Überblick

Betriebssystemabstraktionen
- Adressraum
- Speicher
- Datei
- Namensraum
- Prozess
- Koordinationsmittel
- Zusammenfassung

Adressraumkonzepte und virtuelle Maschinen

physikalischer Adressraum (Hardware) . . . . . Ebene 2

- ist durch die jeweils gegebene Hardwarekonfiguration definiert
- nicht alle Adressen sind gültig, zur Programmspeicherung verwendbar

logischer Adressraum (Kompilierer, Binder, Betriebssystem) . . . Ebene 5/4/3

- abstrahiert von Aufbau/Struktur des Haupt- bzw. Arbeitsspeichers
- alle Adressen sind gültig und zur Programmspeicherung verwendbar

virtueller Adressraum (Betriebssystem) . . . . . Ebene 3

- auf Vorder- und Hintergrundspeicher abgebildeter log. Adressraum
- erlaubt die Ausführung unvollständig im RAM liegender Programme

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

Je nach Hardwarekonfiguration hat der physikalische Adressraum eines Rechners mehr oder weniger viele bzw. große und nicht verwendbare Lücken.

Logischer Adressraum

Ausführungsdomäne von Prozessen im Mehrprogrammbetrieb

Illusion von einem eigenen (nicht zwingend linearen) Adressraum für jedes im Arbeitsspeicher vollständig vorliegende Programm

- die Anfangsadressen aller logischen Adressräume sind (meist) gleich
  - festgelegt durch eine Systemkonstante (Übersetzer, Binder, Lader)
- die Endadressen sind variabel, jedoch nach oben begrenzt
  - bestimmt durch die Programmlängen bzw. Hardwarefähigkeiten

Adressabbildung (engl. *address mapping*) erfolgt mehrstufig:

$$\begin{aligned} \text{Programm} &\mapsto \text{logischer Adressraum} \\ \text{logischer Adressraum} &\mapsto \text{physikalischer Adressraum} \end{aligned}$$

☞ logische Adressen sind mehrdeutig, physikalische dagegen eindeutig

Logischer Adressraum (Forts.)

Abbildungszeitpunkte

Adress(raum)abbildung kann auf verschiedenen Ebenen erfolgen:

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

Zielkonflikt (engl. *trade-off*) in Bezug auf Flexibilität und Effizienz

- je später die Abbildung durchgeführt wird, desto...
  - höher das Abstraktionsniveau und geringer die Hardwareabhängigkeit
  - höher der Systemaufwand und geringer der Spezialisierungsgrad

Verantwortlichkeiten bei der Adressraumabbildung

Zusammenspiel von Betriebssystem und Hardware/MMU

Betriebssystem (Ebene $_3$): Adressraumabbildung zur Ladezeit

- der Lader fordert Betriebsmittel zur Programmausführung an
  - Arbeitsspeicher und Adressraumdeskriptoren, je nach Bedarf/MMU
  - einen Prozess
- Verwaltungsinformationen für die MMU werden aufgesetzt
  - die physikalischen Ladeadressen in die Deskriptoren eintragen
  - ggf. spezielle Attribute (z.B. lesen, schreiben, ausführen) zuordnen
- der neue Prozess wird der Einplanung (engl. *scheduling*) zugeführt

Hardware/MMU (Ebene $_2$): Adressumsetzung zur Laufzeit

- Verwendung der in den Deskriptoren gespeicherten Informationen

Verantwortung trägt allein das Betriebssystem, die MMU führt nur aus

Segmentierung eines logischen Adressraums

Logische Unterteilung zur effektiveren Programmverwaltung

Textsegment (engl. *text segment*)

- Maschinenanweisungen (Ebene $_{2/3}$) und andere Programmkonstanten
- statische oder dynamische Größe, je nach Betriebssystem
- ggf. gemeinsame Verwendung für mehrere Prozesse (engl. *shared text*)

Datensegment (engl. *data segment*)

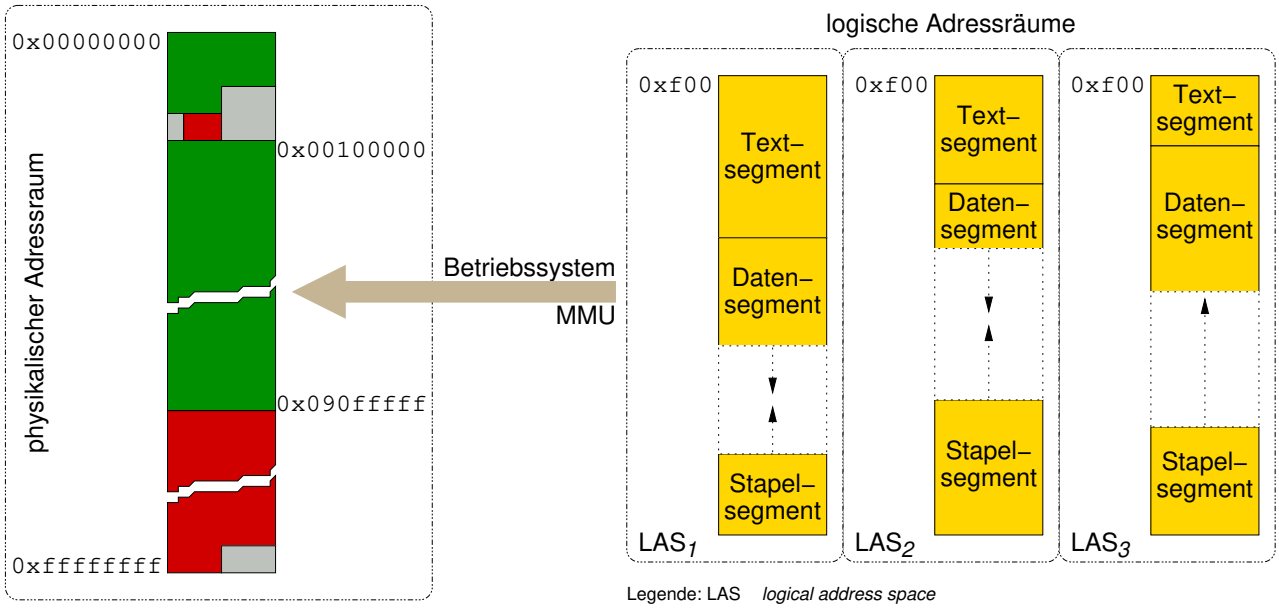
- initialisierte Daten, globale Variablen und ggf. die Halde (engl. *heap*)
- statische oder dynamische Größe, je nach Betriebssystem

Stapelsegment (engl. *stack segment*)

- lokale Variablen, Hilfsvariablen und aktuelle Parameter
- dynamische Größe

Adressraumabbildung auf Ebene $_3$

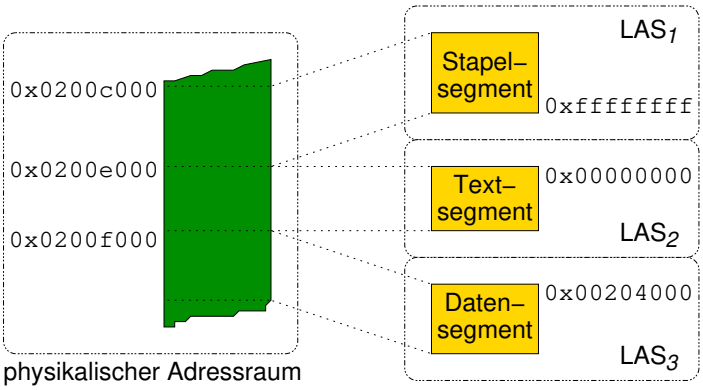
Betriebssystem und MMU implementieren logische Adressräume ☞ 7-14



☞ Segmente müssen nicht angrenzend im logischen Adressraum liegen

Disjunktive Abbildung zur Ladezeit

Betriebssystem ordnet Segmente überschneidungsfrei im physikalischen Adressraum an



Kontrolliert vom BS ist eine gemeinsame Verwendung (engl. *sharing*) von Segmenten möglich, nämlich durch absichtliche Überschneidung im Arbeitsspeicher.

Verletzung der Segmentierung (engl. *segmentation violation*) wird durch die MMU verhindert und bewirkt einen Programmabbruch (S. 6-34):

$$0 \leq Adresse_{log} - Adresse_{min} < Länge(Segment), \text{ sonst Trap}$$

- Konstante $Adresse_{min}$ bestimmt den Anfang eines log. Adressraums

Adressrelokation zur Laufzeit

MMU wandelt jede logische Adresse im Abrufzyklus (engl. *fetch cycle*) der CPU um

Veränderung einer logischen Adresse um eine Relokationskonstante: (Prinzip)

$$Adresse_{phy} = Adresse_{log} - Adresse_{min} + Basis(Segment)$$

- $Basis(Segment)$ ist die Ladeadresse im phys. Adressraum
  - $Adresse_{log} - Adresse_{min}$ relativiert zu Null
    - ist daher auch relative Adresse in Bezug auf $Basis(Segment)$
  - anschließende Addition „verschiebt“ den relativierten Wert
- die Ladeadresse eines Segments ist gleichfalls Relokationskonstante
  - für alle relativ(iert)en Adressen innerhalb von *Segment*

Relokation erfolgt nur bei unverletzter Segmentierung (S. 6-34)

Logischer Adressraum als Schutzdomäne

Robustheit von Softwaresystemen verbessern

Adressraumisolation, eine Maßnahme zur Erhöhung von Sicherheit...

safety Schutz von Menschen und Sachwerten vor dem Versagen technischer Systeme

- Berechnungsfehler oder „Bitkipper“ abfangen
- allgemein (bei BS): Fehlerausbreitung eingrenzen

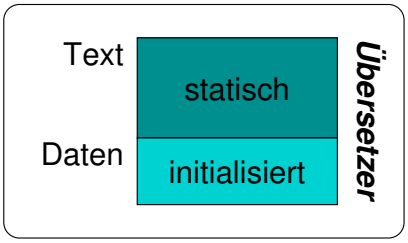
security Schutz von Informationen und Informationsverarbeitung vor „intelligenten“ Angreifern

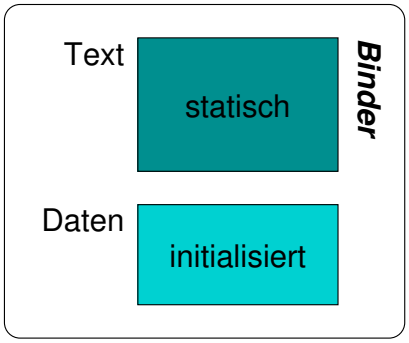
- Adressraumausbrüche erschweren/verhindern
- allgemein (bei BS): Eindringlinge fern halten

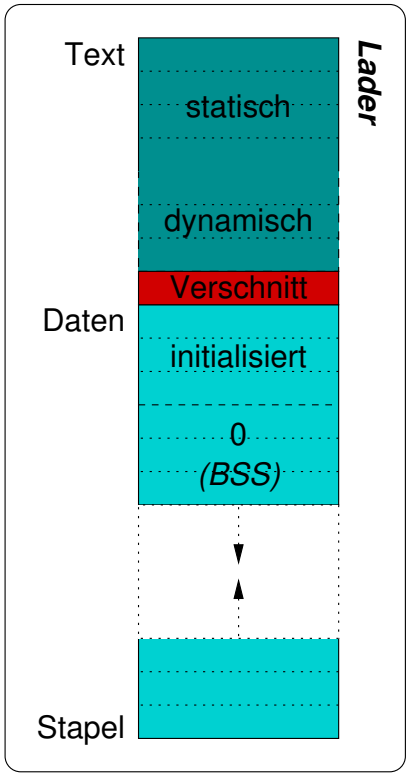
...in Rechensystemen, die im Mehrprogrammbetrieb gefahren werden

UNIX Segmentierung

Dienstprogramm (engl. *utility*) basierter seitennummerierter Ansatz







Linux, MacOS, SunOS

- Übersetzer generieren Text- und Datensegmente aus dem Quellprogramm
- Binder packen Text/Daten aus Bibliotheken dazu und hinterlassen ggf. seitenbündige Segmente
- Lader bringen die statischen Segmente in den RAM, fügen dynamische Text-/Stapelsegmente hinzu und setzen BSS (engl. *block started by symbol*) auf 0

Virtueller Adressraum

Grad des Mehrprogrammbetriebs (engl. *degree of multiprogramming*) erhöhen

Illusion von einem eigenen (nicht zwingend linearen) Adressraum für jedes im Arbeitsspeicher ggf. unvollständig vorliegende Programm

- Erweiterung bzw. Spezialisierung des logischen Adressraums
- meist verbreitet ist die Seitenüberlagerung (S. 6-53)
- Adressraumzugriffe können E/A (Hintergrundspeicher) implizieren

Adressabbildung (engl. *address mapping*) erfolgt mehrstufig:

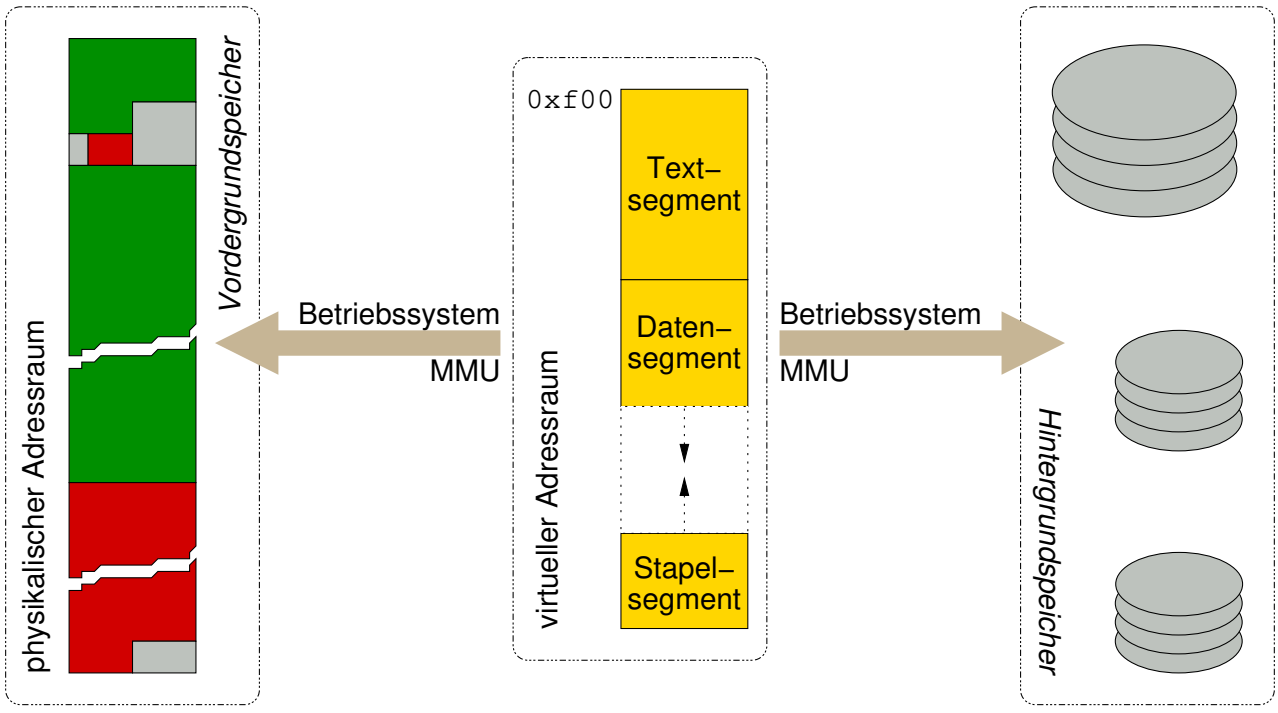
Programm	$\mapsto$	logischer Adressraum
logischer Adressraum	$\mapsto$	virtueller Adressraum
virtueller Adressraum	$\mapsto$	physikalischer Adressraum

☞ virtuelle Adressen „erben“ die Mehrdeutigkeitseigenschaft log. Adressen

Adressraumabbildung auf Ebene 3

Betriebssystem und MMU implementieren virtuelle Adressräume

☞ 7-8



Umfang eines virtuellen Adressraums

Adressbreite einer CPU sagt wenig aus über den Arbeitsspeicher eines Rechners

Adressbreite von N 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})

Rechner sind ggf. nur mit einem Bruchteil des von einer CPU adressierbaren Arbeitsspeichers wirklich bestückt!

