# Betriebssystemtechnik

Softwaretechnik: Hierarchien

Wolfgang Schröder-Preikschat

Lehrstuhl Informatik 4

10. Juli 2012



## Gliederung

1. Einleitung
   - Hierarchische Struktur
2. Arten von Hierarchie
   - Programmhierarchie
   - Prozesshierarchie
   - Mittelvergabehierarchie
   - Schutzhierarchie
3. Funktionale Hierarchie
   - Benutzthierarchie
   - Hierarchiebildung
   - Fallbeispiel
4. Zusammenfassung





## Motiv: Beherrschung der Systemkomplexität

Komplexität (nicht nur) von Betriebssystemen in den Griff zu bekommen, setzt adäquate, hierarchisch organisierte Softwarestrukturen voraus:

- **Programmhierarchie** definiert verschiedene, problemspezifische Ebenen der Abstraktion und fördert den Aufbau einer *Familie von Systemen*
- **Prozesshierarchie** macht ein System relativ unempfänglich bzgl. Anzahl der verfügbaren Prozessoren und ihren relativen Geschwindigkeiten
- **Mittelvergabehierarchie** organisiert ein System in problemspezifische Betriebsmittelzuteiler und -verwalter
- **Schutzhierarchie** verbessert wesentlich die Vertrauswürdigkeit einzelner Systembestandteile und erhöht die Sicherheit des Gesamtsystems

**Lernziel**

- die für Betriebssysteme wichtigen Arten von Hierarchie erfassen





## Definition „hierarchische Struktur“ [14]

„Struktur“ bezieht sich auf die partielle Beschreibung eines Systems und drückt sich aus bzw. stellt das System dar durch:

1. eine Sammlung einzelner Systembestandteile und
2. eine Beziehung zwischen diesen Bestandteilen

Strukturen sind „hierarchisch“, wenn eine Relation $R(\alpha, \beta)$ zwischen Teilepaaren Ebenen wie folgt entstehen lässt:

1. Ebene $_0$ ist Menge von Teilen $\alpha$, so dass es kein $\beta$ gibt mit $R(\alpha, \beta)$
2. Ebene $_{i, i>0}$ ist Menge von Teilen $\alpha$, so dass gilt:
   1. es existiert ein $\beta$ auf Ebene $_{i-1}$ mit $R(\alpha, \beta)$ und
   2. falls $R(\alpha, \gamma)$, dann liegt $\gamma$ auf Ebene $_{i-1}$ oder niedriger

- Relation $R$ repräsentiert als gerichteter azyklischer Graph

## Grundsätzliches

Aussagen der Art „unser Betriebssystem hat eine hierarchische Struktur" liefern wenig bis überhaupt keine Information

- jedes System kann als hierarchisches System repräsentiert sein mit nur einer Ebene und einem Systembestandteil
- jedes System kann in Einzelteile zerlegt werden für die sich eine Relation ausklügeln lässt, um das System hierarchisch darzustellen

**Methode der Aufteilung** des Systems in seine Einzelbestandteile *und* die **Art der Relation** müssen vorgegeben werden

- anderenfalls bleiben o.g. Aussagen inhaltsleer und bedeutungslos

Entscheidungen zur Konzeption eines hierarchisch strukturierten Systems können die Klasse möglicher Systeme (Lösungen) einschränken:

pros
- das erstellte System verfügt über die gewünschten Vorteile

cons
- es bringt ggf. aber auch Nachteile mit sich





## Gliederung

1 Einleitung
- Hierarchische Struktur

2 Arten von Hierarchie
- Programmhierarchie
- Prozesshierarchie
- Mittelvergabehierarchie
- Schutzhierarchie

3 Funktionale Hierarchie
- Benutzthierarchie
- Hierarchiebildung
- Fallbeispiel

4 Zusammenfassung





## Hierarchie abstrakter Maschinen: $R \models$ „benutzt"

**Programmhierarchie**, in der die Systembestandteile Unterprogramme und wie Prozeduren aufrufbar — oder Makros expandierbar — sind [2, 3]

- jedes dieser Programme erfüllt einen bestimmten Zweck, z.B.:

```
erledige FNUZ;
   finde nächste ungerade Zahl in Folge;
   rufe KUZA, falls keine ungerade Zahl auffindbar;
basta.
```

