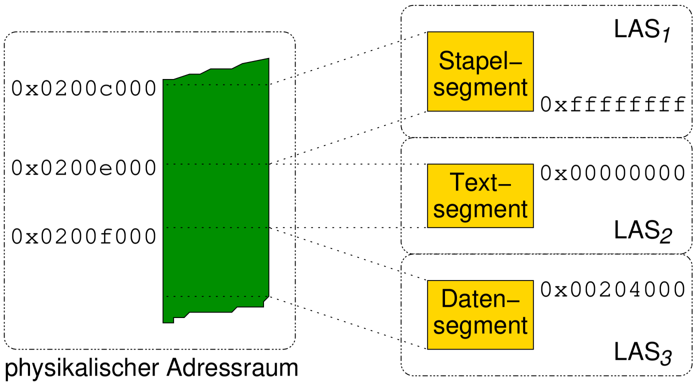
- typischerweise werden mehrere logische Adressräume nebeneinander verwaltet
  - Ebene $_3$ (Betriebssystem) sorgt für die *Adressraumabbildung*
  - Ebene $_2$ (*memory management unit*, MMU) sorgt für die *Adressumsetzung*

Adressraumabbildung durch Betriebssystem/MMU (1)



Relokation zur Laufzeit

- die Abbildung gleichzeitig vorhandener logischer Adressräume ist disjunktiv
  ☞ alle Segmente werden überschneidungsfrei im phys. Adressraum angeordnet



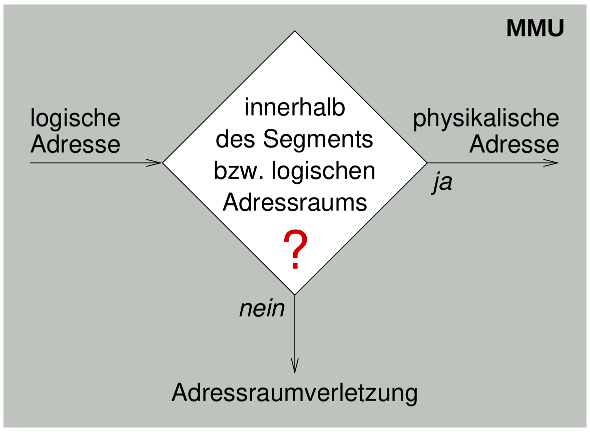
zur **Ladezeit** lokalisiert das BS den passenden Bereich im phys. Adressraum und programmiert die MMU

zur **Laufzeit** setzt die MMU jede logische Adresse um in die korrespondierende physikalische Adresse

- ein „Ausbruch“ aus einem Segment/logischen Adressraum ist zu verhindern

Isolation

- die MMU (Ebene $_2$) kapselt Adressräume und sichert Zugriffsschutz



- Adressraumverletzung führt zum Fehler
  - synchrone Programmunterbrechung
    ☞ *segmentation violation*..........
  - Programmabbruch ist die Folge
- Spiritus rector „Betriebssystem“
  - korrektes Funktionieren ist zwingend

- das Betriebssystem ist verantwortlich für die Integrität *aller* Adressräume

Schutzdomäne

- Hardware-gestützte Adressraumisolation erhöht die *Sicherheit* von Systemen

  safety (Sicherheit, Gefahrlosigkeit, Ungefährlichkeit) ☞ Schutz von Menschen und Sachwerten vor dem Versagen technischer Systeme
  - eine Fehlerausbreitung durch „Bitkipper“ z. B. im Speicher ist eingrenzbar
  - gleiches gilt für die Auswirkung von Berechnungsfehlern in Programmen

  security (Sicherheit, Schutz, Sorglosigkeit, Zuversicht) ☞ Schutz von Informationen und Informationsverarbeitung vor intelligenten Angreifern
  - „Eindringlinge“ können von den Programmen fern gehalten werden
  - Programmen wird ein Ausbruch aus ihren Adressräumen erschwert

- bei fehlerhaftem Betriebssystem nützt die beste Hardware/MMU nichts — letztlich ist ihr korrektes Funktionieren auch überhaupt nicht garantierbar **!!!**

Virtueller Adressraum (1)

- die Anzahl gültiger phys. Adressen dimensioniert einen logischen Adressraum
  - in Realität entspricht dies der Größe des gesamten Arbeitsspeichers (RAM)
  - der Speicherbedarf eines Programms kann diese Größe leicht übersteigen
  - umso problematischer: gleichzeitig nebeneinander bestehende Programme
- Überbelegung des Arbeitsspeichers im Mehrprogrammbetrieb ist Normalität
  - Einsatz von Hintergrundspeicher (Platte) entschärft die Engpasssituation:
    ☞ zur Zeit nicht benötigte Programmbereiche liegen auf der Platte
    ☞ nur die zur „Arbeitsmenge“ gehörenden Bereiche sind im RAM
- ein virtueller Adressraum ist dimensioniert durch die Adressbreite der CPU