Intermezzo — Metrisches System und Informationstechnik

Internationales Einheitensystem (frz. *Système International d'Unités*, SI)

Begriffe des metrischen Systems wurden bedenkenlos übernommen

- noch schlimmer: sie werden inkonsistent verwendet

Medium	Einheit	
	dual	*dezimal*
RAM, ROM, CD	×	
Flash, HD, DVD		×
Floppy	×	×

Floppy Disk. Der Zugriff auf das Medium erfolgt sektorweise. Die Größe eines Sektors wird als Zweierpotenz angegeben, die Anzahl der Sektoren kommt als Zehnerpotenz.

Abweichungen

kibi ↔ kilo	2,4%
mebi ↔ mega	∼ 4,8%
gibi ↔ giga	∼ 7,3%
tebi ↔ tera	∼ 9,9%
pebi ↔ peta	∼ 12,6%

Standardisierung [43] erfolgte erst sehr spät (Ende der 90er) — zu spät...

Experiment zur Adressraumgröße

Aufgabe eines Prozesses soll es sein, seinen Adressraum byteweise zu löschen

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

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

Bei 1 ns Zugriffszeit dauert das Löschen eines Bytes 13 ns!

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 $3 \times 4 + 1 = 13$ Bytes zu.

Experiment zur Adressraumgröße (Forts.)

Virtueller Speicher kann die Programmausführung verlangsamen

Größe	Laufzeit	
2^{16}	851.968 Mikrosekunden	
2^{20}	13.631 Millisekunden	
2^{32}	55.835 Sekunden	
⋮	⋮	← heute
2^{48}	42.352 Tage	
2^{64}	7604.251 Jahre	(ohne Schaltjahre)

Virtueller Speicher: Die zur Zeit nicht benötigten Bereiche eines virtuellen Adressraums liegen im Hintergrundspeicher. Bei Bedarf werden diese „seitenweise“ in den Vordergrundspeicher eingelagert. Angenommen, jede Seite ist 4 KiB groß und die mittlere Zugriffszeit des Hintergrundspeichers (Platte), um eine Seite einzulagern, liegt bei 5 ms. Damit kostet ein Bytezugriff durchschnittlich 1,2 μs! Der Löschvorgang eines 2^{32} Bytes umfassenden Adressraums würde somit bereits mehr als 1,5 Stunden dauern!

UNIX Systemfunktionen

Laufzeit- bzw. Betriebssystem

Linux, MacOS, SunOS

```
pa = mmap(addr, len, prot, flags, fd, offset)
ok = munmap(addr, len)
ok = mlock(addr, len)
ok = munlock(addr, len)
ok = mprotect(addr, len, prot)
ok = madvise(addr, len, behav)
ps = getpagesize()
   ⋮
```


Speicherkonzepte und -medium

Kurz-, mittel- und langfristige Informationsspeicherung

Vordergrundspeicher: Haupt- oder Arbeitsspeicher (RAM)

- entsprechend bestückter Bereich im physikalischen Adressraum
- Zentralspeicher zur Programmausführung („von Neumann Rechner“)
- kann phys. Adressraum überschreiten: Speicherbankumschaltung
- kurzfristige Speicherung, Zugriffszeiten im ns-Bereich

Hintergrundspeicher: Massenspeicher (Band, Platte, CD, DVD)

- über Rechnerperipherie (E/A-Geräte) angeschlossene Bereiche
- dient der Datenablage und Implementierung virtueller Adressräume
- ist größer als der phys. Adressraum: Petabytes (2^{50} bzw. 10^{15})
- mittel- bis langfristige Speicherung, Zugriffszeiten im ms-Bereich

Virtualisierung kann „Zugriffstransparenz“ mit sich bringen (Multics [14])

Speicherverwaltung (engl. *memory management*)

Symbiose von Laufzeit- und Betriebssystem

Laufzeitsystem (bzw. Bibliotheksebene) verwaltet den lokal vorrätigen Speicher eines logischen/virtuellen Adressraums (☞ Aufgabe 4)

- Speicherblöcke können von sehr feinkörniger Struktur/Größe sein
  - einzelne Bytes bzw. Verbundobjekte
- Verfahrensweisen orientieren sich (mehr) an Programmiersprachen

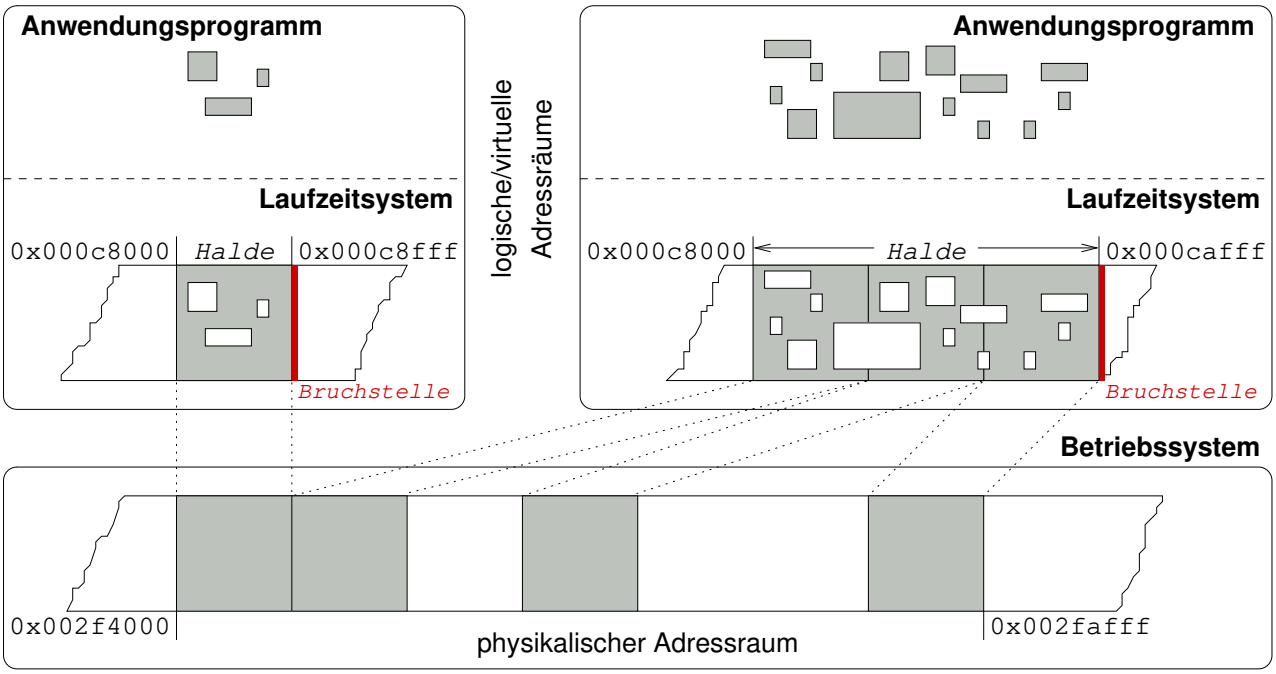
Betriebssystem verwaltet den global vorrätigen Speicher (d.h. den bestückten RAM-Bereich) des physikalischen Adressraums

- Speicherblöcke sind üblicherweise von grobkörniger Struktur/Größe
  - z.B. eine Vielfaches von Seiten
- Verfahrensweisen fokussieren auf Benutzer- bzw. Systemkriterien

Trennung von Belangen (engl. *separation of concerns* [44])

Synergie bei der Speicherverwaltung

Betriebssystemaufruf als „Ausnahme“



UNIX Systemfunktionen

Laufzeitsystem — C Bibliothek

Linux, MacOS, SunOS

(☞ Aufgabe 4)

```
ptr = malloc(size)
ptr = valloc(size)
ptr = calloc(count, size)
ptr = realloc(ptr, size)
      ⋮
free(ptr)
```

Freigabe (`free()`) von Speicher hat nur lokale Signifikanz

- keine freiwillige Rückgabe ans Betriebssystem
- die Wiedergewinnung freigegebener Bereiche erfolgt nur bei Beendigung des Programms und/oder auf Basis virtuellen Speichers

UNIX Systemfunktionen

Überbleibsel vergangener Systeme mit nur einem expandierbaren Adressraumsegment

Linux, MacOS, SunOS

```
addr = brk(brkval)
addr = sbrk(incr)
```

Festlegung einer neuen „Bruchstelle“ (engl. *break value*) für das Datensegment eines Prozesses

- verändert die diesem Segment zugeordnete Speichermenge
- kann eine vom System vorgegebene Größe nicht überschreiten
- ist die der Endadresse des Datensegments folgende Speicheradresse

Aufruf erfolgt im Zuge von `*alloc()`, nicht jedoch `free()`

Langfristige Datenspeicherung

Abstraktion von Informationen tragenden Betriebsmitteln

Da'tei (engl. *file*) Sammlung von Daten, eine...

- zusammenhängende, abgeschlossene Einheit von Daten
- „beliebige" Anzahl eindimensional adressierter Bytes

Dauerhaftigkeit von Dateien ist eine Frage des Speichermediums:

nicht-flüchtige Datenträger	Platte, Band, CD, DVD, ..., EEPROM
flüchtige Datenträger	RAM

- die Datei selbst ist ein durchaus unbeständiges Gebilde

Kommunikationsmittel für kooperierende Prozesse

- im Sinne einer Röhre (engl. *pipe*) zum Informationsaustausch

Arten von Dateien

Unterscheidung von Programmtext und Programmdaten

ausführbare Dateien: Binär- und Skriptprogramme

- von einem Prozessor ausführbarer Programmtext

Binär	$\rightsquigarrow$	CPU, FPU, MCU, JVM, ..., Basic, Lisp, Prolog
Skript	$\rightsquigarrow$	`perl(1)`, `python(1)`, {a,ba,c,tc}`sh(1)`, `tcl(n)`

- der Prozessor liegt in Hard-, Firm- und/oder Software vor

nicht-ausführbare Dateien: Text-, Bild- und Tondaten

- von einem Prozessor verarbeitbare Programmdaten
  - .{`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 in Form von Programmtext vor

Bezeichnung von Dateien

Symbolische und numerische Dateiadressen

Dateien sind „von aussen“ über symbolische Adressen erreichbar...

- benutzerdefinierter Name von beliebiger aber maximaler Länge
- auch als Dateiname (engl. *file name*) bekannt
  - wird ggf. vom Betriebssystem (teilweise) interpretiert

...„nach innen“ besitzt jede Datei eine numerische Adresse

- systemdefinierte Kennung einer Datenstruktur der Dateiverwaltung
- identifiziert den sogenannten Dateikopf (engl. *file head*)

☞ symbolische und numerische Dateiadresse bilden ein (festes) Paar

Erweiterung eines Dateinamens

Anreicherung um semantische Information

Dateinamensuffix (engl. *file extension*): eine meist durch einen Punkt vom Dateinamen abgegrenzte symbolische Erweiterung des Dateinamens

- liefert einen Hinweis auf das Dateiformat bzw. den Dateitypen

`.doc`	Textdokumente	MS-Word	
`.fm`		Framemaker	`maker(1)`
`.tex`		LaTeX	`latex(1)`
`.h`	Programme	Präprozessor	`cpp(1)`
`.c`		Kompilierer	`cc(1)`
`.s`		Assemblierer	`as(1)`
`.o`		Binder	`ld(1)`

- ist Dienstprogrammen und/oder dem Betriebssystem bekannt
  - bei UNIX die Dienstprogramme, bei Windows das Betriebssystem

Dateikopf und Dateikopfnummer

Systeminterne Verwaltungdaten einer UNIX-Datei

(☞ Aufgabe 5)

„*inode*“ (bzw. „*i-node*“, 1. Edition, 1971) enthält Dateiattribute:

- Eigentümer (*user ID*)
- Gruppenzugehörigkeit (*group ID*)
- Typ (reguläre/spezielle Datei)
- Rechte (lesen, schreiben, ausführen; Eigentümer, Gruppe, „Welt“)
- Zeitstempel (letzter Zugriff, letzte Änderung [Typ, Zugriffsrechte])
- Anzahl der Verweise („*hard links*“)
- Größe (in Bytes)
- Adresse(n) der Daten auf dem Speichermedium

„*inode number*“, Index in eine Tabelle von Dateiköpfen („*inode table*“):

- die numerische Adresse der Datei (innerhalb des Dateisystems)

Dateityp

Dateien zur Abstraktion von Daten, Geräten und Kommunikationsmitteln

reguläre Datei (engl. *regular file, ordinary file*)

- problemorientiertes, eindimensionales Bytefeld

spezielle Datei ein „Sammelsurium“ von (UNIX) Konzepten:

- Verzeichnis (engl. *directory*)
  - Katalog von regulären und/oder speziellen Dateien
- Gerätedatei (engl. *device file*)
  - Zugang zu zeichen-/blockorientierten Geräte(treiber)n
- symbolische Verknüpfung (engl. *symbolic link*)
  - Abbildung eines Dateinamens auf einen Pfadnamen (S. 7-40)
- benannte Röhre (engl. *named pipe*)
  - Kommunikationskanal zwischen unverwandten lokalen Prozessen
- Buchse (engl. *socket*)
  - Endpunkt zur bi-direktionalen Kommunikation zwischen Prozessen