- für Programme $p_i$ und $p_j$ kann „benutzt" wie folgt definiert sein [14]:

$$USES(p_i, p_j) \iff \begin{array}{l} p_i \text{ ruft } p_j \text{ und} \\ p_i \text{ ist fehlerhaft, sollte } p_j \text{ nicht funktionieren} \end{array}$$

- daraus folgt für $p_i = FNUZ$, $p_j = KUZA$: $USES(p_i, p_j) =$ falsch
  - Aufgabe von *FNUZ* ist es u.a., *KUZA* bedingt aufzurufen
  - Zweck und Korrektheit von *KUZA* ist aber irrelevant für *FNUZ*





## Hierarchie abstrakter Maschinen (Forts.)

Ausschluss von Aufrufen der *KUZA*-Art macht Hierarchiebildung möglich

- ohne diese Herausnahme könnte ein Programm nicht höher in der Hierarchie angeordnet sein, als die Maschine, die es benutzt
- typischer Fall: Programmunterbrechung $\mapsto$ *trap*, *interrupt*
  - die Hardware ruft bedingt, im Ausnahmefall, eine Softwareroutine auf

Programme sind hierachisch strukturiert, wenn die Relation „benutzt" Ebenen von Unterprogrammen wie zuvor beschrieben festlegt

- die Relation zwischen Programmen tieferer und höherer Ebenen entspricht der Relation zwischen Hardware und Software
- weshalb der Begriff „abstrakte Maschine" allgemein gebräuchlich ist

**Anmerkungen**

- keine Annahmen über interne Abläufe/Strukturen der Programme
- tiefere Ebenen sind allemal ohne die höheren Ebenen nutzbar
- Aufteilung der Programme in Ebenen oder Module ist orthogonal

## Hierarchie sequentieller Prozesse: $R \models$ „beauftragt“

**Aktivitäten** eines Systems — genauer: THE [2] — werden über (pseudo-) parallele, d.h. gleichzeitige, sequentielle „Prozesse“ organisiert[1]

- Motivation dafür ist, das System relativ unempfänglich zu machen:
  1. in Bezug auf die Anzahl der verfügbaren (realen) Prozessoren und
  2. hinsichtlich der relativen Geschwindigkeiten dieser Prozessoren
- die Abfolge der Ereignisse innerhalb eines Prozesses ist sodann vergleichsweise einfach zu bestimmen
- im Gegensatz zur Ereignisabfolge in verschiedenen Prozessen, die i.A. als unvorhersehbar/unberechenbar festzuhalten ist
  - bei unbekannten relativen Geschwindigkeiten der Prozessoren

Betriebsmittelvergabe erfolgt vermittels Prozesse, die darüberhinaus Arbeitsaufträge und Informationen austauschen

- zur Durchführung einer Aufgabe, kann **Arbeitsteilung** geschehen
- ein Prozess „beauftragt“ andere zur Übernahme von Teilaufgaben

[1] Auch als „Habermann“-Hierarchie [6] bezeichnet.





## Hierarchie sequentieller Prozesse (Forts.)

Relation „beauftragt“ legt eine Herarchie „stimmiger Kooperation“ [6] unter den beteiligten Prozessen fest

- im Falle eines (prozessstrukturierten) Betriebssystems gilt zu zeigen:
  - eine Systemanforderung ruft eine endliche Anzahl von Anforderungen an individuelle Prozesse hervor, um bearbeitet zu werden
  - darüberhinaus ist diese Anzahl einigermaßen klein
- bei vorliegender hierarchischer Struktur reicht es. . .
  1. jeden Prozes einzeln zu untersuchen und sicherzustellen, dass
  2. jede an ihn gestellte Anforderung immer nur eine endliche Zahl von Anforderungen an andere Prozesse nach sich zieht
- definiert die Relation keine Hierarchie, ist globale Analyse notwendig
  - die wegen dann unvorhersehbaren Ereignisabfolgen i.A. schwer ist

**THE: Beide bisher untersuchten Hierarchien decken sich!**

- jede abstrakte Maschine ist durch gleichzeitige Prozesse realisierbar
- jeder davon kann Prozesse tieferer abstrakter Maschinen beauftragen





## Hierarchie sequentieller Prozesse (Forts.)

Programmhierarchie ist immer nur dann von Bedeutung, wenn an dem Programm gearbeitet, es also konstruiert oder verändert wird

