

# 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 unempfindlich bzgl. Anzahl der verfügbaren Prozessoren und ihren relativen Geschwindigkeiten

**Mittelvergabehierarchie** organisiert ein System in problemspezifische Betriebsmittelzuteiler und -verwalter

**Schutzhierarchie** verbessert wesentlich die Vertrauenswürdigkeit 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<sub>0</sub> ist Menge von Teilen  $\alpha$ , so dass es kein  $\beta$  gibt mit  $R(\alpha, \beta)$
- 2 Ebene <sub>$i, i > 0$</sub>  ist Menge von Teilen  $\alpha$ , so dass gilt:
  - 1 es existiert ein  $\beta$  auf Ebene <sub>$i-1$</sub>  mit  $R(\alpha, \beta)$  und
  - 2 falls  $R(\alpha, \gamma)$ , dann liegt  $\gamma$  auf Ebene <sub>$i-1$</sub>  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]:

$$USES(p_i, p_j) \iff p_i \text{ ruft } p_j \text{ und}$$

$p_i$  ist fehlerhaft, sollte  $p_j$  nicht funktionieren

- daraus folgt für  $p_i = FNUZ$ ,  $p_j = KUZA$ :  $USES(p_i, p_j) = \text{falsch}$ 
  - 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 hierarchisch 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<sup>1</sup>

- Motivation dafür ist, das System relativ unempfindlich 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 unbekanntem relativen Geschwindigkeiten der Prozessoren

Betriebsmittelvergabe erfolgt mittels Prozesse, die darüberhinaus Arbeitsaufträge und Informationen austauschen

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

---

<sup>1</sup>Auch als „Habermann“-Hierarchie [6] bezeichnet.

# Hierarchie sequentieller Prozesse (Forts.)

Relation „beauftragt“ legt eine Hierarchie „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 Prozess 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<sub>0</sub> einer Programmhierarchie darstellen
- ggf. mit Prozesshierarchie ab Ebene <sub>$i, i > 0$</sub> : 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<sup>2</sup>

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

<sup>2</sup>THE basiert stattdessen auf einem zentralen Zuteiler, dem *Banker*.



# Hierarchie verwalteter Betriebsmittel (Forts.)

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

- eine Betriebsmittelvergabe-hierarchie 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<sup>3</sup> [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

<sup>3</sup>Hauptsteuerprogramm (engl. *supervisor*) unten, Benutzerprogramm oben.

# Hierarchie spezifischer Schutzzonen (Forts.)

## Verwendung der Schutzringe (B 6000) in Multics

Ring	Funktion	Bereich
7 6	Subsysteme	Anwendungssystem
5 4	Benutzerprogramme	
3 2	Subsysteme/Dienstprogramme	Betriebssystem
1 0	Hauptsteuerprogramm	

## *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

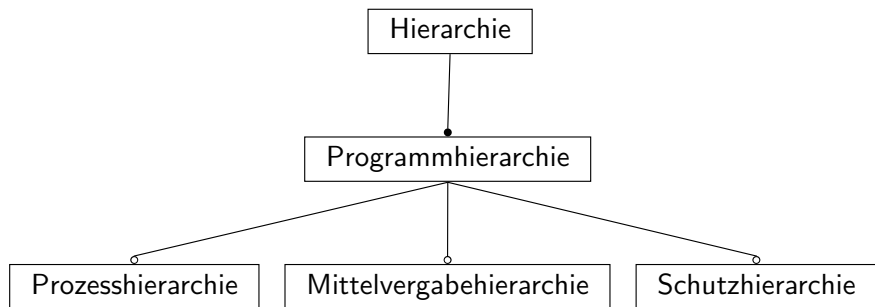
Ring	Funktion	Bereich
3	keine	ungenutzt
2	Anwendungsprozeduren	Benutzerprogramme
1	Ein-/Ausgabeprozeduren	
0	Kern und Gerätetreiber	Betriebssystem