Dateiverzeichnis

Konzept zur Gruppierung von Dateinamen

Katalog (engl. *catalogue*, *directory*) von symbolischen Namen

- definiert einen gemeinsamen Kontext
  - symbolische Adressen sind nur innerhalb ihrer Kontexte eindeutig
- implementiert eine „Umsetzungstabelle“:

$$\left\{ \begin{array}{c} \text{symbolische Adresse} \\ \text{Dateiname} \\ \texttt{foo} \end{array} \right\} \rightsquigarrow \left\{ \begin{array}{c} \text{numerische Adresse} \\ \text{Dateikopf} \\ 4711 \end{array} \right\}$$

- speichert die Abbildung Dateiname $\mapsto$ Dateikopfnummer

Verknüpfung von Dateiname und Dateikopf

UNIX „*hard link*“ (auch kurz: *link*)

Eintrag im Dateiverzeichnis: Dateiname $\mapsto$ Dateikopfnummer

UNIX V7, `dir.h` [45]

```
typedef unsigned short ino_t;

#define DIRSIZ 14

struct direct {
    ino_t d_ino;
    char  d_name[DIRSIZ];
};
```

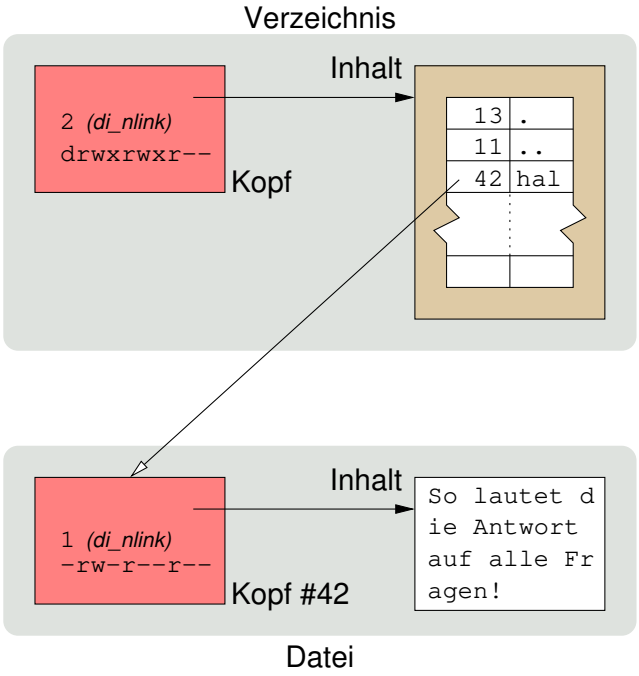
- die Abbildung ist als Wertepaar gespeichert
- mehrere Einträge können auf denselben Dateikopf verweisen
  - identische Dateikopfnummern
  - verschiedene Dateinamen
- ein Referenzzähler im Dateikopf vermerkt die Anzahl der Verweise

Anlegen/Löschen erfordert nur Schreibzugriffsrecht auf das Verzeichnis

- unabhängig von den Zugriffsrechten auf die referenzierte Datei

Verknüpfung von Dateiname und Dateikopf (Forts.)

Einträge anlegen/löschen ist eine Operation auf Verzeichnisse



Verzeichnisse sind „Spezialdateien“

- die selbst einen Namen und Dateikopf haben
- die erreichbar sind über eine Verknüpfung
  - eines anderen Verzeichnisses
- die Namen getrennt von Dateien speichern

Verknüpfungen anlegen/löschen zu können, ist eine Berechtigung, die sich nur auf das Verzeichnis der betreffenden Verknüpfungen bezieht!

Referenzzähler (engl. *reference count*)

☞ 7-43

Unterstützung der „Müllsammlung“ (engl. *garbage collection*)

Buchführung über die Anzahl der Verknüpfungen zu einem Dateikopf geschieht über einen Verknüpfungszähler (engl. *link count*; `di_nlink`)

`di_nlink != 0` Datei/Verzeichnis wird referenziert

- der Dateikopf ist (aus Sicht des Systems) in Benutzung

`di_nlink == 0` Datei/Verzeichnis wird nicht referenziert

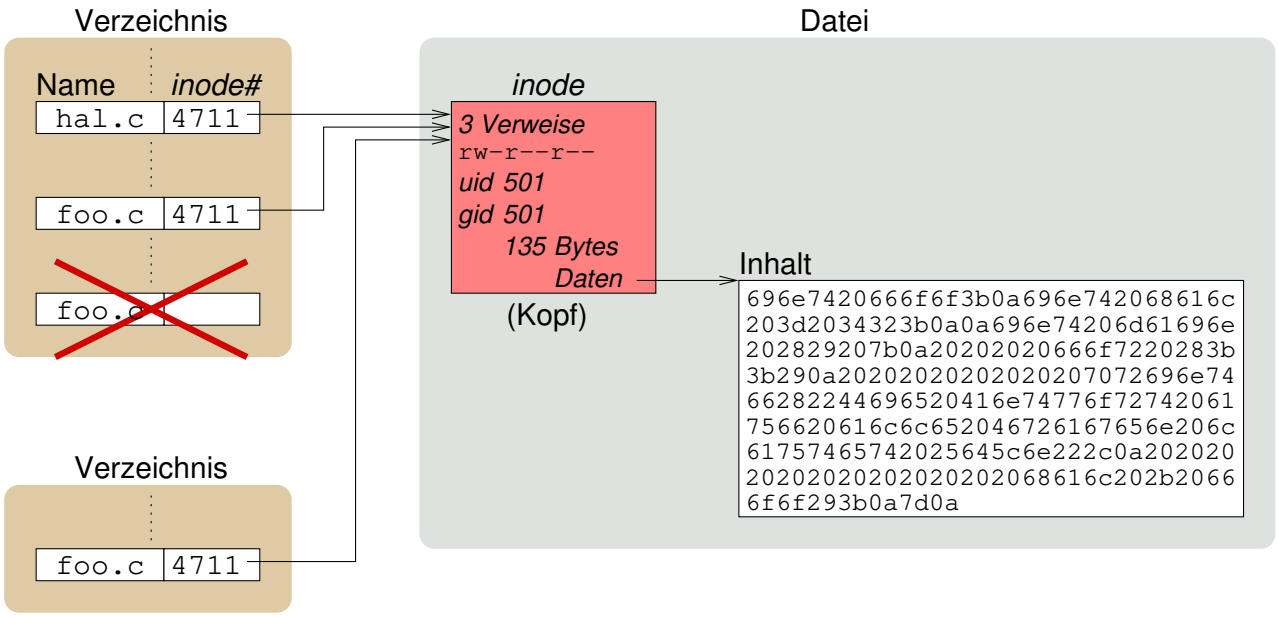
- der Dateikopf samt anhängender Daten kann freigegeben werden

Veränderung des Verknüpfungszählerwertes erfolgt beim Eintragen (++) bzw. Löschen (--) von Verknüpfungen im jeweiligen Verzeichnis:

- Verzeichnisverknüpfung: `mkdir(2)`/`rmdir(2)`
  - auf Verzeichnisse verweisen mindestens zwei Verknüpfungen (S. 7-42)
- Dateiverknüpfung: `link(2)`/`unlink(2)`

UNIX Dateiverzeichnis und Datei

Verknüpfung, Dateikopf und Dateiinhalt



UNIX Systemfunktionen

Operationen auf Dateiköpfe

Linux, MacOS, SunOS

```
fd  = open(path, fags, mode)
num = read(fd, buf, nbytes)
num = write(fd, buf, nbytes)
off = lseek(fd, offset, whence)
ok  = close(fd)
ok  = stat(path, buf)
    ⋮
```

(☞ Aufgabe 5)

Dateideskriptor (engl. *file descriptor*)

- von der Dateiverwaltung des Betriebssystems implementierter eindeutiger Bezeichner (engl. *identifier*) einer geöffneten Datei
- meist als Ganzzahl (engl. *integer*) repräsentiert

Bedeutung von Namen

Kontextfreie Namen sind bedeutungslos

Java bedeutet im Kontext...

- „Geographie“ eine Insel
- „Genussmittel“ ein Heissgetränk
- „Informatik“ eine Programmiersprache

C bedeutet im Kontext...

- „Sprache“ einen Buchstaben
- „Musik“ eine Note
- „Informatik“eine Programmiersprache

Namensräume (engl. *name spaces*) ordnen Namen Bedeutungen zu

Aufbau von Namensräumen

flache Struktur: definiert nur einen einzigen Kontext

- Eindeutigkeit muss mit der Namenswahl selbst gewährleistet werden

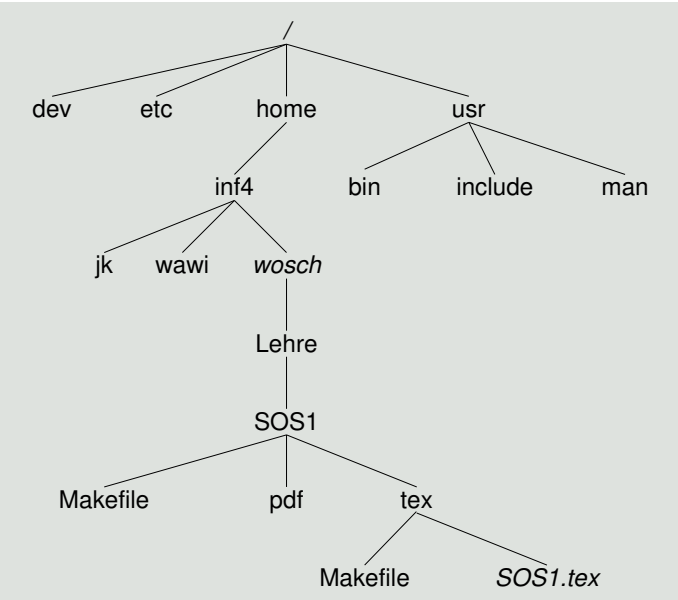
hierarchische Struktur: definiert mehrere Kontexte

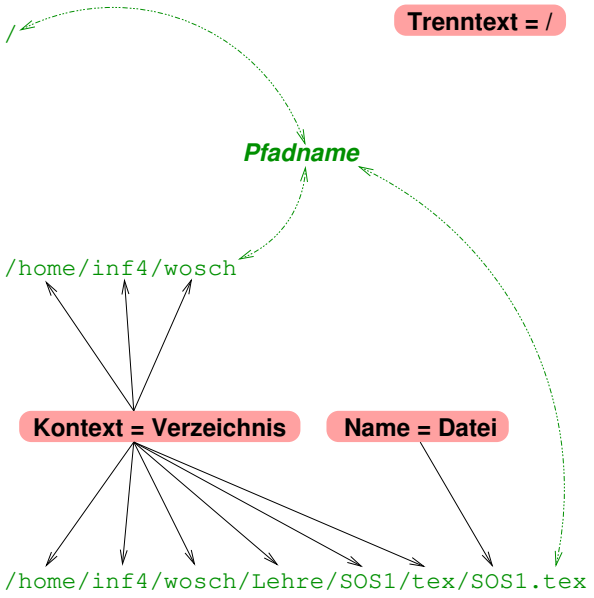
- Eindeutigkeit wird durch einen Kontextnamen als Präfix erreicht
  - Kontexte enthalten Namen von Dateien und/oder (anderer) Kontexte
  - der Name einer Datei entspricht dann einem „Blatt“ des Namensbaums
- als Separatoren werden meist Sonderzeichen („Trenntext“) verwendet:

Schrägstrich (*slash*)	⇒ UNIX
zurückgelehnter Schrägstrich (*backslash*)	⇒ Windows

Hierarchischer Namensraum

Dateibaum (engl. *file tree*)





Navigation im Namensraum

Eindeutigkeit der symbolischen Adresse (einer Datei) ist durch einen Pfad (engl. *path*) im Namensraum gegeben

- der Pfadname (engl. *path name*) ist ein vollständiger Dateiname

Formaler Aufbau eines (UNIX) Pfadnamens in EBNF [46]

$$
\begin{aligned}
\textit{pathname} &= \textit{resolver} \mid [\textit{resolver}], \{\textit{name}, \textit{resolver}\}, \textit{name};\\
\textit{resolver} &= \{\textit{separator}\}-;\\
\textit{separator} &= \text{“/“};\\
\textit{name} &= \{\textit{character}\}-;\\
\textit{character} &= \textit{character set} - \textit{separator};\\
\textit{character set} &= \text{ASCII};
\end{aligned}
$$

z.B.: `/`, `.`, `..`, `foo`, `foo/bar`, `/foo`, `bar/`, `./bar/..`, `../foo/.//bar//`

Spezielle Kontexte
Sonderverzeichnisse

Wurzelverzeichnis (engl. *root directory*)

- bezeichnet die Wurzel des Dateibaums (solitär '/', bei UNIX)
- wird vom System bzw. Administrator (engl. *super user*) gesetzt
  - `chroot(2)`, privilegierte Operation

Arbeitsverzeichnis (engl. *working directory*)

- die gegenwärtige Position eines Programms/Prozesses im Dateibaum
- ändert sich beim „Durchklettern“ des Dateibaums
  - `chdir(2)`

Heimatverzeichnis (engl. *home directory*)

- das initiale Arbeitsverzeichnis eines Benutzers/Prozesses
- wird vom System gesetzt bei Sitzungsbeginn
  - `login(1)`

Relative Adressierung von Kontexten
Systemdefinierte Verzeichnisnamen

. („*dot*“): das gültige Arbeitsverzeichnis (engl. *current working directory*)

- benennt die Verknüpfung zu selbigem Verzeichnis (Selbstreferenz)
  - ermöglicht die eindeutige Identifikation eines Arbeitsverzeichnisses, ohne dessen wirklichen Namen kennen zu müssen (`stat(2)`)
  - erzwingt einen lokalen Bezugspunkt (als Namenspräfix „`./`“)
- erster Eintrag in jedem Verzeichnis

.. („*dot dot*“): das gegenwärtige Elternverzeichnis

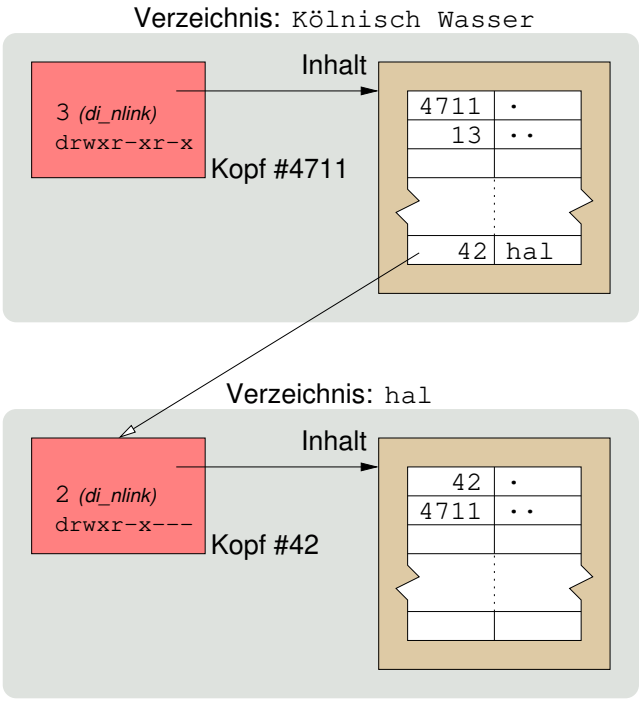
- benennt die Verknüpfung zum übergeordneten Verzeichnis, das die Verknüpfung zum Arbeitsverzeichnis enthält
  - entspricht '`.`', falls es kein Elternverzeichnis gibt (Wurzelverzeichnis)
- zweiter Eintrag in jedem Verzeichnis

☞ `mkdir(2)`

Relative Adressierung von Kontexten (Forts.)