- sind die Programme Makros, hinterlassen sie keine Spur im System

Prozesshierarchie (nach Habermann [6]) ist dagegen eine Einschränkung auf das Laufzeitverhalten des Systems; darüberhinaus [14]:

> *Die von Habermann aufgestellten Theoreme gelten auch dann, sollte ein durch ein Programm tieferer Ebene der (Programm-) Hierarchie kontrollierter Prozess einen Prozess „beauftragen“, den seinerseits ein Programm kontrolliert, das ursprünglich auf höherer Ebene in der (Programm-) Hierarchie lag.*

**Beachte: Ein Mikrokern [10] allein impliziert keine Prozesshierarchie!**

- der Mikrokern kann Ebene$_0$ einer Programmhierarchie darstellen
- ggf. mit Prozesshierarchie ab Ebene$_{i, i>0}$: Thoth [1], AX [16]





## Hierarchie verwalteter Betriebsmittel: $R \models$ „belegt von“

**Mittelvergabehierarchie**, erzwungen auf Grundlage der den Prozessen zugeschriebenen Eigentümerschaft von Betriebsmitteln

- in RC4000 [8] ursprünglich auf Speicherbereiche beschränkt
- später um die Kontrolle weiterer Betriebsmittel verallgemeinert [19]

Betriebsmittel werden dabei allerdings nicht immer direkt den Prozessen zugeschrieben, die diese zu verwenden beabsichtigen

- ggf. verfügen administrative Einheiten über die Betriebsmittel und kontrollieren die Zuteilung an andere Prozesse
  - so lässt sich Zyklenentstehung im „belegt von“-Graphen vorbeugen
  - da die Betriebsmittelzuteilung die lineare Ordnung zusichert
- administrative Einheit $\equiv$ **Betriebsmittelzuteiler**
  - definiert für jede Ebene$_{i, i \geq 0}$ in der Hierarchie[2]

- dasselbe Betriebsmittel kann auf mehreren Ebenen verwaltet werden
  - z.B. die ebenenspezifische exklusive Belegung der CPU

[2] THE basierte stattdessen auf einen zentralen Zuteiler, dem *Banker*.

## Hierarchie verwalteter Betriebsmittel (Forts.)

Koinzidenz mit einer Programm- oder Prozesshierarchie ist nicht gegeben bzw. nicht selbstverständlich

- eine Betriebsmittelvergabehierarchie ist nicht als Alternative zu sehen, die eine Programm-/Prozesshierarchie ersetzen könnte
- vielmehr ist sie eine Ergänzung, z.B. zur *Verklemmungsvorbeugung*

### Anmerkungen

- nachteilig ist die Gefahr schlechter Betriebsmittelauslastung
  - manche Prozesse erfahren Mangel, andere Überfluss an Betriebsmittel
  - Ursache: einzelne Ebenen haben eigene Betriebsmittelzuteiler
- ggf. hoher Mehraufwand bei angespannter Betriebsmittelauslastung
  - Betriebsmittelanforderungen müssen die Hierarchie (Prozesswechsel) durchlaufen, bevor sie abgewiesen oder zugelassen werden
  - beispielsweise Speicherverwaltung: 1. Benutzer- 2. Systemebene; im System, 3. Platzierung; bei VM, 4. lokale und 5. globale Ersetzung





## Hierarchie spezifischer Schutzzonen: $R \models$ „zugreifbar“

**Schutzdomänen**, hier ringartig organisiert, ersetzen das konventionelle zweistufige Modell[3] [5, 12] $\models$ **Schutzhierarchie**

- anfangs (mit Multics) nur rein in Software implementiert
  - auf Grundlage des zweistufigen Ansatzes (modifizierte GE 645)
- spätere Hardwarerealisierung (B 6000 [18]) $\rightsquigarrow$ Leistungsgewinn

  innere Ringe: tiefere Ebenen, mehr sensitiv für Daten/Sicherheit

  äußere Ringe: höhere Ebenen, weniger sensitiv für Daten/Sicherheit
- nur untere Ebenen haben uneingeschränkten Zugriff auf höhere

### Beachte: Schutzhierarchie $\neq$ Programmhierarchie

...obwohl die geschützten Objekte Programme sind:

- die Programmaufrufe können in beide Richtungen geschehen und
- Programme tieferer Ebenen können von Programmen höherer Ebenen profitieren, um ihre Funktion(en) zu erfüllen

[3] Hauptsteuerprogramm (engl. *supervisor*) unten, Benutzerprogramm oben.