Intermezzo

Adressbreite vs. Adressraumgröße

Adressbreite (N Bits)	Adressraumgröße (2^N Bytes)	Dimension		
16	65 536	64	Kilo	2^{10}
20	1 048 576	1	Mega	2^{20}
32	4 294 967 296	4	Giga	2^{30}
48	281 474 976 710 656	256	Tera	2^{40}
64	18 446 744 073 709 551 616	16 384	Peta	2^{50}

☞ Ein Rechner ist nur mit einem Bruchteil des von einer (zukünftigen) CPU adressierbaren Arbeitsspeichers wirklich bestückt!

Intermezzo — Adressraumdimensionen (1)

Die Aufgabe soll sein, den gesamten Adressraum byteweise `:-(` zu löschen. Dabei wird pro Byte eine Zugriffszeit von 1 Nanosekunde (ns) angenommen:

```
void clear () {
    char* p = 0;
    do *p++ = 0;
    while (p);
}
```

gcc -O6 -S

```
_clear:
        li      r2,0
        li      r0,0
L2:
        stb     r0,0(r2)
        addic.  r2,r2,1
        bne+    cr0,L2
        blr
```

PowerPC G4. Jeder Befehl ist vier Bytes lang. Die Löschschleife (`L2`) umfasst drei Befehle, die von der CPU aus dem Speicher zu lesen sind. Der Löschbefehl (`stb r0,0(r2)`) schreibt ein Byte mit dem Wert 0 in die nächste Speicherzelle. Jeder Schleifendurchlauf greift somit auf $4 \times 3 + 1 = 13$ Bytes zu.

[Warum funktioniert das so nicht?]

☞ Der Löschvorgang eines Bytes kostet (schlimmstenfalls) 13 ns!

Intermezzo — Adressraumdimensionen (2)

Adressraumgröße (Bytes)	Löschdauer	Einheit
2^{16}	0,000851968	Sekunden
2^{20}	0,013631488	Sekunden
2^{32}	55,834574848	Sekunden
2^{48}	42,351558995816	Tage
2^{64}	7604,251425615937	Jahre (ohne Schaltjahre)

Virtueller Speicher. Die zur Zeit nicht benötigten Programmbereiche bzw. Abschnitte des logischen Adressraums liegen im Hintergrundspeicher. Bei Bedarf werden diese Abschnitte „seitenweise" in den Vordergrundspeicher geholt, d. h., eingelagert. Angenommen, jede Seite ist 4 KB groß und die mittlere Zugriffszeit auf den Hintergrundspeicher (Platte), um eine Seite einzulagern, liegt bei 5 ms. Wie hoch ist dann die mittlere Zugriffszeit auf jedes einzelne Byte? 1,220703125 μs. Wie lange dauert dann jeweils der Löschvorgang für die Adressräume? $>$ 1,5 Stunden (2^{32}).

Virtueller Adressraum (2)

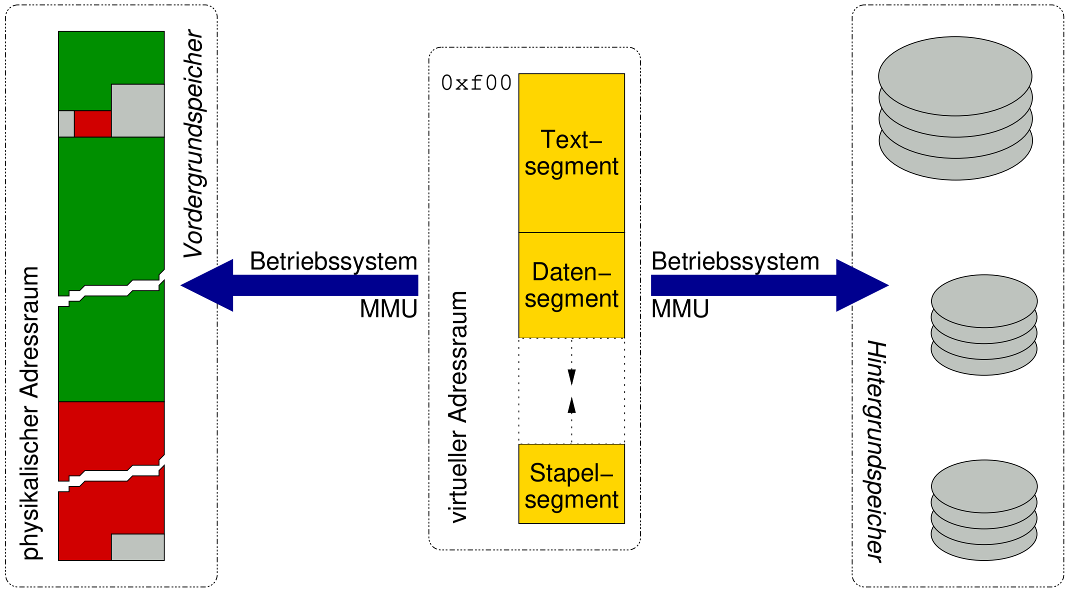
- die *Adressabbildung* (logischer Adressräume) ist um eine Stufe zu erweitern:

Programmadresse	$\rightsquigarrow$	logische Adresse
logische Adresse	$\rightsquigarrow$	virtuelle Adresse
virtuelle Adresse	$\rightsquigarrow$	physikalische Adresse

- Zugriffe über virtuelle Adressen können implizit Ein-/Ausgabe zur Folge haben
  - die MMU lässt nur gültige Zugriffe auf den *Vordergrundspeicher* zu
  - ggf. werden Zugriffe dann partiell interpretiert vom Betriebssystem (☞ *trap*)
  - Ergebnis ist die Einlagerung des „Operanden" vom *Hintergrundspeicher*
  - dies kann zur Auslagerung anderer Vordergrundspeicherinhalte führen

- die Ein-/Auslagerung erfolgt typischerweise auf Seitenbasis (☞ *demand paging*)

Adressraumabbildung durch Betriebssystem/MMU (2)



UNIX Systemaufrufe (1)

`pa = mmap (addr, len, prot, flags, fd, offset)` legt ein Segment über einen (gemeinsamen) Speicherbereich oder eine (gemeinsame) Datei

- für den aufrufenden Prozess ändert sich seine *Adressraumabbbildung*
- die zurück gelieferte Adresse, `pa`, identifiziert den Segmentanfang
- mit `addr` $= 0$ erhält das System Freiheit über die Bestimmung von `pa`[24]
- ein an `pa` ggf. vorher liegendes Segment wird ausgeblendet (`munmap(2)`)

`ok = munmap (addr, len)` entfernt die Abbildung für das Segment

[24] Als „Nebeneffekt" ist so die Erzeugung eines Speichersegments vorgegebener Länge (`len`) möglich.

Speicher

- die Abstraktion „Speicher" kommt mit zwei fundamentalen Ausprägungen:

  Vordergrundspeicher auch *Hauptspeicher* (RAM)
  - der entsprechend bestückte (RAM-) Bereich im physikalischen Adressraum
  - in ihm erfolgt die Ausführung der Programme („von Neumann Rechner")
  - kann größer sein als der physikalische Adressraum (*bank switching*)

  Hintergrundspeicher auch *Massenspeicher* (Band, Platte, CD, DVD)
  - die über die *Rechnerperipherie* (E/A-Geräte) angeschlossenen Bereiche
  - dient der Datenablage und Implementierung virtueller Adressräume
  - ist größer als der physikalische Adressraum: *Petabytes* (2^{50} bzw. 10^{15})

- die Differenzierung kann „nach aussen" unsichtbar sein (☞ Multics [8])

Speicherverwaltung

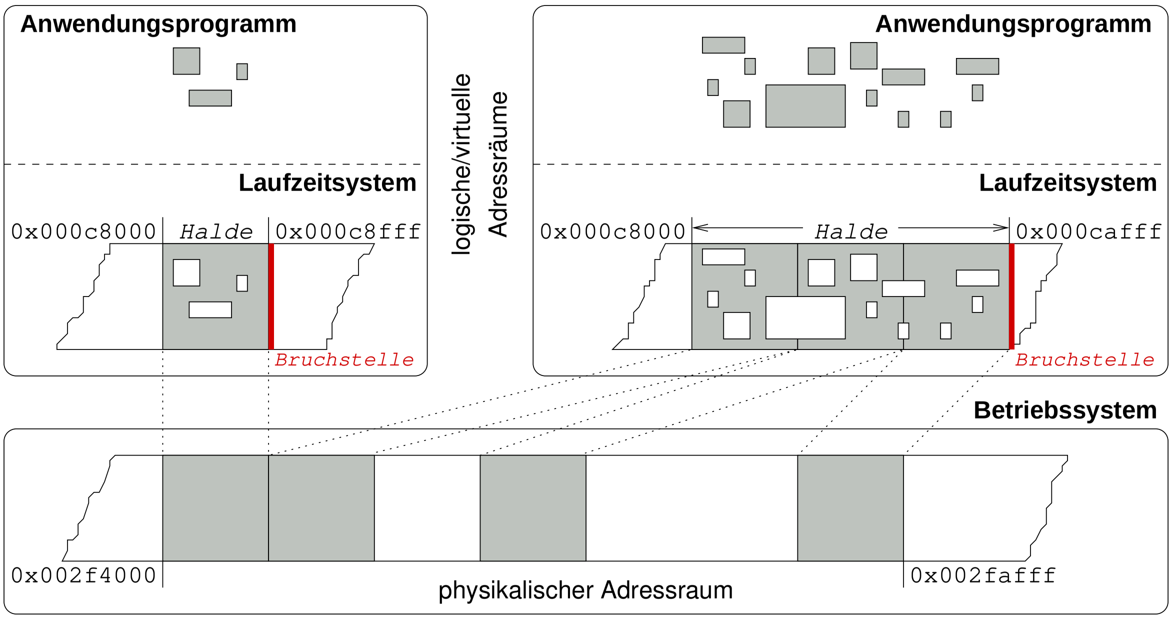
- die Verwaltung von *Hauptspeicher* erfolgt typischerweise auf zwei Ebenen:

  das **Laufzeitsystem** bzw. die Bibliotheksebene verwaltet den Speicher (lokal) innerhalb eines logischen/virtuellen Adressraums
  - Speicherblöcke können von sehr feinkörniger Struktur/Größe sein
  - Bedürfnisse von Programmiersprachen bestimmen die Verfahrensweisen

  das **Betriebssystem** verwaltet den im physikalischen Adressraum (global) vorrätigen Speicher
  - Speicherblöcke sind üblicherweise von grobkörniger Struktur/Größe
  - Arten des Rechnerbetriebs üben starken Einfluss auf Verfahrensweisen aus

- „*separation of concerns*“ [13] — beide Ebenen/Systeme ergänzen sich einander

Synergie bei der Speicherverwaltung



Speicher — C Bibliotheksfunktionen

`ptr = malloc (size)` legt einen (ausgerichteten) Speicherblock an

- Resultat ist die Anfangsadresse eines Blocks von mindestens `size` Bytes

`ptr = realloc (addr, size)` verkleinert/vergrößert den Speicherblock

- bei Verkleinerung steht der Rest ggf. der Wiedervergabe zur Verfügung

`free (ptr)` macht den Speicherblock zur Wiedervergabe verfügbar

- der Speicherbereich wird nicht ans Betriebssystem zurück gegeben **!**

Speicher — UNIX Systemaufrufe (2)

Die Festlegung einer neuen „Bruchstelle“[25] für das Datensegment eines Prozesses bewirkt die Veränderung der diesem Segment zugeordneten Speichermenge. Die Bruchstelle kann dabei eine vom System vorgegebene Größe nicht überschreiten.

`addr = brk (brkval)` setzt die Bruchstelle auf den angegebenen Wert

`addr = sbrk (incr)` addiert den angegebenen Wert zur Bruchstelle

- beim negativen Summanden (`incr`) wird das Datensegment verkleinert
- der vor der Veränderung gültige Wert der Bruchstelle wird zurück geliefert

☞ `getrlimit(2)`, `mmap(2)`

[25] Der „*break value*“, d. h. die der gegenwärtigen Endadresse des Datensegments folgende Speicheradresse.

Datei

Da'tei allgemein eine Sammlung von Daten; im Kontext von Rechensystemen:

☞ eine *zusammenhängende, abgeschlossene Einheit von Daten*
☞ eine „beliebige" Anzahl eindimensional adressierter Bytes

- die „Datei" als Synonym für anhaltende (langfristige) Datenspeicherung
  - *persistente Speicherung* der Daten ist allerdings nicht zwingend:
    ☞ „*RAM Disk*", „Rohr" (*pipe*), „*raw device*"
- nicht-flüchtige, blockorientierte Speichermedien (Platten, Magnetbänder) kommen bevorzugt zum Einsatz

Arten von Dateien

ausführbare Dateien Binär- und Skriptprogramme

- Dateien, die von einem Prozessor ausführbaren *Programmtext* enthalten

  Binärprogramm ☞ CPU, FPU, MCU, JVM, . . . , Basic, Lisp, Prolog
  Skriptprogramm ☞ `perl(1)`, `python(1)`, {a,ba,c,tc}`sh(1)`, `tcl(n)`

- der Prozessor kann in Hard-, Firm- und/oder Software realisiert sein

nicht-ausführbare Dateien Text-, Bild- und Tondateien

- Dateien, die von einem Prozessor verarbeitbare *Programmdaten* enthalten
  ☞ .{`doc`, `fig`, `gif`, `jpg`, `mp3`, `pdf`, `tex`, `txt`, `wav`, `xls`, . . . }
  ☞ .{`a`, `c`, `cc`, `f`, `F`, `h`, `l`, `o`, `p`, `r`, `s`, `S`, `y`, . . . }
- der Prozessor liegt als Programmtext (zumeist als ausführbare Datei) vor