Verknüpfungen für Arbeits- und Elternverzeichnisse

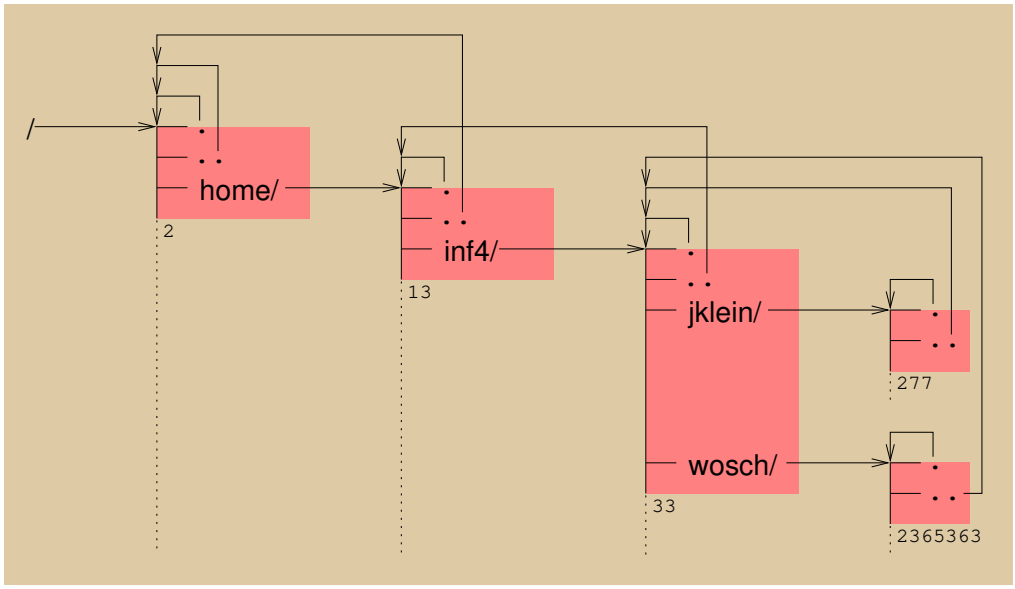
☞ 7-34



- `.` bezeichnet den Dateikopf, der das Verzeichnis selbst beschreibt
  - Dateikopfnummer des Arbeitsverzeichnisses
- `..` bezeichnet den Dateikopf des Verzeichnisses, das den Verzeichnisnamen speichert
  - Dateikopfnummer des Elternverzeichnisses

`di_nlink` # Verknüpfungen, die `.` referenzieren SunOS, Linux
speichert MacOSX

Dynamische Datenstruktur „Dateibaum“



Verzeichnisnamen entsprechen einer Vorwärtsverkettung (z.B. `wosch`)

`.` ist eine Selbstreferenz

`..` entspricht einer Rückwärtsverkettung

Arten von Pfadnamen

relativer Pfadname — vom gegenwärtigen Arbeitsverzeichnis ausgehend, z.B. von `/home/inf4/wosch` aus:

- `Lehre/SOS1/SOS1.tex`
- `./Lehre/SOS1/SOS1.tex`
- `../wosch/Lehre/SOS1/SOS1.tex`

...oder von `/home/inf4/jk` aus:

- `../wosch/Lehre/SOS1/SOS1.tex`

absoluter Pfadname — vom Wurzelverzeichnis ausgehend:

- `/home/inf4/wosch/Lehre/SOS1/SOS1.tex`

Bindung und Auflösung von Namen bzw. Pfadnamen

Abbildung/Umsetzung: symbolische Adresse $\mapsto$ numerische Adresse

Namensbindung (engl. *name binding*) bedeutet die Abbildung der symbolischen Adresse in eine numerische Adresse

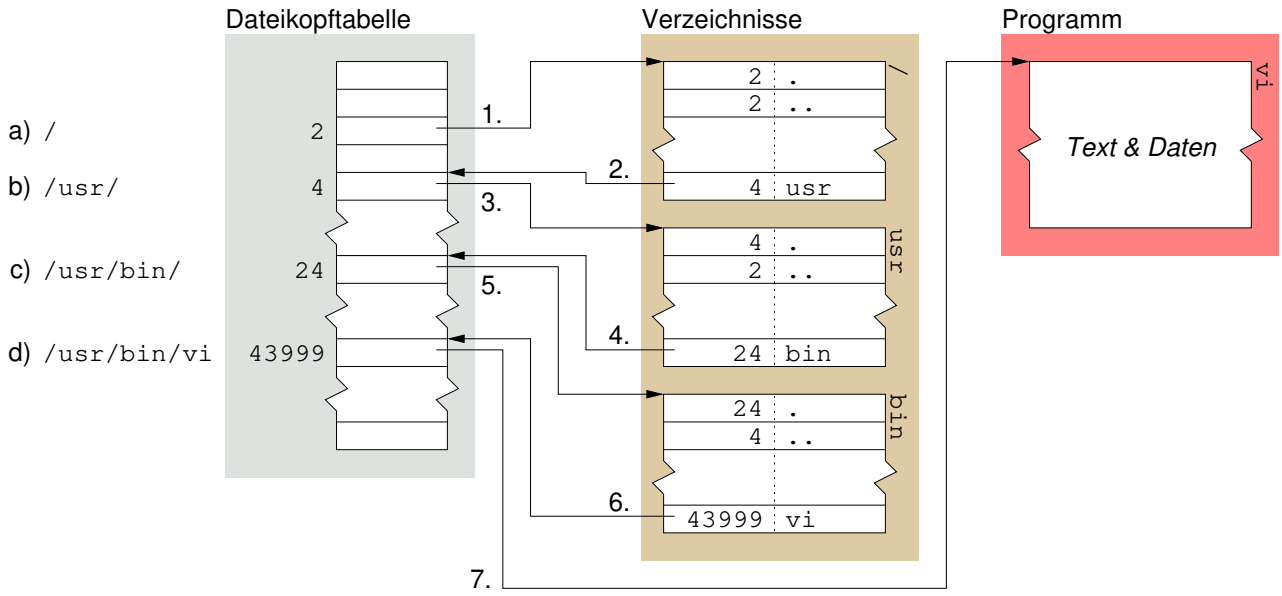
- zum Erzeugungszeitpunkt einen Datei-/Verzeichnisnamen...
  - mit einem freien/belegten Dateikopf verknüpfen
  - in ein Dateiverzeichnis eintragen
- Pfadnamen mit Dateikopf assoziieren: `creat(2)`, `link(2)`

Namensauflösung (engl. *name resolution*) bedeutet die Umsetzung der symbolischen Adresse in eine numerische Adresse

- zum Benutzungszeitpunkt Dateiverzeichnisse durchsuchen...
  - schrittweise für jeden einzelnen Verzeichnisnamen im Pfad
  - schließlich für den Dateinamen
- Dateikopf des Pfadnamens lokalisieren: `open(2)`

Auflösung von Namen bzw. Pfadnamen

Beispiel: `/usr/bin/vi`



Verwaltung von Dateien und Dateibäumen

Dateisystem (engl. *file system*)

Komplex von Datenstrukturen zur Verwaltung von Hintergrundspeicher

- der Dateisystemkopf (UNIX *super block*)
  - speichert Verwaltungsinformationen und Systemparameter
  - legt die Grenzwerte des Dateisystems fest
- die Dateikopftabelle (UNIX *inode table*)
  - zur Beschreibung von Dateien und/oder Verzeichnisse
- die Datenblöcke (engl. *data blocks*)
  - zur Speicherung der Inhalte der Dateien/Verzeichnisse

Beschreibung einer Partition (engl. *partition*) im Hintergrundspeicher

- die logische Unterteilung in einen Satz zusammenhängender Sektoren

Montieren von Dateisystemen

Auf- und Abbau einer Dateisystemhierarchie

Montierpunkt (engl. *mount point*) ist eine Stelle im Wirtsdateisystem, an der ein Gastdateisystem eingebunden werden kann

- ein (beliebiges) Verzeichnis im Wirtsdateisystem
- wird mit der Wurzel (S. 7-41) des Gastdateisystems überlagert

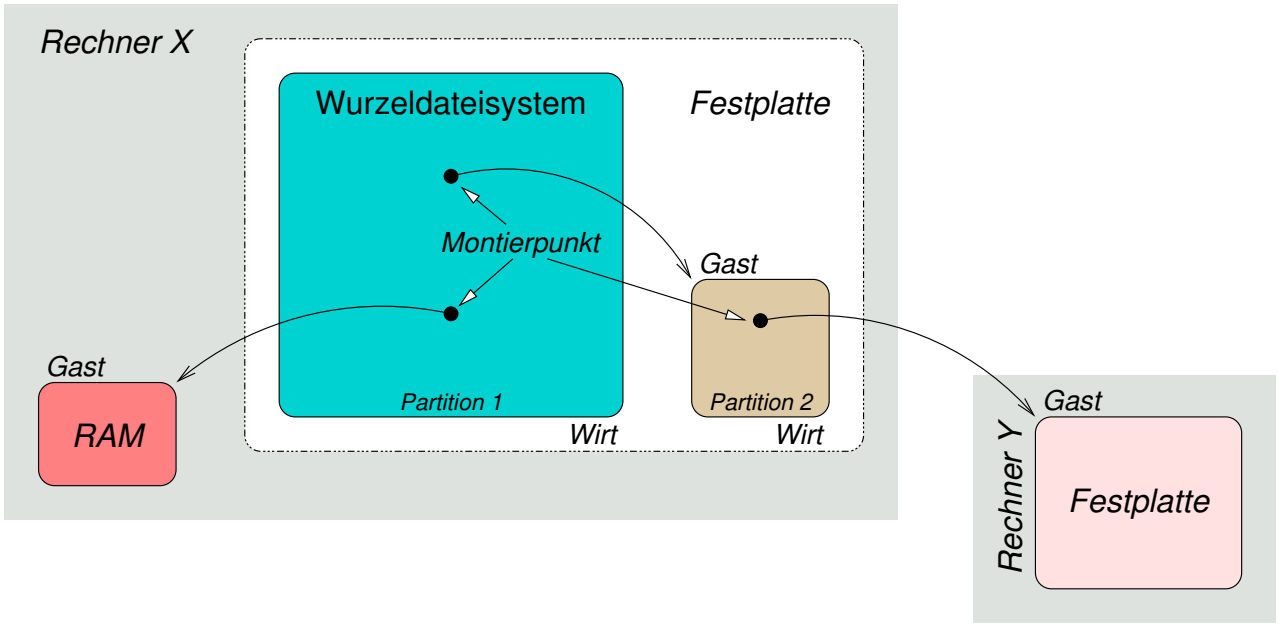
Wirts- und Gastdateisystem bilden jeweils eigene Partitionen...

- auf demselben oder einem anderen (dateisystemverträglichen) Gerät
  - z.B. Band, Fest-/Wechselplatte, CD, DVD, EEPROM, ..., RAM
  - ggf. auch auf unterschiedlichen Rechnern eines Rechnerverbunds
- von gleicher oder verschiedener (logischer) Struktur
  - ggf. ein Mix z.B. von S5FS, UFS, FFS, EXT2 und NTFS

Ausgangspunkt ist das Wurzeldateisystem (engl. *root file system*)

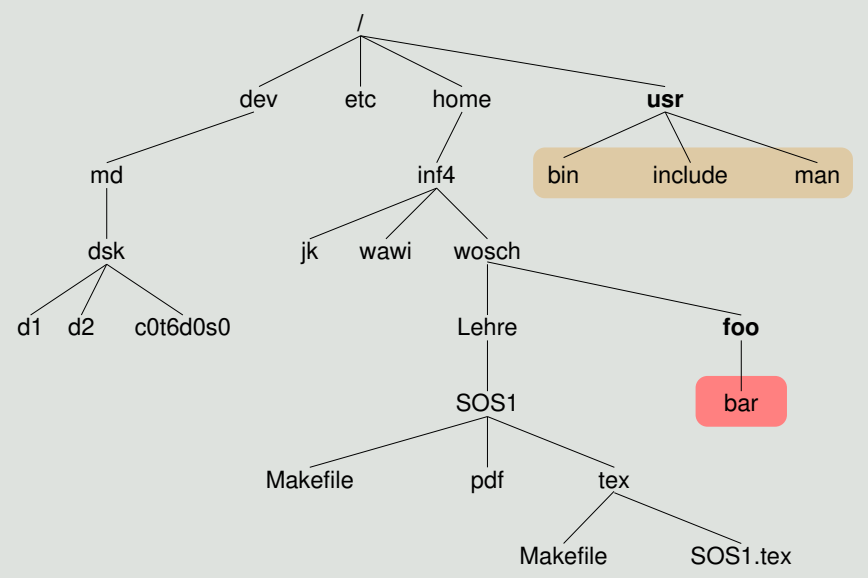
- d.h. das Dateisystem, von dem das Betriebssystem urgeladen wird

Hierarchiebildung von Dateisystemen



Einbindung von Namensräumen

Integration von Dateibäumen montierbarer Dateisysteme



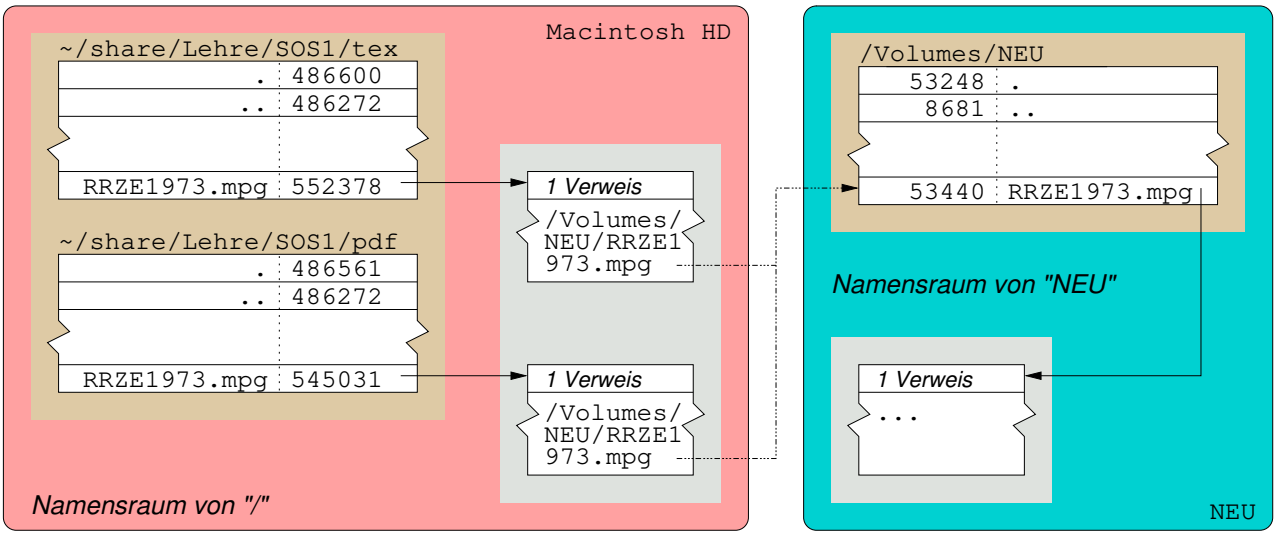
`/ -> /dev/md/dsk/d1`
Wurzel–/Wirtsdateisystem