## Hierarchie spezifischer Schutzzonen (Forts.)

### Verwendung der Schutzringe (B 6000) in Multics

| Ring | Funktion | Bereich |
|---|---|---|
| 7<br>6 | Subsysteme | Anwendungssystem |
| 5<br>4 | Benutzerprogramme | Anwendungssystem |
| 3<br>2 | Subsysteme/Dienstprogramme | Betriebssystem |
| 1<br>0 | Hauptsteuerprogramm | Betriebssystem |

### *post* Multics: Schutzhierarchie auf Basis von Befähigungen

- Hydra [20] $\mapsto$ C.mmp, Cal [9] $\mapsto$ CDC 6400; iAPX 432 [13]





## Hierarchie spezifischer Schutzzonen (Forts.)

### Verwendung der Schutzringe (Intel 80386) in OS/2

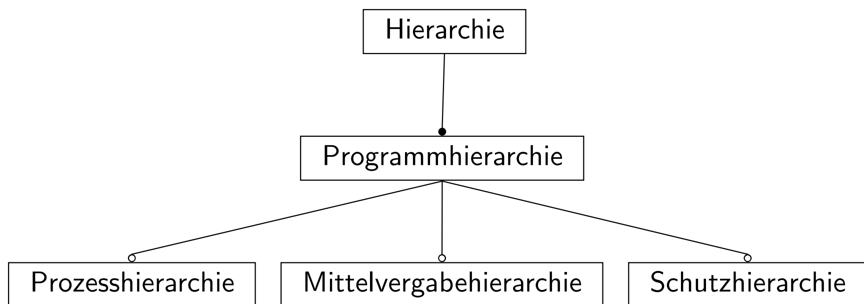
| Ring | Funktion | Bereich |
|---|---|---|
| 3 | keine | ungenutzt |
| 2 | Anwendungsprozeduren | Benutzerprogramme |
| 1 | Ein-/Ausgabeprozeduren | Benutzerprogramme |
| 0 | Kern und Gerätetreiber | Betriebssystem |

### Anmerkung: Multics, Hydra, Cal, OS/2 — Nischendasein

Schutzhierarchien haben sich (bisher) nicht weitläufig durchgesetzt

- eine solche Hierarchie in „unstrukturierte“ Systeme nachträglich einbringen zu wollen, ist alles andere als einfach bzw. scheitert
- eine Programmhierarchie als „Rückgrat“ kann sehr förderlich sein

## Gegenüberstellung




**Programmhierarchie** bildet *das* Rückgrat
- ist bzgl. ihrer „Spezialisierungen“ auszulegen

**andere** bilden „spezialisierte“ Hierarchien
- Prozess, Mittelvergabe, Schutz, ...
- alles braucht Programme, um zu sein!!!





## Gliederung


1. Einleitung
   - Hierarchische Struktur
2. Arten von Hierarchie
   - Programmhierarchie
   - Prozesshierarchie
   - Mittelvergabehierarchie
   - Schutzhierarchie
3. Funktionale Hierarchie
   - Benutzthierarchie
   - Hierarchiebildung
   - Fallbeispiel
4. Zusammenfassung






## Rekapitulation: Ereigniszustellung (Kap. III-2, S. 2*ff*)





## Wiederholung: Benutztbeziehung

Grundlage jeder Programmhierarchie, deren Ebenen entsprechend einer bestimmten funktionalen Hierarchie zueinander angeordnet sind:

$$[15]\ \text{Benutzthierarchie} \overset{[4,\ S.\ 151]}{\equiv} \text{funktionale Hierarchie}\ [7]$$

- für zwei Programme $A$ und $B$ bedeutet $A$ „benutzt“ $B$, ...
  - wenn die korrekte Ausführung von $B$ notwendig ist für die Vollendung der Arbeit von $A$
  - wenn es Situationen gibt, in denen das korrekte Funktionieren von $A$ abhängt vom Vorhandensein einer korrekten Implementierung von $B$
- „benutzt“ unterscheidet sich von „ruft“ (z.B. Prozeduraufruf)
  1. manche Programmaufrufe sind keine Ausprägung von „benutzt“
     ↪ der Aufruf von *KUZA* (vgl. S. 7)
  2. Programm $A$ kann $B$ „benutzen“, obwohl es Programm $B$ nie aufruft
     ↪ asynchrone Programmunterbrechungen d.h. *Interrupts*
- „benutzt“ $\models$ „erfordert die Existenz einer korrekten Version von“