### 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. 2ff)

3.1.2 Ereigniszustellung 1 Grundlagen 1.1 Funktionale Hierarchie

## Gliederung

- 1 Grundlagen
  - 1 Funktionale Hierarchie

- 1 Grundlagen
  - 1 Funktionale Hierarchie
    - 1 Funktionale Hierarchie

- 1 Zusammenfassung

©wosch (Lehrstuhl Informatik 4) Betriebssystemtechnik SS 2012 7 / 24

3.1.2 Ereigniszustellung 1 Grundlagen 1.1 Funktionale Hierarchie

## Modularisierung und Hierarchiebildung

Begrifflichkeiten „Ebene“ und „Modul“ decken sich nicht zwangsläufig, zwischen beiden besteht keine notwendige Beziehung

Ebene eine Menge von Funktionsnamen

- implementiert durch Funktionen tieferer Ebenen

Modul kapselt Datenstrukturen (ggf.) und eine Menge von Funktionen

- die Funktionen teilen sich Wissen über Entwurfsentscheidungen, z.B. Details der Datenstrukturen
- Geheimnisprinzip (engl. *information hiding*, [7])

**Beachte** → [7]

- eine Ebene kann in mehrere verschiedene Module aufgeteilt sein
- ein einzelnes Modul kann selektiv mehrere Ebenen überspannen
  - d.h. eine **Schichtenanordnung** (engl. *sandwich*, [8]) bilden

©wosch (Lehrstuhl Informatik 4) Betriebssystemtechnik SS 2012 8 / 24

3.1.2 Ereigniszustellung 1 Grundlagen 1.1 Funktionale Hierarchie

## Stufenweiser Maschinenentwurf

Ansatz: **Programmarchitektur** bzw. Hierarchie abstrakter Maschinen, wie z.B. mit THE [2], Venus [6] oder FAMOS [4] vorgeführt:

- das System stellt sich als Hierarchie von Abstraktionsebenen dar

Dabei definiert sich eine Ebene nicht nur durch die Abstraktion, die sie bereitstellt (z.B. virtuellen Speicher), sondern auch durch die zur Umsetzung der Abstraktion benutzten Betriebsmittel.

- tiefer (näher an der Hardware liegende) Ebenen besitzen keine Kenntnis über die Betriebsmittel höherer Ebenen
- höhere Ebenen dürfen Betriebsmittel tieferer Ebenen nur mittels Funktionen eben dieser Ebenen nutzen
- die einzelnen Ebenen sind (logisch) fest voneinander abgegrenzt
- in Analogie zur Hardware und ihren Programmen (Software)

©wosch (Lehrstuhl Informatik 4) Betriebssystemtechnik SS 2012 9 / 24

3.1.2 Ereigniszustellung 1 Grundlagen 1.2 Benutzungshierarchie

## Benutzbeziehung

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

$$[8] \text{ Benutzungshierarchie} \stackrel{[8, S.150]}{=} \text{funktionale Hierarchie} [4]$$

- für zwei Programme A und B bedeutet A „benutzt“ B, ...
  - wenn die korrekte Ausführung von B notwendig ist für die Vollaufgabe 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)
  - manche Programmaufrufe sind keine Ausprägung von „benutzt“
    - Beispielweise der bedingte Aufruf einer *senespezifischen Prozedur*
  - Programm A kann B „benutzen“, obwohl es Programm B nie aufruft
    - asynchrone Programmunterbrechungen d.h. *Interrupts*
  - „benutzt“  $\neq$  „erfordert die Existenz einer korrekten Version von“

©wosch (Lehrstuhl Informatik 4) Betriebssystemtechnik SS 2012 10 / 24

3.1.2 Ereigniszustellung 1 Grundlagen 1.1 Funktionale Hierarchie

## Hierarchiebildung basierend auf Funktionen

Strukturierung eines Systems in Ebenen sagt noch längst nichts aus über das **Zusammenspiel** der Ebenen untereinander<sup>1</sup>