`usr -> /dev/md/dsk/d2`
Gastdateisystem

`foo -> /dev/md/dsk/c0t6d0s0`
Gastdateisystem

Verweis hinein in einen anderen Namensraum

Symbolische Verknüpfung (engl. *symbolic link*)



Ursprung ist der symbolische Name (engl. *symbolic name*) in Multics [47]

- zur dynamischen Bindung von Namen an (besondere) E/A-Geräte

Symbolische Verknüpfungen „*Considered Harmful*“

Namen sind Schall und Rauch . . . auf den Inhalt kommt es an!

```
wosch@lorien 1$ mkdir -p Laptop/faui43w; cd Laptop
wosch@lorien 2$ ln -s faui43w lorien
wosch@lorien 3$ ls -l
total 8
drwxr-xr-x  2 wosch  wosch  68 29 Apr 13:01 faui43w
lrwxr-xr-x  1 wosch  wosch   7 29 Apr 13:02 lorien -> faui43w
wosch@lorien 4$ cd lorien
wosch@lorien 5$ cd ..; rmdir faui43w; cd lorien
-bash: cd: lorien: No such file or directory
wosch@lorien 6$ ls -l
total 8
lrwxr-xr-x  1 wosch  wosch   7 29 Apr 13:02 lorien -> faui43w
wosch@lorien 7$ mkdir faui43w; cd lorien
wosch@lorien 8$ ln -s fata\ morgana SOS1
wosch@lorien 9$
```


UNIX Systemfunktionen

Operationen auf Verzeichnisse

Linux, MacOS, SunOS

```
fd = creat(path, mode)
   ↦ open(path, O_CREAT | O_TRUNC | O_WRONLY, mode)
ok = link(path1, path2)
ok = symlink(path1, path2)
ok = unlink(path)
ok = mkdir(path, mode)
ok = rmdir(path)
ok = mount(type, dir, flags, data)
ok = umount(dir, flags)
   ⋮
```

UNIX Systemfunktionen

Operationen auf Verzeichnisse (Forts.)

Linux, MacOS, SunOS

```
dirp = opendir(path)
dp   = readdir(dirp)
loc  = telldir(dirp)
void   seekdir(dirp, loc)
void   rewinddir(dirp, loc)
ok   = closedir(dirp)
     ⋮
```

Vorsicht ist angebracht: `readdir(3)`'s Implementierung ist eintrittsvariant

- nebenläufige Ausführung kann Nebeneffekte zur Folge haben
- eintrittsinvariant (engl. *reentrant*) dagegen ist `readdir_r(3)`

Prozess . . . Programm in Ausführung

Kontrolliert durch Programme, exekuiert auf einen Prozessor

Prozess, kann die Ausführung mehrerer Programme bedeutet (S. 5-37)

- ein Anwendungsprogramm ruft ein Betriebssystemprogramm auf
  - Systemaufruf (engl. *system call*)
  - Programmunterbrechung (engl. *trap*, *interrupt*)
- ein Prozess ist Aktivitätsträger von ggf. mehreren Programmen
  - Adressraumüberlagerung mit einem anderem Programm (`exec(2)`)

Programm, kann von mehreren Prozessen ausgeführt werden

- nicht-sequentielles Programm, im Falle von Uniprozessorsystemen
  - präemptive (d.h. verdrängende) Programmverarbeitung
  - Aufgabe (engl. *task*), Faden (engl. *thread*)
- paralleles Programm im Falle von Multiprozessorsystemen

Prozess $\neq$ Programm

Programm ist statisch, Prozess ist dynamisch

Wissen über das gegenwärtig ausgeführte Programm sagt nicht viel aus über die zu dem Zeitpunkt im System stattfindende Aktivität.

- Welches Zugriffsrecht besitzt das Programm zur Zeit?
  - auf ein Adressraumsegment, auf eine Datei, auf ein Gerät, ...
  - allgemein: auf ein Betriebsmittel
- Welcher Kontrollfluss ist im mehrfädigen Programm zur Zeit aktiv?
  - Uni- vs. Multiprozessorsystem (SMP)
- Wieviel Programmunterbrechungen sind zur Zeit gestapelt?

⋮

Im Betriebssystemkontext ist das Konzept „Prozess“ daher nützlicher als das Konzept „Programm“, um Abläufe zu beschreiben und zu verwalten.

Prozess $\neq$ Prozessinstanz

Analogie zu Typ oder Klasse einerseits und Instanz bzw. Objekt andererseits

Prozess, ein abstraktes Gebilde

- ein „Programm in Ausführung“, sequentieller Kontrollfluss
- ein „Ablauf“, der eine Verwaltungseinheit ist

Prozessinstanz (auch: Prozessinkarnation), ein konkretes Gebilde

- die „physische Instanz“ des abstrakten Gebildes „Prozess“
  - an Betriebsmittel (Ressource; engl. *resource*) gebunden
  - die Identität (engl. *identity*) einer Programmausführung
- die Verwaltungseinheit, die einen Prozess beschreibt und repräsentiert
  - „dynamische Datenstruktur“ verschiedenartiger Strukturelemente

☞ synonyme Verwendung der Begriffe kann zu Missverständnissen führen

Prozessmodelle
Gewichtsklassen

schwergewichtiger Prozess (engl. *heavyweight process*)

- Prozessinstanz und Benutzeradressraum bilden eine Einheit
- Prozesswechsel $\rightsquigarrow$ zwei Adressraumwechsel: $AR_x \Rightarrow BS \Rightarrow AR_y$
  - „klassischer“ UNIX Prozess

leichtgewichtiger Prozess (engl. *lightweight process*)

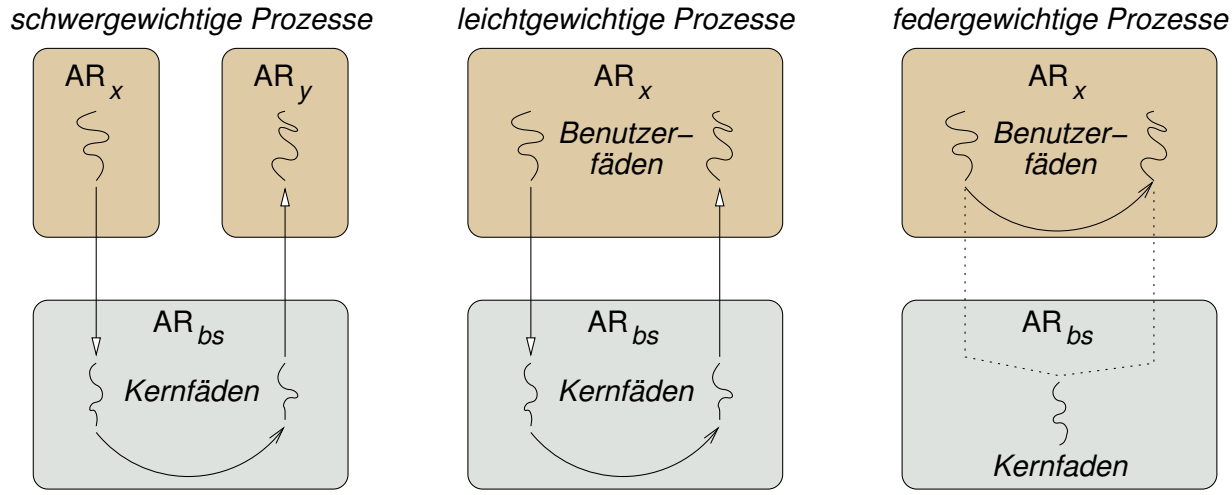
- Prozessinstanz und Adressraum sind voneinander entkoppelt
- Prozesswechsel $\rightsquigarrow$ einen Adressraumwechsel: $AR_x \Rightarrow BS \Rightarrow AR_x$
  - Kernfaden (engl. *kernel thread*): Faden auf Kernebene

federgewichtiger Prozess (engl. *featherweight process*)

- Prozessinstanzen und Adressraum bilden eine Einheit
- Prozesswechsel $\rightsquigarrow$ kein Adressraumwechsel: $AR_x \Rightarrow AR_x$
  - Benutzerfaden (engl. *user thread*): Faden auf Benutzerebene

Kern-/Benutzerfaden $\Rightarrow$ Betriebssystem-/Benutzerprogramm (S. 5-37)

Prozessmodelle (Forts.)
Schwer- vs. leicht- vs. federgewichtige Prozesse



Adressraumwechsel sind (je nach MMU) mehr oder weniger „teuer“

- die zur Adressumsetzung benötigten Deskriptoren werden mit jedem Wechselvorgang aus dem Zwischenspeicher (engl. *cache*) verdrängt
- erneute Adressraumaktivierung hat zur Folge, dass die MMU die Adressraumdeskriptoren erst wieder zwischenspeichern muss

Prozessbenutzthierarchie

Implementierung von Prozessen

schwergewichtiger Prozess

⇓

leichtgewichtiger Prozess

⇓

federgewichtiger Prozess

Basis ist der federgewichtige Prozess

- der eigentliche Kontrollfluss
- Steuerbefehle sind Prozeduren des auszuführenden Programms
  - erzeugen, wechseln, zerstören

Erweiterungen zum Mehrprogrammbetrieb bedeuten „Gewichtszunahme“

- leichtgewichtiger Prozess: vertikale Isolation vom Betriebssystem
  - Steuerbefehle sind Systemaufrufe an den Betriebssystemkern
- schwergewichtiger Prozess: horizontale Isolation von anderen Fäden
  - jeder Faden besitzt seinen eigenen (logischen/virtuellen) Adressraum

Implementierungskonzept von Prozess(instanz)en ist die Koroutine [48]

- in mehr oder weniger stark funktional angereicherter Form

Einplanung von Prozessen

Planung ihres zeitlichen Ablaufs (engl. *scheduling*)

Prozesseinplaung (engl. *process scheduling*) stellt sich allgemein zwei grundsätzlichen Fragestellungen:

1. Zu welchem Zeitpunkt sollen Prozesse ins System eingespeist werden?
2. In welcher Reihenfolge sollen Prozesse ablaufen?

Zuteilung von Betriebsmitteln an konkurrierende Prozesse kontrollieren

Einplanungsalgorithmus (engl. *scheduling algorithm*)

Implementiert die Strategie, nach der ein von einem Rechnersystem zu leistender Ablaufplan zur Erfüllung der jew. Anwendungsanforderungen entsprechend aufzustellen und zu aktualisieren ist.

Einplanung von Prozessen (Forts.)

Reihenfolge festlegen, Aufträge sortieren

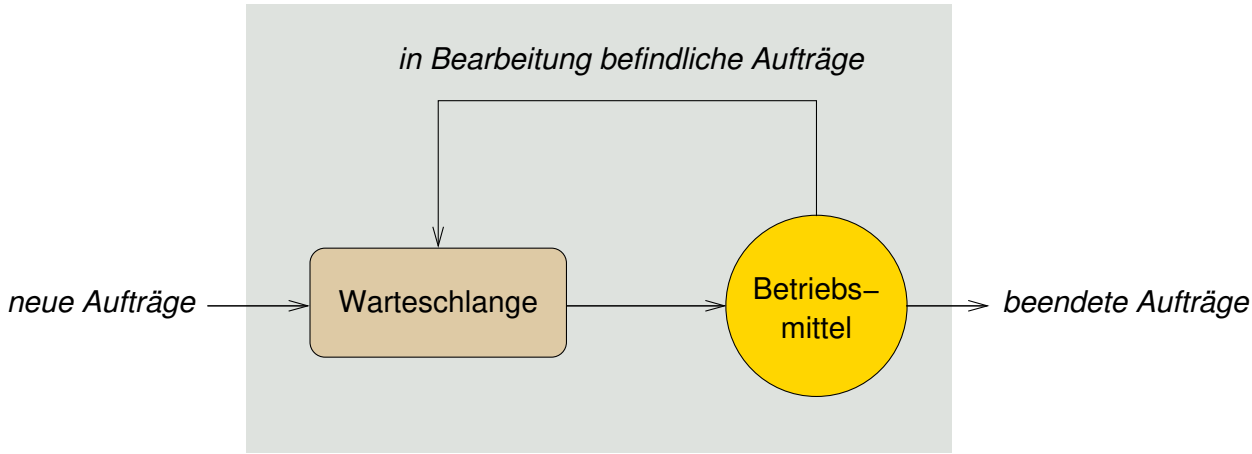
Ablaufplan (engl. *schedule*) zur Betriebsmittelzuteilung erstellen

- geordnet nach Ankunft, Zeit, Termin, Dringlichkeit, Gewicht, ...
- entsprechend der jeweiligen Einplanungsstrategie
- zur Unterstützung einer bestimmten Rechnerbetriebsart

Prozessen das Betriebsmittel { Prozessor, Speicher, Gerät, ⋮, Datei, Nachricht, Puffer } koordiniert zuteilen

Prinzipielle Funktionsweise von Einplanungsalgorithmen

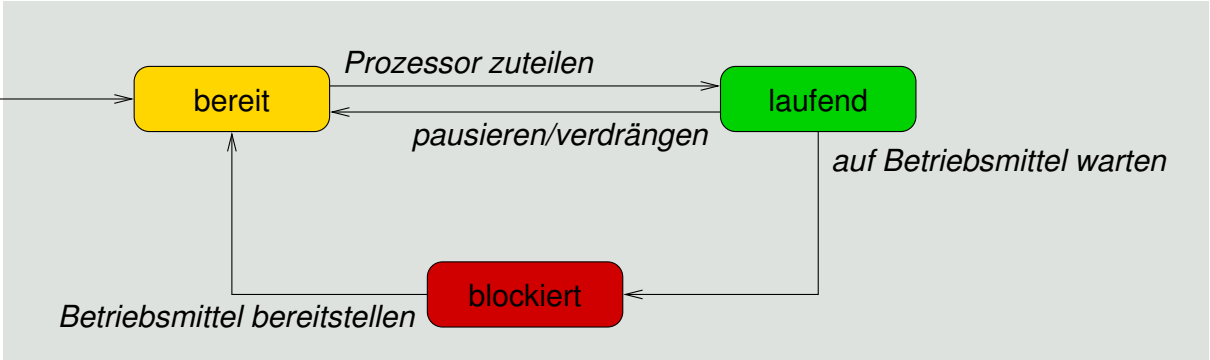
Verwaltung von (betriebsmittelgebundenen) Warteschlangen



Ein einzelner Einplanungsalgorithmus charakterisiert sich durch die Reihenfolge von Prozessen in der Warteschlange und die Bedingungen, unter denen die Prozesse in die Warteschlange eingereiht werden. [32]

Verarbeitungszustände von Prozessen

Zustandsübergänge implementiert ein Planer (engl. *scheduler*)



Prozessverarbeitung impliziert die Verwaltung mehrerer Warteschlangen:

- häufig ist jedem Betriebsmittel eine eigene Warteschlange zugeordnet
  - in der die Prozesse dann auf die Zuteilung dieses Betriebsmittels warten
- im Regelfall sind in Warteschlangen stehende Prozesse blockiert. . .
  - mit Ausnahme der Bereitliste (engl. *ready list*) der CPU
  - die auf Zuteilung der CPU wartenden Prozesse sind laufbereit

Warteschlangentheorie

Theoretische Grundlagen des Scheduling

Betriebssysteme durch die „theoretische/mathematische Brille“ gesehen:

- R. W. Conway, L. W. Maxwell, L. W. Millner. *Theory of Scheduling*.
- E. G. Coffman, P. J. Denning. *Operating System Theory*.
- L. Kleinrock. *Queuing Systems, Volume I: Theory*.