## Benutztbeziehung: „benutzt" vs. „ruft"

1. manche Programmaufrufe sind keine Ausprägung von „benutzt"
   - wenn von $A$ gefordert ist, dass es $B$ nur bedingt „ruft", dann erfüllt $A$ seine Spezifikation sobald der Aufruf an $B$ generiert wurde
     - dies trifft insbesondere auch dann zu, falls $B$ falsch oder abwesend ist
   - ein Korrektheitsnachweis für $A$ muss lediglich Annahmen über die Art und Weise des Aufrufs an $B$ machen ⇝ vgl. S. 7, Ausnahme werfen
2. Programm $A$ kann $B$ „benutzen", obwohl es $B$ nie aufruft
   - die meisten Programme (eines Rechensystems) gehen davon aus, dass Unterbrechungsbehandlungsroutinen korrekt funktionieren
     - den Prozessorzustand unterbrochener Programme invariant halten
   - d.h., dass diese von der Hardware ausgelösten Routinen terminieren, obwohl ein Aufruf an sie in keinem Programm kodiert ist

**Beachte**

zu 2. lässt den Schluss zu, dass Unterbrechungsbehandlungsroutinen die unterste Ebene der Programmhierarchie ausmachen (können)
↪ Funktionen zur Zustandssicherung/-wiederherstellung





## Benutztbeziehung: „benutzt" vs. „erfordert die Existenz"

**„benutzt" $\models$ „erfordert die Existenz einer korrekten Version von"**

1. was bedeutet „erfordert die Existenz"?
2. was ist unter „korrekte Version" zu verstehen?

zu 1. das Vorhandensein des Namens eines Artefaktes im Entwurf *oder* in der Implementierung

zu 2. eine Variante der Implementierung von $B$, die gemäß der für $A$ geltenden Spezifikation korrekt ist
- die Spezifikation der Schnittstelle von $B$ erfüllt die in der Spezifikation von $A$ niedergelegten Anforderungen
- dies umfasst alle funktionalen/nichtfunktionalen Eigenschaften

**Beachte**

- auch die *leere Implementierung* einer Funktion ist zu berücksichtigen
- keine Spuren zu hinterlassen, kann existenziell und korrekt sein!





## Zuweisung von Programmen zu Ebenen in der Hierarchie

**Grundregeln**:

1. Ebene$_0$ umfasst die Menge aller Programme, die kein anderes Programm „benutzen"
2. Ebene$_i$, für $i > 0$, umfasst die Menge aller Programme, die wenigstens ein Programm auf Ebene$_{i-1}$ „benutzen"
   - jedoch kein Programm einer Ebene höher als $i - 1$

Existenz einer solchen hierarchischen Ordnung ermöglicht es, dass jede Ebene eine test- und nutzbare Teilmenge des Systems bildet
- nützliche Eigenschaft, um beliebig größere Systeme zu konstruieren
- wesentlich für die Entwicklung einer breiten Familie von Systemen

**Beachte** ↪ [15, S. 4]

*Die Aufteilung des Systems in frei rufbare Unterprogramme ist gleichzeitig mit den Entscheidungen zur Benutztbeziehung zu führen, da sich beides gegenseitig beeinflusst*





## Entscheidungshilfe zu $A$ „benutzt" $B$

1. $A$ ist wesentlich einfacher, da es $B$ benutzt
   - $B$ wurde bereitgestellt, um $A$ zu unterstützen
2. $B$ wäre nicht wesentlich einfacher, wenn es $A$ benutzte
   - $B$ funktioniert zweifelsfrei ohne $A$, aber:
     - Hilfestellung durch $A$ könnte $B$ einfacher ausgelegt erscheinen lassen
     - Kontextwissen in $A$ könnte Verfahren in $B$ effizienter ablaufen lassen
3. es gibt eine nutzbare Teilmenge, die $B$ enthält aber $A$ nicht benötigt
   - $A$ unterstützt nur bestimmte Anwendungsfälle, andere dagegen nicht
   - $B$ ist Plattform zur Unterstützung verschiedener Anwendungsfälle
   - $A$ dient mehr einer Spezialisierung, $B$ eher der Generalisierung
4. es gibt keine vorstellbare Teilmenge, die $A$ aber nicht $B$ enthält
   - denn dann würde $A$ die von $B$ bereitgestellte Funktion nicht erfordern