- jede Ebene beinhaltet eine Menge von Funktionen, deren Namen statisch bekannt sind
  - was jedoch hinter dem Namen steht, ergibt sich ggf. erst zur Laufzeit
    - d.h. **dynamisches Binden** von Funktionen ist nicht ausgeschlossen
- Ebenen  $L_0, L_1, \dots, L_n$  sind so angeordnet, dass Funktionen in Ebene  $L_i$  ebenfalls  $L_{i+1}$  bekannt sind
  - je nach Ermessen von  $L_{i+1}$ , sind sie auch  $L_{i+2}$  bekannt, usw.
- Ebene  $L_0$  entspricht der Befehlsstzebene der Zielmaschine
  - jede Ebene stellt der nächst höheren Ebene neue „Hardware“ bereit

**Der Systementwurf ist hierarchisch, nicht seine Implementierung [4]**

- in einer funktionalen Hierarchie können die Funktionen Makros sein
- eine Funktionsaufrufkette kann in einen einzelnen Maschinenbefehl resultieren, wenn überhaugt

<sup>1</sup>Kapitel „Hierarchische Struktur“ greift diesen Punkt später erneut auf.

©wosch (Lehrstuhl Informatik 4) Betriebssystemtechnik SS 2012 11 / 24

3.1.2 Ereigniszustellung 1 Grundlagen 1.2 Benutzungshierarchie


## Schichtenanordnung (engl. *sandwich*)

Ebene gleichgesetzt mit Modul führt oft zu Situationen, in denen erst durch **gegenseitige Benutzung** Programme voneinander profitieren

- typisch, falls Modul als „Ebene der Abstraktion“ verstanden wird
  - vgl. auch S. 5 Modul  $\neq$  Ebene
- Konflikt, der Neugestaltung der betroffenen Ebenen erfordert
  - lineare Ordnung durchsetzen  $\sim$  **Zyklus aufbrechen**

**Auflösung**

Eines der Programme so in zwei Teile „schneiden“, dass sich beide Programme „benutzen“ können:



- $A \rightarrow A_1, A_2; A_1 \text{ „benutzt“ } B \text{ „benutzt“ } A_2$
- $B \rightarrow B_1, B_2; B_1 \text{ „benutzt“ } A \text{ „benutzt“ } B_2$

©wosch (Lehrstuhl Informatik 4) Betriebssystemtechnik SS 2012 12 / 24

# Wiederholung: Benutzbeziehung

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

[15] Benutzthierarchie  $\stackrel{[4, S. 151]}{\equiv}$  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“



# Benutzbeziehung: „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  $\rightsquigarrow$  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)
- $\rightarrow$  Funktionen zur Zustandssicherung/-wiederherstellung

# Benutzbeziehung: „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 Benutzbeziehung 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 $\leftrightarrow$ Externe vs. interne Sicht

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

# Schichtanordnung *deLUXE*: 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 *deLUXE*: 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 *deLUXE*: 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 *deLUXE*: 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 *deLUXE*: 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 <sub>sad</sub> , ...
3	apply, being, board; invoke, launch, regain, resume; switch
2	check, clear, defer, entry, start, store, unban <sub>tip</sub>
1	CAS, FAA, TAS
↑	pull, push; dump
0	
↓	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 Benutzrelation

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

### Schichtanordnung $\leftrightarrow$ deLUXE

- Ebene<sub>0</sub> umfasst Programme verschiedener Abstraktionen
  - Teile von Funktionen zur Prozessor- und Adressraumverwaltung
- unter diesen Programmen lässt sich keine Benutzrelation 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  $\iff$  **Programmhierarchie**

## funktionale Hierarchie

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

## Fallbeispiel

- Prozesssynchronisation, -ein/-umplanung, -einlastung
- Schichtanordnung *deLUXE*

# 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 beim 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](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