Einplanungsverfahren stehen und fallen mit den Vorgaben der Zieldomäne

- die „Eier-legende Wollmilchsau“ kann es nicht geben
- Kompromisslösungen sind geläufig, aber nicht in allen Fällen tragfähig

Scheduling ist ein Querschnittsbelang (engl. *cross-cutting concern*)

UNIX Scheduling

Charakteristische Eigenschaften — Ausnahmen bestätigen die Regel

Linux, MacOS, SunOS

- die Verfahren wirken verdrängend (engl. *preemptive*)
  - Prozesse können das Betriebsmittel „CPU“ nicht monopolisieren
  - dem laufenden Prozess kann die CPU entzogen werden (CPU-Schutz)
- der fortgeschriebene Ablaufplan ist nicht-deterministisch
  - nicht zu jedem Zeitpunkt ist bestimmt, wie weitergefahren wird
  - die exakte Vorhersage der Prozessorauslastung ist nicht möglich
- Prozessausführung und -einplanung sind gekoppelt (engl. *online*)
  - dynamische Prozesseinplanung während der Programmausführung
  - Planungsziel: Antwortzeiten minimieren, Interaktivität fördern
- das System arbeitet im Zeitmultiplexbetrieb (engl. *time sharing*)

UNIX Prozess

Schwergewicht: Prozess und Adressraum bilden eine Einheit

```
int foo;
int hal = 42;

int main () {
    for (;;)
        printf("Die Antwort auf alle Fragen lautet %d\n",
            hal + foo);
}
```

Wie ist der Adressraum bzw. Speicher des Prozesses organisiert, der die Ausführung dieses Programms bewirkt?

UNIX Prozess (Forts.)

Adressraumabbildung unter SunOS

```
wosch@faui40 40$ gcc -O6 -static -o hal hal.c; ./hal
Die Antwort auf alle Fragen lautet 42 ...^Z
wosch@faui40 41$ ps
   PID TTY      TIME CMD
 28426 pts/4    0:00 hal
   205 pts/4    0:00 ps
 25965 pts/4    0:00 tcsh-6.0
wosch@faui40 42$ pmap -x 28426
28426:  ./hal
 Address  Kbytes     RSS    Anon  Locked Mode   Mapped File
00010000     216     216       -       - r-x--  hal
00054000      16      16       8       - rwx--  hal
00058000       8       8       8       - rwx--    [ heap ]
FFBFE000       8       8       8       - rw---    [ stack ]
-------- ------- ------- ------- -------
total Kb     248     248      24       -
wosch@faui40 43$
```


UNIX Prozess (Forts.)

Pseudobefehle stecken Text-/Datenbereiche ab (gcc -S -O

```
    .file    "hal.c"
    .global  hal
    .section ".data"
    .align   4
    .type    hal,#object
    .size    hal,4
hal:
    .uaword  42
    .common  foo,4,4
    .section ".rodata"
    .align   8
.LLC0:
    .asciz   "Die Antwort auf alle Fragen lautet %d\n"
    .section ".text"
    .align   4
    .global  main
    .type    main,#function
    .proc    04
```

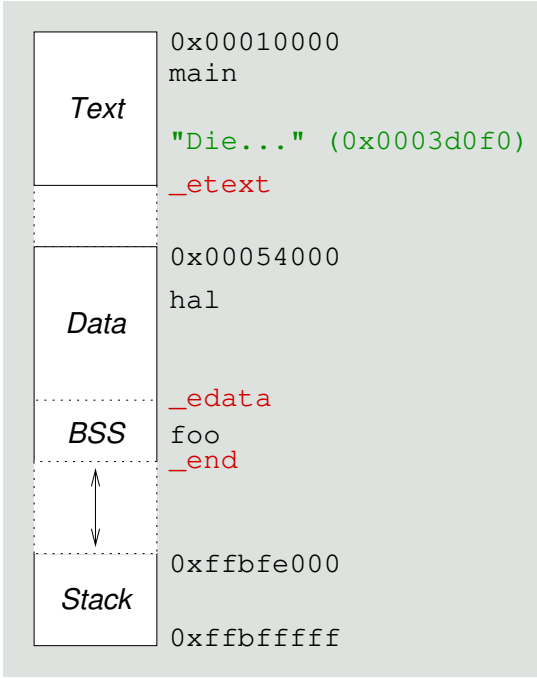
```
main:
    !#PROLOGUE# 0
    save     %sp, -112, %sp
    !#PROLOGUE# 1
    sethi    %hi(hal), %l2
    sethi    %hi(foo), %l1
    sethi    %hi(.LLC0), %l0
    ld       [%l2+%lo(hal)], %g1
.LL5:
    or       %l0, %lo(.LLC0), %o0
    ld       [%l1+%lo(foo)], %o3
    call     printf, 0
    add      %g1, %o3, %o1
    b        .LL5
    ld       [%l2+%lo(hal)], %g1
.LLfe1:
    .size    main,.LLfe1-main
    .ident   "GCC: (GNU) 3.0.4"
```

UNIX Prozess (Forts.)

Adressraumsegmente unter SunOS: Text, Daten, BSS, Stapel

```
wosch@faui40 43> nm -p -g hal
 ⋮
 0000066112 T main     ↦ 0x00010240
 0000352140 D hal      ↦ 0x00055f8c
 0000360336 B foo      ↦ 0x00057f90
 ⋮
 0000286461 D _etext   ↦ 0x00045efd
 0000358433 D _edata   ↦ 0x00057821
 0000361444 D _end     ↦ 0x000583e4
 ⋮
```

Nicht alle Übersetzer/Binder unter UNIX verwenden den Unterstrich ('_'), um die Symbole problemorientierter Programmiersprachen von Symbolen der Assembliersprachen unterscheiden zu können.



UNIX Prozess (Forts.)

Symbolische Adressen unterteilen den statischen Adressraum

Symbole, die vom Binder definiert und mit Werten belegt werden:

`extern etext`

- die erste Adresse nach dem Programmtext

`extern edata`

- die erste Adresse nach dem initialisierten Datenbereich

`extern end`

- die erste Adresse nach dem uninitialisierten Datenbereich
- entspricht anfangs der „Bruchstelle“ des Programms (☞ Aufgabe 4)
  - kann zur Ausführungszeit verschoben werden (`brk(2)`/`sbrk(2)`)
- `sbrk((intptr_t*)0)` liefert den aktuell gültigen Wert

BSS (engl. *block started by symbol*, [52, 53]) initialisiert der Lader mit 0

- der Binder legt nur die Größe fest: [`edata`, `end`[

UNIX Systemfunktionen

Operationen auf Prozesse und Prozessadressräume

Linux, MacOS, SunOS

```
pid = fork()
pid = wait(status)
void _exit(status)
pid = getpid()
pid = getppid()
ok  = nice(incr)
err = execv(path, argv)
err = execve(path, argv, envp)
    ⋮
```


Parthenogenese in UNIX

Aufspalten, abwarten und beenden

(☞ Aufgabe 3)

```
#include <sys/types.h>
#include <sys/wait.h>

char parent[] = "Elter: ", child[] = " Kind: ";

main() {
    pid_t pid;
    int zwerg;

    switch ((pid = fork())) {
        case -1:
            perror("fork");
            exit(1);
        case 0:
            printf("%sHier ist der Kindprozess, meine PID ist %d.\n", child, getpid());
            printf("%sDie PID meines Elterprozesses ist %d.\n", child, getppid());
            printf("%sGib mir einen (kleinen Wert als) Exitstatus: ", child);
            scanf("%d", &zwerg);
            printf("%sDanke und Tschüss!\n", child);
            exit(zwerg);
        default:
            printf("%sHier ist ein Elterprozess, meine PID ist %d.\n", parent, getpid());
            printf("%sDie PID meines Kindprozesses ist %d...\n", parent, pid);
            wait(&zwerg);
            printf("%sDer Exitstatus meines Kindprozesses ist %d.\n", parent, WEXITSTATUS(zwerg));
            printf("%sHollaröhdullıöh!\n", parent);
    }
}
```

Parthenogenese in UNIX (Forts.)

Ablaufprotokoll der Interaktion Elter ↔ Kind

```
wosch@gondor 71$ gcc -O6 -o fork fork.c
wosch@gondor 72$ ./fork
Elter: Hier ist ein Elterprozess, meine PID ist 1984.
Elter: Die PID meines Kindprozesses ist 1985...
 Kind: Hier ist der Kindprozess, meine PID ist 1985.
 Kind: Die PID meines Elterprozesses ist 1984.
 Kind: Gib mir einen (kleinen Wert als) Exitstatus: 42
 Kind: Danke und Tschüss!
Elter: Der Exitstatus meines Kindprozesses ist 42.
Elter: Hollaröhdullıöh!
wosch@gondor 73$
```


Koordination durch Kommunikation

Interprozesskommunikation (engl. *inter-process communication*, IPC)

Interaktion von Prozessen ist zwingend, um in einem Mehrprozesssystem Fortschritte in der Programmverarbeitung zu erreichen, und zwar:

implizit innerhalb des Betriebssystems

- nebenläufige Ausführung mehrfädiger Systemprogramme:
  - asynchrone Programmunterbrechungen
  - verdrängende Prozesseinplanung
  - ggf. auch SMP (engl. *symmetric multiprocessing*)
- die Prozesse konkurrieren um die Betriebsmittelzuteilung

explizit innerhalb des Anwendungssystems

- arbeitsteilige Ausführung eines Programms durch mehrere Prozesse
  - paralleles/verteiltes Programm
- die Prozesse kooperieren bei der gemeinsamen Programmausführung

Nebenläufigkeit „*Considered Harmful*“

Kritischer Abschnitt (engl. *critical section*)

Rückblick: nebenläufiges Zählen (S. 5-53)

- `wheel++` ist nicht immer eine unteilbare Operation
  - diese Operation der Ebene $_5$ ist nur „scheinbar elementar“
  - ggf. bildet sie eine Sequenz von Elementaroperationen der Ebene $_4$
  - in dem Fall wäre sie eine teilbare Operation und damit kritisch
- unterbrechungsbedingte Überlappungs(d)effekte können die Folge sein

Warteschlangen, z.B., stellen andere potentielle „Brennpunkte“ dar

- oft ist eine beliebige Permutation der Zugriffsoperationen möglich
  - eintragen überlappt austragen bzw. sich selbst, und umgekehrt
- auch die Auslegung der Datenstruktur „Schlange“ ist von Bedeutung

☞ „Untiefen“ dieser Art gibt es einige in Betriebssystemen...

Einreihung in eine einfach verkettete Liste

„Elementaroperation“ zum Anhängen eines Kettenglieds

```
typedef struct chainlink {
    struct chainlink *link;
} chainlink;

void chain (chainlink **next, chainlink *item) {
    *next = (*next)->link = item;
}
```

Die Funktion hängt ein neues Kettenglied hinter `*next` ein: `(*next)->link = item`. Der Zuweisungswert ist gleichfalls die Adresse des Kettenglieds, an dem das nächste Kettenglied angehängt werden soll. Diese Adresse wird vermerkt: `*next` = *Wert*. Damit ist `next` Einfügezeiger in eine einfach verkettete Liste.

`gcc -O6 -fomit-frame-pointer -S chain.c`. Auf der Assembliersprachenebene ist erkennbar, dass die Einfügeoperation als Folge von Einzelschritten auf der CPU zur Ausführung kommt. Im Falle der nicht-sequentiellen Ausführung ist nach jedem Einzelschritt mit einer asynchronen Programmunterbrechung zu rechnen, die eventuell die Verdrängung des laufenden Prozesses bewirkt und einen anderen Prozess startet, der dieselbe Funktion zeitlich überlappend (und damit nebenläufig) ausführen könnte.

x86

```
chain:
  movl 4(%esp), %ecx
  movl 8(%esp), %edx
  movl (%ecx), %eax
  movl %edx, (%eax)
  movl %edx, (%ecx)
  ret
```

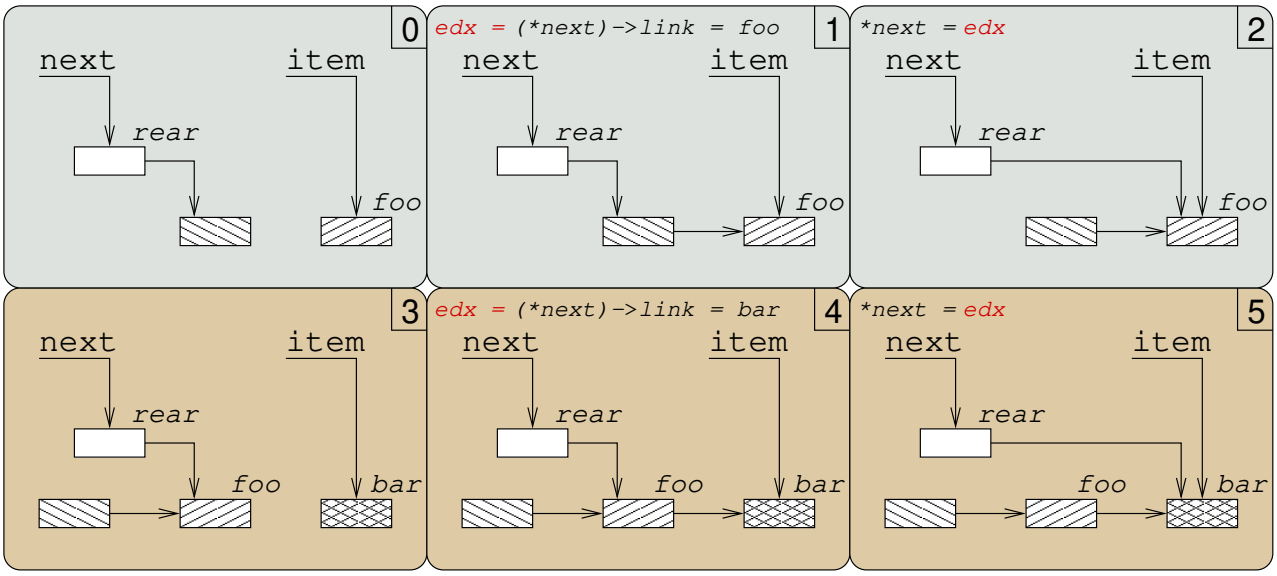
PPC

```
_chain:
  lwz r5,0(r3)
  stw r4,0(r5)
  stw r4,0(r3)
  blr
```

Einreihung in eine einfach verkettete Liste (Forts.)