**Anmerkung**

zu 2. der Fall ist ggf. Grund zur Neugestaltung oder Schichtanordnung

zu 4. beachte hier auch die *leere Implementierung*: `inline void B() {}`

## Schichtanordnung in Betriebssystemen

Programme eines Betriebssystems in eine **Benutzthierarchie** zu bringen, ist bestimmt durch die zu unterstützende **Rechnerbetriebsart**

- die Systemfunktionen lassen sich keiner Ebene fest zuschreiben
  - z.B. „benutzt“ Einplanung Programmunterbrechungen nur bedingt
  - nicht bei FCFS oder SPN, wohl aber bei RR, SRTF oder HRRN [17]
- maßgeblich ist die mächtigste Variante einer „benutzten“ Funktion
  - bspw. HRRN annehmen, obwohl die aktuelle Konfiguration FCFS fährt
  - Einplanung daraufhin einer geeignet hoch liegenden Ebene zuordnen
- einen ganzheitlichen Entwurf zu liefern, ist überaus anspruchsvoll
  - im Nachhinein konkretisieren sich anfangs noch recht vage Sichten
  - entsprechend ist die Schichtanordnung der Funktionen anzupassen
- gleichwohl anstreben, den Entwurf funktional vollständig auszulegen

**Beachte ↪ Externe vs. interne Sicht**

- der Platz für Betriebssysteme in Mehrebenenmaschinen ist etabliert
- nicht aber die Mehrebenenmaschinenstruktur eines Betriebssystems





## Schichtanordnung *de*LUXE: Randbedingungen

Umstände, die die Herleitung der **Benutzthierarchie** von Funktionen eines Betriebssystems schwierig gestalten:

**Konsistenz**
- betrifft die **Invarianz** des Prozessorstatus', für Prozesse:
  1. die sich nicht im Zustand LAUFEND befinden oder
  2. die im Zustand LAUFEND unterbrochen worden sind

**Lebendigkeit**
- betrifft die Prozessen gewährte **Fortschrittsgarantie**:
  1. für das Gesamtsystem (Sperrfreiheit) oder
  2. für jeden einzelnen Prozess (Wartefreiheit)

**Integrität**
- betrifft die **Unversehrtheit** von Adressräumen:
  1. aller Anwendungsprogramme und
  2. des Betriebssystems

⇒ wie sind die fraglichen Funktionen „benutzthierarchisch“ anzusiedeln?





## Schichtanordnung *de*LUXE: Funktionale Anordnung (1)

Konsistenz

| Ebene | Funktion | Abstraktion |
|---|---|---|
| 5 | Bedingungssynchronisation | Prozess |
| 4 | Wechselseitiger Ausschluss | |
| 3 | Ein- und Umplanung | |
| 2 | Einlastung | |
| 1 | Transaktionale Operationen | Prozessor |
| 0 | Statussicherung und -wiederherstellung | |

- Bedingungssynchronisation „benutzt“ wechselseitigen Ausschluss in logischer Hinsicht: Option zum Schutz des allgemeinen Semaphors
- transaktionale Operationen (z.B. TAS, CAS) sind auch in Software nachbildbar und können somit aus Befehlsfolgen bestehen
- dass Statussicherung und -wiederherstellung (hier) die unterste Ebene ausmacht, bedingt sich durch asynchrone Programmunterbrechungen





## Schichtanordnung *de*LUXE: Funktionale Anordnung (2)

Lebendigkeit

| Ebene | Funktion | Abstraktion |
|---|---|---|
| 6 | Bedingungssynchronisation | Prozess |
| 5 | Wechselseitiger Ausschluss | |
| 4 | Ein- und Umplanung | |
| 3 | Einlastung | |
| 2 | Unterbrechungsbehandlung und -weiterleitung | Vor-/Nachspann |
| 1 | Transaktionale Operationen | Prozessor |
| 0 | Statussicherung und -wiederherstellung | |

- Unterbrechungsbehandlung (FLIH) muss terminieren, dass Verfahren höherer Ebenen ihre Fortschrittsgarantien durchsetzen können
- ebenso Unterbrechungsweiterleitung (SLIH), die ebenfalls asynchron zu Funktionen höherer Ebenen erfolgt und Prozesse verzögert

## Schichtanordnung *de* LUXE: Funktionale Anordnung (3)

Integrität

| Ebene | Funktion | Abstraktion |
|---|---|---|
| 6 | Bedingungssynchronisation | Prozess |
| 5 | Wechselseitiger Ausschluss | |
| 4 | Ein- und Umplanung | |
| 3 | Einlastung | |
| 2 | Unterbrechungsbehandlung und -weiterleitung | Vor-/Nachspann |
| 1 | Transaktionale Operationen | Prozessor |
| ↑ 0 | Statussicherung und -wiederherstellung | |
| ↓ | Segmentvalidierung | Adressraum |

- korrekte Programmabläufe implizieren Adressraumintegrität, für die in logischer Hinsicht Segmentierungskonzepte hilfreich sind
- die zu „benutzenden" Maßnahmen greifen vor oder zur Systemlaufzeit





## Schichtanordnung *de* LUXE: Funktionale Anordnung (4)

Funktionen, die in BST behandelt wurden

| Ebene | Primitiven |
|---|---|
| 6 | $P_g$, $V_g$ |
| 5 | $P_b$, $V_b$; lock, unlock; acquire, release |
| 4 | clean, doubt, snoop, trace; delay, ready, seize, stash, $unban_{sad}$, ... |
| 3 | apply, being, board; invoke, launch, regain, resume; switch |
| 2 | check, clear, defer, entry, start, store, $unban_{tip}$ |
| 1 | CAS, FAA, TAS |
| ↑ 0 | pull, push; dump |
| ↓ | allot, prune |

**Adressraum ↪ Segmentvalidierung**

*allot*
- teilt (einem Prozess) einen zusätzlichen Adressbereich zu

*prune*
- beschneidet den Adressraum, invalidiert einen Adressbereich





## Modularisierung und Hierarchiebildung (Kap. III-2, S. 5 (Forts.))

Programme, die in einer Schicht zusammengefasst sind, teilen sich nicht zwingend dasselbe Wissen über die dortigen Datenstrukturen

- Modul und Schicht sind zwei voneinander unabhängige Konzepte
- egal, ob Modul ein abstrakter Datentyp oder eine Klasse darstellt

Schicht — einer funktionalen Hierarchie — fasst Programme derselben **Niveaumenge** zusammen, bezogen auf die Benutztrelation

- allen ist die gleiche „Abhängigkeitsebene" zugeordnet
- alle „benutzen" den gleichen (logischen) Unterbau
- alle werden von möglicherweise verschiedenen Oberbauten „benutzt"

**Schichtanordnung ↪ *de* LUXE**

- $Ebene_0$ umfasst Programme verschiedener Abstraktionen
  - Teile von Funktionen zur Prozessor- und Adressraumverwaltung
- unter diesen Programmen lässt sich keine Benutztrelation mehr finden





## Gliederung


1. Einleitung
   - Hierarchische Struktur
2. Arten von Hierarchie
   - Programmhierarchie
   - Prozesshierarchie
   - Mittelvergabehierarchie
   - Schutzhierarchie
3. Funktionale Hierarchie
   - Benutzthierarchie
   - Hierarchiebildung
   - Fallbeispiel
4. Zusammenfassung

## Resümee

Struktur $\models$ partielle Beschreibung eines Systems
hierarchisch $\models$ Relation zwischen Teilepaaren/Ebenen

### Arten von Hierarchie

- Programm-, Prozess-, Mittelvergabe- und Schutzhierarchie
- Familie von Systemen $\Longleftrightarrow$ Programmhierarchie

### funktionale Hierarchie

- stufenweiser Maschinenentwurf, basierend auf Funktionen
- Benutztbeziehung, Unterschied zur Aufrufbeziehung
- Hierarchiebildung, Schichtanordnung

### Fallbeispiel

- Prozesssynchronisation, -ein/-umplanung, -einlastung
- Schichtanordnung *de*LUXE





## Literaturverzeichnis

[1] CHERITON, D. R.: *Multi-Process Structuring and the Thoth Operating System*. Ontario, Canada, University of Waterloo, Diss., 1978

[2] DIJKSTRA, E. W.: The Structure of the THE-Multiprogramming System. In: *Communications of the ACM* 11 (1968), Mai, Nr. 5, S. 341–346