Sequentielle Ausführung

```
chainlink *rear, *foo, *bar;
chain(&rear, foo);
```



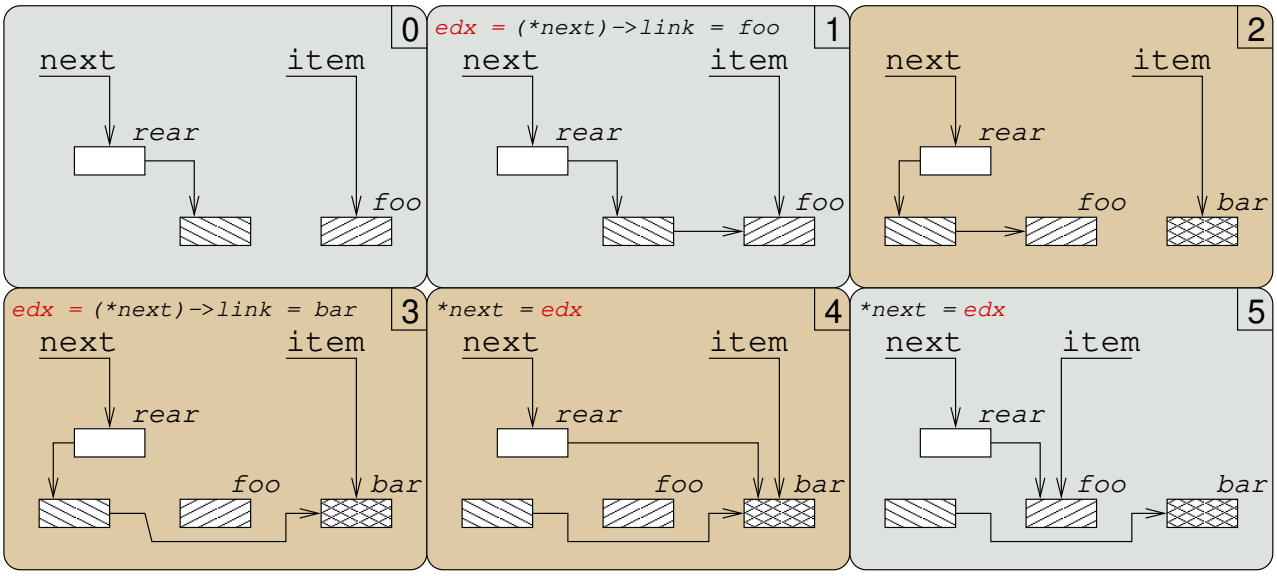

```
chain(&rear, bar);
```


Einreihung in eine einfach verkettete Liste (Forts.)

Nicht-sequentielle Ausführung

```
chainlink *rear, *foo, *bar;
chain(&rear, foo);
```

```
chain(&rear, bar);
```




```
chain(&rear, foo);
```

Koordinationsvariable

Semaphor (engl. *semaphore*)

Eine „nicht-negative ganze Zahl“, für die zwei unteilbare Operationen definiert sind [54]:

P (hol. *prolaag*, „erniedrige“; auch *down*, *wait*)

- hat der Semaphor den Wert 0, wird der laufende Prozess blockiert
- ansonsten wird der Semaphor um 1 dekrementiert

V (hol. *verhoog*, erhöhe; auch *up*, *signal*)

- inkrementiert den Semaphor um 1
- auf den Semaphor ggf. blockierte Prozesse werden deblockiert

Ein abstrakter Datentyp zur Signalisierung von Ereignissen zwischen gleichzeitigen Prozessen (deren Ausführung sich zeitlich überschneidet).

Koordination von Kooperation und Konkurrenz

Sequentialisierung nicht-sequentieller Programme

Synchronisation (engl. *synchronization*) bringt die Aktivitäten von verschiedenen Prozessen in eine Reihenfolge [55, S. 26]:

- dadurch wird prozessübergreifend das erreicht, wofür innerhalb eines Prozesses die Sequentialität von Aktivitäten sorgt
- Nebenläufigkeit bzw. Parallelität wird damit gezielt unterbunden

```
void chain (chainlink **next, chainlink *item) {
    P();
    *next = (*next)->link = item;
    V();
}
```

`P()` und `V()` klammern den kritischen Abschnitt: `P()` stellt sicher, das die Anweisungen bis zum `V()` nicht zugleich von mehreren Prozessen ausgeführt werden können.

UNIX Systemfunktionen

Operationen auf Semaphore

Linux, MacOS, SunOS

```
id  = semget(key, nsem, flag)
val = semctl(id, semnum, cmd, ...)
ok  = semop(id, sembuf, nops)
      ⋮
```

Sequenzen von Semaphoroperationen können unteilbar ausgeführt werden

- technisch ist die Sequenz als ein `sembuf`-Feld repräsentiert

```
struct sembuf {
    u_short sem_num;
    short   sem_op;
    short   sem_flg;
};
```

- jede `sembuf`-Instanz beschreibt eine (ggf. andere) auszuführende Operation
- die `sembuf`-Reihenfolge bestimmt die Ausführungsreihenfolge der Operationen

UNIX Semaphor

Einrichten und initialisieren einer Koordinationsvariablen

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int mutex;
struct sembuf sema;

int main () {
    mutex = semget(IPC_PRIVATE, 1, IPC_CREAT);
    if (mutex == -1) perror("semget");
    else if (semctl(mutex, 0, SETVAL, 1) == -1)
        perror("semctl");
    ...
}
```

UNIX Semaphor (Forts.)

Nachbildung von P und V

```
void P () {
    int ok;

    sema.sem_op = -1;
    ok = semop(mutex, &sema, 1);

    assert(ok != -1);
}
```

```
void V () {
    int ok;

    sema.sem_op = 1;
    ok = semop(mutex, &sema, 1);

    assert(ok != -1);
}
```

Scheitern von P/V sieht die klassische Definition [54] nicht vor:

- durch Zusicherung (engl. *assertion*) wird Gelingen „garantiert“
  - gelingt `semop()`, kehrt die aufrufende Funktion zurück
  - gelingt `semop()` nicht, wird das Programm abgebrochen
- Achtung: `-DNDEBUG` bzw. `#define NDEBUG` stellen Zusicherungen ab

Kommunikation durch Nachrichten

Motivation zum Botschaftenaustausch (engl. *message passing*)

Konsequenz der physikalischen Adressraumtrennung durch eine MMU:

- in Ausführung befindliche Programme sind abgeschottet
  - Prozesse sind in (log./virt.) Adressräumen eingesperrte „Gefangene“
  - sie können nicht ohne weiteres mit der „Außenwelt“ kommunizieren
- Kooperation muss Adressraumgrenzen überwinden können

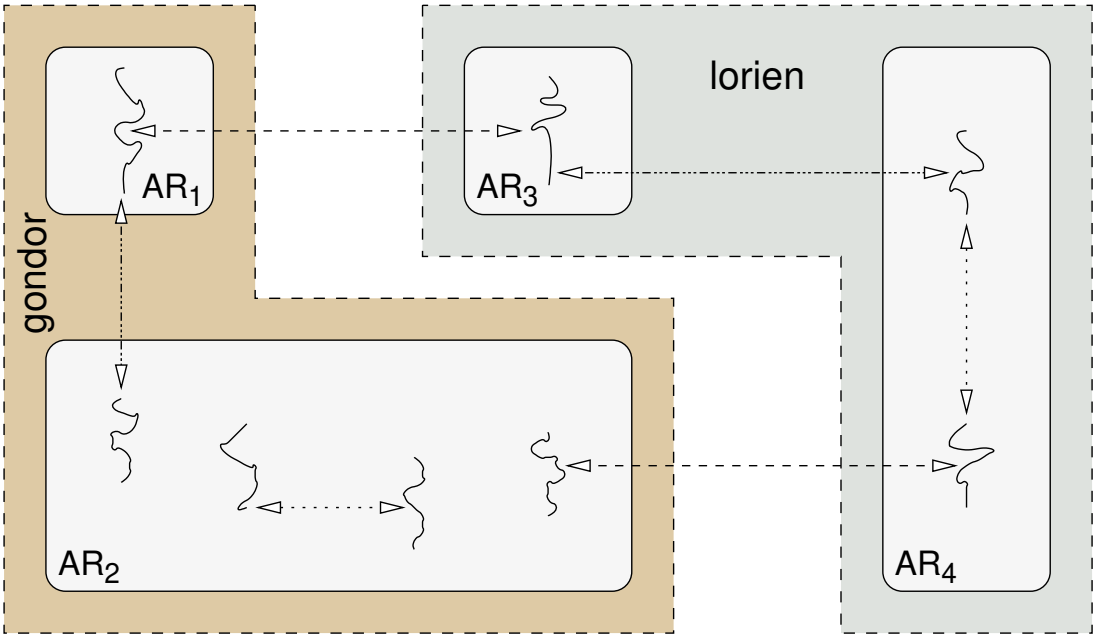
Konsequenz mehrerer Ausführungskontexte innerhalb eines Adressraums:

- Programme werden ggf. mehrfädig (engl. *multi-threaded*) ausgeführt
  - Fäden (engl. *threads*) sind eigene Kontrollflüsse im Programm
  - sie können nicht ohne weiteres mit anderen Fäden kommunizieren
- Kooperation muss Kontrollflussgrenzen überwinden können

Ein Semaphor eignet sich zur Anzeige des Ereignisses, dass Daten den einen Prozess verlassen haben und bei einem anderen Prozess eingetroffen sind. Den Datenaustausch selbst bewerkstelligt ein Semaphor nicht.

Problemdomänen der Kommunikation

Notwendigkeit domänenspezifischer Kommunikationsmechanismen



Botschaftenaustausch zwischen Prozessen

Prinzipielle Aktionen

Datentransfer vom Sende- zum Empfangsadressraum

- über einen den Prozessen gemeinsamen Kommunikationskanal

Synchronisation von Sende- und Empfangsprozess

- der Fortschritt des Empfangsprozesses hängt ab vom Sendeprozess
  - die Nachricht ist ein konsumierbares Betriebsmittel
  - der Empfangsprozess ist Konsument, der Sendeprozess ist Produzent
  - konsumiert werden kann nur, nachdem produziert worden ist
- der Fortschritt des Sendeprozesses hängt ab vom Empfangsprozess
  - der Nachrichtenpuffer ist ein wiederverwendbares Betriebsmittel
  - der Sendeprozess füllt, der Empfangsprozess leert den Puffer
  - gefüllt werden kann nur, wenn noch Platz ist ($\hookrightarrow$ leeren)
- die Koordination geschieht implizit mit der angewandten Primitive

Kommunikationssemantiken

Primitiven zum Botschaftenaustausch [56]

Sendeprimitiven wirken unterschiedlich auf den ausführenden Prozess, je nach Grad der Synchronisation mit dem Empfangsprozess:

no-wait send Sendeprozess wartet, bis die Nachricht im Transportsystem zum Absenden bereitgestellt worden ist

- Interprozesskommunikation im Vorübergehen (durch Pufferung)

synchronization send Sendeprozess wartet, bis die Nachricht vom Empfangsprozess angenommen worden ist

- Rendezvous zwischen Sender und Empfänger (ohne Pufferung)

remote-invocation send Sendeprozess wartet, bis die Nachricht vom Empfangsprozess verarbeitet und beantwortet worden ist

- Fernaufruf einer vom Empfangsprozess auszuführenden Funktion

Empfangsprimitiven wirken (im Regelfall) gleich auf den ausführenden Prozess: er wartet, bis eine Nachricht von einem Sendeprozess eintrifft

Kommunikationsmodelle

Rollenspiele bei der Interprozesskommunikation

Gleichberechtigte Kommunikation

Die miteinander kommunizierenden Prozesse spielen dieselbe Rolle; zwei Kommunikationspartner, P_1/P_2, sind sowohl Sender als auch Empfänger:

$$P_1 \left\{ \begin{array}{ccc} send & \longrightarrow & receive \\ receive & \longleftarrow & send \end{array} \right\} P_2$$

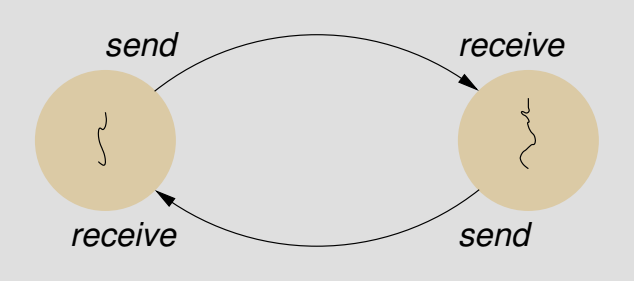
Ungleichberechtigte Kommunikation

Die miteinander kommunizierenden Prozesse spielen verschiedene Rollen; ein Kommunikationspartner, P_2, ist Dienstgeber (engl. *server*), der andere, P_1, ist Dienstnehmer (engl. *client*):

$$\text{(Klient) } P_1 \left\{ \begin{array}{ccc} send & \longrightarrow & receive \\ & \longleftarrow & reply \end{array} \right\} P_2 \text{ (Anbieter)}$$

Gleichberechtigte Kommunikation

no-wait send oder *synchronization send*



Die an der Kommunikation beteiligten Prozesse sind in ihrer Rollenfunktion gleichzeitig Produzent und Konsument von Nachrichten.

send Bereitstellung eines konsumierbaren Betriebsmittels
- *in* Identifikation des Empfängers (Konsument)
- *in* Basis/Länge der Nachricht

receive Anforderung eines konsumierbaren Betriebsmittels
- *in* Basis/Länge eines Empfangspuffers
- *out* Identifikation des Senders (Produzent)

Gleichberechtigte Kommunikation (Forts.)

Gegenseitige Abhängigkeit der Botschaften austauschenden Prozesse