[3] DIJKSTRA, E. W.: Complexity Controlled by Hierarchical Ordering of Functions and Variability. In: NAUR, P. (Hrsg.) ; RANDELL, B. (Hrsg.): *Software Engineering, Report on the Conference of the NATO Science Committee*. Brussels, Belgium : Science Affairs Division NATO, Okt. 1969, S. 181–186

[4] GARLAN, D. ; HABERMANN, J. F. ; NOTKIN, D. : Nico Habermann's Research: A Brief Retrospective. In: *Proceedings of the 16th International Conference on Software Engineering (ICSE '94)*. New York, NY, USA : ACM Press, 1994, S. 149–153

[5] GRAHAM, R. C.: Protection in an Information Processing Utility. In: *Communications of the ACM* 11 (1968), Mai, Nr. 5, S. 365–369

[6] HABERMANN, A. N.: *On the Harmonious Co-Operation of Abstract Machines*. Eindhoven, The Netherlands, Technische Hogeschool Eindhoven, Diss., Okt. 1967. – 115 S.

[7] HABERMANN, A. N. ; FLON, L. ; COOPRIDER, L. W.: Modularization and Hierarchy in a Family of Operating Systems. In: *Communications of the ACM* 19 (1976), Mai, Nr. 5, S. 266–272





## Literaturverzeichnis (Forts.)

[8] HANSEN, P. B.: The Nucleus of a Multiprogramming System. In: *Communications of the ACM* 13 (1970), Apr., Nr. 4, S. 238–241/250

[9] LAMPSON, B. W. ; STURGIS, H. E.: Reflections on an Operating System Design. In: *Communications of the ACM* 19 (1976), Mai, Nr. 5, S. 251–265

[10] LIEDTKE, J. : Towards Real Microkernels. In: *Communications of the ACM* (1996), Sept., S. 70–77

[11] MCCLUSKEY, E. J. (Hrsg.) ; BREDT, T. (Hrsg.) ; LAMPSON, B. W. (Hrsg.): *Proceedings of the 3rd ACM Symposium on Operating System Principles (SOSP '71)*. Bd. *6*. New York, NY, USA : ACM Press, 1971 (ACM SIGOPS Operating Systems Review 1–2)

[12] ORGANICK, E. I.: *The Multics System: An Examination of its Structure*. MIT Press, 1972. – ISBN 0–262–15012–3

[13] ORGANICK, E. I.: *A Programmer's View of the Intel 432 System*. New York, NY, USA : McGraw-Hill, Inc., 1976. – ISBN 0–07–047719–1





## Literaturverzeichnis (Forts.)

[14] PARNAS, D. L.: On a 'Buzzword': Hierarchical Structure. In: ROSENFELD, J. L. (Hrsg.): *Information Processing 74, Proceedings of the IFIP Congress 74*. New York, NY, USA : North-Holland Publishing Company, 1974. – ISBN 0–7204–2803–3, S. 336–339

[15] PARNAS, D. L.: Some Hypothesis About the "Uses" Hierarchy for Operating Systems / TH Darmstadt, Fachbereich Informatik. 1976 (BSI 76/1). – Forschungsbericht

[16] SCHRÖDER-PREIKSCHAT, W. : *Eine Familie von UNIX-ähnlichen Betriebssystemen — Anwendung von Prozessen und des Nachrichtenübermittlungskonzeptes bem strukturierten Betriebssystementwurf*, Technische Universität Berlin, Diss., Dez. 1986

[17] SCHRÖDER-PREIKSCHAT, W. ; KLEINÖDER, J. : *Systemprogrammierung*. `http://www4.informatik.uni-erlangen.de/Lehre/WS08/V_SP`, 2008 *ff.*

[18] SCHROEDER, M. D. ; SALTZER, J. H.: A Hardware Architecture for Implementing Protection Ring. In: [11], S. 42–54

[19] VARNEY, R. C.: Process Selection in a Hierarchical Operating System. In: [11], S. 106–108

[20] WULF, W. A. ; COHEN, E. S. ; CORWIN, W. M. ; JONES, A. K. ; LEVIN, R. ; PIERSON, C. ; POLLACK, F. J.: HYDRA: The Kernel of a Multiprocessor Operating System. In: *Communications of the ACM* 17 (1974), Jun., Nr. 6, S. 337–345