```
void ibm () {
    char buf[SBUFSZ];
    pid_t from, peer = lookup("hal");

    ipc_send(peer, "?", 1);
    from = ipc_receive(buf, sizeof(buf));
    if (from == peer) printf("%s\n", buf);
}
```

```
void hal () {
    char buf[RBUFSZ];
    pid_t peer;

    peer = ipc_receive(buf, sizeof(buf));
    if (peer && (buf[0] == '?'))
        ipc_send(peer, "42", 3);
}
```

Sender „`ibm`“ muss die Adresse des Empfängers ermitteln, um ihm eine Nachricht zustellen zu können:

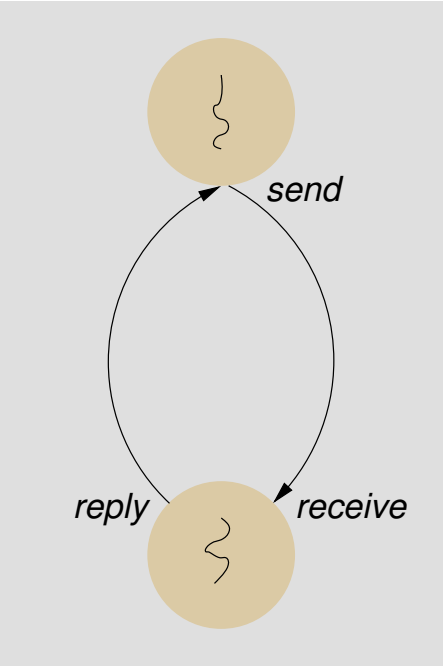
- Namensdienst (engl. *name service*)
- `lookup()` liefert die PID von „`hal`“

Empfänger „`hal`“ erhält die Adresse des Senders implizit beim Empfang der Nachricht

Ungleichberechtigte Kommunikation

remote-invocation send

Die an der Kommunikation beteiligten Prozesse besitzen unterschiedliche Rollenfunktionen, Klient einerseits und Anbieter andererseits:



send einer Anforderungsnachricht
- *in* Identifikation des Anbieters
- *in* Basis/Länge der Nachricht
- *in* Basis/Länge eines Empfangspuffers
- *out* Identifikation eines Anbieters

receive einer Anforderungsnachricht
- *in* Basis/Länge eines Empfangspuffers
- *out* Identifikation des Klienten

reply einer Antwortnachricht
- *in* Identifikation des Klienten
- *in* Basis/Länge der Nachricht

Ungleichberechtigte Kommunikation (Forts.)

Anbieter sind unabhängig von der Empfangsbereitschaft der Klienten

```
void ibm () {
    char buf[SBUFSZ];
    pid_t peer = lookup("hal");

    if (send(peer, "?", 1, buf, sizeof(buf)) == peer)
        printf("%s\n", buf);
}
```

```
void hal () {
    char buf[RBUFSZ];
    pid_t peer;

    peer = receive(buf, sizeof(buf));
    if (peer && (buf[0] == '?'))
        reply(peer, "42", 3);
}
```

Klient „`ibm`“ ruft Dienst ab und erwartet Antwort (`send()`)

Anbieter „`hal`“ erbringt Dienst (`receive()`) und gibt Antwort (`reply()`)

Senke der Interprozesskommunikation

Adressierung des Kommunikationspartners — direkt vs. indirekt

Faden (engl. *thread*) Konsument der Nachricht

- direkte Adresse der die Nachricht verarbeitenden Instanz
- die Prozessidentifikation (PID)

Tor (engl. *port*) Anschluss zur Weiterleitung/Zustellung von Nachrichten, der einem bestimmten Prozess zugeordnet ist

- Prozesse können mehrere solcher Anschlüsse besitzen
  - Ein- und/oder Ausgangstore für Nachrichten
- die Zuordnung ist statisch oder dynamisch

Briefkasten (engl. *mailbox*) Zwischenspeicher für Nachrichten, der durch Senden gefüllt und Empfangen geleert wird

- der Pufferbereich ist keinem Prozess zugeordnet
- N Prozesse können dahin senden und daraus empfangen

Kommunikation und Betriebsmittel

Synchrone vs. asynchrone Interprozesskommunikation

Prozesse synchronisieren sich zur Kommunikation, indem sie Betriebsmittel anfordern und bereitstellen (S. 7-88):

Sender benötigt das wiederverwendbare Betriebsmittel „Puffer“

synchrone IPC	$\Rightarrow$	der Zielpuffer (des Empfängers)
asynchrone IPC	$\Rightarrow$	ein Zwischenpuffer

Empfänger benötigt das konsumierbare Betriebsmittel „Nachricht“

asynchrone IPC	$\Rightarrow$	ein Zwischenpuffer
synchrone IPC	$\Rightarrow$	der Quellpuffer (des Senders)

Betriebsmittelmangel ist die Ursache dafür, dass Prozesse bei der Kommunikation ggf. blockieren werden:

- Empfänger erwartet Nachricht, Sender erwartet freien Puffer
- „asynchron“ bedeutet nicht „nicht-blockierend“ oder „wartefrei“

Verbindungen zwischen kommunizierenden Prozessen

Gütemerkmale (engl. *quality of service*) garantieren

IPC verwendet (in dem Fall) Torverbindungen und verläuft in drei Phasen:

Aufbauphase plant die zur Durchsetzung der jeweils angeforderten Gütemerkmale notwendigen Betriebsmittel ein
- Puffer, Fäden, Bandbreite, ..., Protokoll

Nutzungsphase Botschaftenaustausch gemäß Gütemerkmale

Abbauphase gibt reservierte Betriebsmittel frei, löst die Verbindung auf

Betriebsarten der durch Torverbindungen entstehenden Verbindungskanäle bestimmen u.a. die möglichen Richtungen des Datentransfers:

unidirektional	Tor_s	$\longrightarrow$	Tor_r	halbduplex
bidirektional	Tor_{sr}	$\longleftrightarrow$	Tor_{rs}	vollduplex

UNIX Systemfunktionen

Linux, MacOS, SunOS

```
s   = socket(domain, type, protocol)
ok  = bind(s, name, namelen)
num = recvfrom(s, buf, buflen, flags, from, fromlen)
num = sendto(s, msg, msglen, flags, to, tolen)
ok  = connect(s, name, namelen)
ok  = listen(s, backlog)
d   = accept(s, addr, addrlen)
num = recv(d, buf, buflen, flags)
num = send(s, msg, msglen, flags)
ptr = gethostbyname(name)
    ⋮
```

Botschaftenaustausch mit UNIX Systemfunktionen

Sendeprimitive (passend zu S. 7-92) — mit Bedacht zu verwenden

- jede Aktivierung fordert *Socket* und Puffer an: Betriebsmittel sind nicht garantiert

```
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/un.h>
#include "./message.h"
#include "./pid2rid.h"

int ipc_send (pid_t peer, char *data, unsigned size) {
    int ok, s, nbytes;
    struct sockaddr_un addr;
    message *msg;
    char name[RIDSZ];

    if ((ok = s = socket(AF_UNIX, SOCK_DGRAM, 0)) != -1) {
        if ((ok = (msg = (message*)malloc(nbytes = sizeof(message) + size)) != 0)) {
            msg->ipc_from = getpid(); bcopy(data, msg->ipc_data, size);

            addr.sun_len = 0; addr.sun_family = AF_UNIX;
            bcopy(pid2rid(peer, name), addr.sun_path, RIDSZ);

            ok = sendto(s, msg, nbytes, 0, (struct sockaddr *)&addr, sizeof(addr));
            free(msg);
        }
        close(s);
    }
    return ok;
}
```


Botschaftenaustausch mit UNIX Systemfunktionen (Forts.)

Empfangsprimitive (passend zu S. 7-92) — mit Bedacht zu verwenden

- jede Aktivierung fordert *Socket* und Puffer an: Betriebsmittel sind nicht garantiert
- nur für die Dauer von `recvfrom(2)`ist die (lokale) PID des Empfängers an einen (globalen) Namen gebunden: Sendeversuche können scheitern

```
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/un.h>
#include "./message.h"
#include "./pid2rid.h"

pid_t ipc_receive (char *data, unsigned size) {
    int ok, s, nbytes, addrlen;
    message *msg;
    char name[RIDSZ];
    struct sockaddr_un addr;

    if ((ok = s = socket(AF_UNIX, SOCK_DGRAM, 0)) != -1) {
        if ((ok = (msg = (message*)malloc(nbytes = (sizeof(message) + size))) != 0)) {
            addr.sun_len = 0; addr.sun_family = AF_UNIX;
            bcopy(pid2rid(getpid(), name), addr.sun_path, RIDSZ);

            if ((ok = bind(s, (struct sockaddr *)&addr, sizeof(addr))) != -1) {
                addrlen = sizeof(addr);
                if ((ok = recvfrom(s, msg, nbytes, 0, (struct sockaddr *)&addr, &addrlen)) != -1) {
                    ok = msg->ipc_from; bcopy(msg->ipc_data, data, size);
                }
                unlink(name);
            }
            free(msg);
        }
        close(s);
    }
    return ok;
}
```

Botschaftenaustausch mit UNIX Systemfunktionen (Forts.)

Protokolldateneinheit (engl. *protocol data unit*, PDU) und Namensfunktion

Nachricht als Dienstdateneinheit (engl. *service data unit*, SDU), die in einer PDU vom Sender zum Empfänger transportiert wird:

```
typedef struct ipc_message {
    pid_t ipc_from;
    char  ipc_data[0];
} message;
```

Schablone mit der PID des Senders und einem generischen Feld zur Aufnahme der SDU.

Kommunikationsendpunkt ist eine Prozessinstanz, aus deren PID eine symbolische Adresse (*receiver ID*, RID) generiert wird:

```
#define RIDSZ (int)((sizeof(pid_t) * 2.5) + 3)

inline char* pid2rid (pid_t pid, char rid[]) {
    sprintf(rid, "rid%u", pid);
    return rid;
}
```

Die RID wird im UNIX Namensraum abgelegt (`bind(2)`).

UNIX Kommunikationsfunktionen „*Considered Harmful*“

Typisch „Allgemeinzweck“ — und dennoch problematisch für IPC

Pros

- Datenstromkonzept
  - *no-wait send* Semantik, kanalorientiert
- in E/A-Schnittstelle integriert
  - `read(2)` empfängt, `write(2)` sendet
  - `close(2)` gibt *Socket* frei, `unlink(2)` entfernt *Socket*-Verknüpfung
- verschiedenste Protokolle und Betriebsarten

Cons

- rein asynchrones Modell
  - *synchronization/remote-invocation send* fehlt
- sehr komplexe Schnittstelle
  - knifflige, kostspielige problemorientierte Lösungen auf Bibliotheksebene
  - synchrone IPC nur sehr aufwändig nachbildbar
- man muss mit Kanonen auf Spatzen schießen...

☞ ganz gut, um schwergewichtige Klient/Anbieter-Systeme zu bauen...

UNIX Dienstgeber

Anbieter (engl. *server*)

Betriebsmittel für die Lebendsauer des Dienstgebers, um ein Scheitern des Sendens bei hoher Anfragefrequenz zu vermeiden:

- Anbieter-*Socket* (`socket(2)`),
- *Socket*-Verknüpfung (`bind(2)`)

(vergl. S. 7-100).

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

void fail (char* msg) { perror(msg); exit(0); }

int main (int argc, char *argv[]) {
    struct sockaddr_in addr;
    int sockfd, fd, addrlen = sizeof(addr);
    char buffer[256];

    if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0) fail("socket");

    bzero((char *)&addr, sizeof(addr));
    addr.sin_family = AF_INET;
    addr.sin_addr.s_addr = INADDR_ANY;
    addr.sin_port = htons(atoi(argv[1]));
    if (bind(sockfd, (struct sockaddr *)&addr, sizeof(addr)) < 0) fail("bind");

    if (listen(sockfd, 5) < 0) fail("listen");
    for (;;) {
        bzero(buffer, 256);
        if ((fd = accept(sockfd, (struct sockaddr *)&addr, &addrlen)) < 0) fail("accept");
        if (read(fd, buffer, 256) < 0) fail("read");
        close(fd);
        printf("%s\n", buffer);
    }
}
```


UNIX Dienstnehmer

Klient (engl. *client*)

Klient/Anbieter-Systeme sind mit einem mehr auf IPC und weniger auf Datenstromkommunikation ausgerichteten Konzept besser bedient:

- *remote-invocation send* ist ein Muss [57],
- ein Mikrokern [58, 59] ist vorteilhaft

(vergl. S. 7-94).

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>

void fail (char *msg) { perror(msg); exit(0); }

int main(int argc, char *argv[]) {
    int sockfd;
    struct sockaddr_in addr;
    struct hostent *server;

    if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0) fail("socket");

    bzero((char *)&addr, sizeof(addr));
    addr.sin_family = AF_INET;
    if ((server = gethostbyname(argv[2])))
        bcopy((char *)server->h_addr, (char *)&addr.sin_addr.s_addr, server->h_length);
    addr.sin_port = htons(atoi(argv[1]));
    if (connect(sockfd, (struct sockaddr*)&addr, sizeof(addr)) < 0) fail("connect");

    if (write(sockfd, argv[3], strlen(argv[3])) < 0) fail("write");
}
```

UNIX Dienstgeber/-nehmer

Int{ra,er}rechner-Interprozesskommunikation

Anbieter: „printf()-Server“

```
wosch@gondor 85$ gcc -O6 server.c -o server
wosch@gondor 86$ ./server 4711
Die Anwort auf alle Fragen ist
42
```

Klient 1

```
wosch@gondor 11$ gcc -O6 client.c -o client
wosch@gondor 12$ ./client 4711 gondor 42
wosch@gondor 13$
```

Klient 2

```
wosch@lorien 62$ gcc -O6 client.c -o client
wosch@lorien 63$ ./client 4711 gondor “Die Antwort auf alle Fragen ist“
wosch@lorien 64$
```


Resümee eines Betriebssystemüberblicks

Grundlegende Funktionen im Schnelldurchlauf

- Betriebssysteme bieten eine „Hand voll“ nützlicher Abstraktionen
  - Adressräume, Speicher, Dateien, Namensräume
  - Prozesse, Koordinationsmittel
- von zentraler Bedeutung ist die Abbildung von Namen auf Adressen
  - symbolische $\mapsto$ numerische $\mapsto$ logische $\mapsto$ virtuelle $\mapsto$ physikalische
  - das Konzept findet im Kontext von Rechnernetzen seine Fortführung
- nicht weniger bedeutend ist die Einplanung von Prozessen
  - d.h., die Planung des zeitlichen Ablaufs der Prozessorzuteilung
  - allgemein: die Planung der Zuteilung von Betriebsmitteln an Prozesse
- jeder Prozess gehört einer bestimmten Gewichtsklasse an
  - schwer-, leicht- oder federgewichtiger Prozess
  - u.a. eine Frage der Verzahnung von Prozessinstanz und Adressraum
- Mehrprozessbetrieb erfordert die Koordination von Prozessen