

Betriebssystemtechnik

Adressräume: Trennung, Zugriff, Schutz

IV. Hierarchien

Wolfgang Schröder-Preikschat

16. Mai 2019



Einleitung

Hierarchische Struktur

Arten von Hierarchie

Programmhierarchie

Prozesshierarchie

Mittelvergabehierarchie

Schutzhierarchie

Funktionale Hierarchie

Benutzthierarchie

Hierarchiebildung

Zusammenfassung



Voraussetzung: hierarchisch organisierte Softwarestrukturen à la

■ Programmhierarchie

- definiert verschiedene, problemspezifische Ebenen der Abstraktion
- fördert den Aufbau einer *Familie von Systemen*

■ Prozesshierarchie

- macht ein System relativ unempfindlich für die Anzahl der verfügbaren Prozessoren und ihren relativen Geschwindigkeiten

■ Mittelvergabehierarchie

- organisiert ein System in problemspezifische Betriebsmittelverwalter

■ Schutzhierarchie

- verbessert die Vertrauenswürdigkeit einzelner Systembestandteile
- erhöht die Sicherheit des Gesamtsystems

↪ Lernziel

- die für Betriebssysteme wichtigen Arten von Hierarchie erfassen



„Struktur“ bezieht sich auf die **partielle Beschreibung** eines Systems

- sie 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 entstehen lässt wie folgt:

- Ebene 0 ist Menge von Teilen α , so dass es kein β gibt mit $R(\alpha, \beta)$
- Ebene $i, i > 0$ ist Menge von Teilen α , so dass gilt:
 - i es existiert ein β auf Ebene $i-1$ mit $R(\alpha, \beta)$ und
 - ii falls $R(\alpha, \gamma)$, dann liegt γ auf Ebene $i-1$ oder niedriger

↪ Relation R ist repräsentiert als **gerichteter azyklischer Graph**



Aussagen der Art wie

„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 bleibt o.g. Aussage inhaltsleer und bedeutungslos
- ↪ diese Vorgaben können die Klasse möglicher Systeme einschränken
 - das erstellte System verfügt über die gewünschten Vorteile
 - es bringt (für andere Fälle) ggf. aber auch Nachteile mit sich



Einleitung

Hierarchische Struktur

Arten von Hierarchie

Programmhierarchie

Prozesshierarchie

Mittelvergabehierarchie

Schutzhierarchie

Funktionale Hierarchie

Benutzthierarchie

Hierarchiebildung

Zusammenfassung



Systembestandteile sind Unterprogramme und wie Prozeduren aufrufbar oder Makros expandierbar [2, 3]

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

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

- für Programme p_i und p_j kann „benutzt“ wie folgt definiert sein [15]:

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

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



- Ausschluss „bedingter Aufrufe“ macht Hierarchiebildung erst möglich
 - sonst 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

↪ Anmerkung:

- 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



Aktivitäten eines Systems über (pseudo-) parallele, d.h. gleichzeitige, sequentielle „Prozesse“ organisieren¹

- 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
- **beachte:** die Abfolge von Ereignissen innerhalb eines Prozesses ist 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

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



Prozesse stehen in einer Hierarchie „stimmiger Kooperation“ [6]

- im Falle eines derart strukturierten 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
- ↪ **beachte:** beide bisher untersuchten Hierarchien decken sich!
- jede abstrakte Maschine ist durch gleichzeitige Prozesse realisierbar
 - jeder davon kann Prozesse tieferer abstrakter Maschinen beauftragen



- eine **Programmhierarchie** ist *vor Laufzeit* des Systems bedeutsam
 - statisch: wenn Software konstruiert, entwickelt oder verändert wird
 - sind die Programme Makros, hinterlassen sie keine Spuren im System
- eine **Prozesshierarchie** (nach Habermann [6]) dagegen *zur Laufzeit*; darüberhinaus [15]:

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 [11] allein impliziert keine Prozesshierarchie!
- angenommen, er macht Ebene₀ in der (Programm-) Hierarchie aus
 - die Prozesshierarchie besteht dann erst ab Ebene _{$i, i > 0$}
 - Thoth [1] und AX [18] sind Beispiele dafür



Grundlage bildet die den Prozessen zugeschriebene Eigentümerschaft von Betriebsmitteln

- ursprünglich auf Speicherbereiche beschränkt, z.B. RC4000 [8]
- später um die Kontrolle weiterer Betriebsmittel verallgemeinert [21]

Betriebsmittel sind dabei nicht immer den Prozessen zugeschrieben, die diese zu verwenden beabsichtigen

- **administrative Einheiten** kontrollieren die Zuteilung an die Prozesse
 - diese Einheiten (Systemprozesse) agieren als Betriebsmittelzuteiler
 - definiert für jede Ebene $i, i \geq 0$ in der Hierarchie²
 - **lineare Ordnung** beugt Zyklenentstehung im „belegt von“-Graphen vor
- dasselbe Betriebsmittel kann auf mehreren Ebenen verwaltet werden
 - z.B. die ebenenspezifische exklusive Belegung der CPU

²THE basiert stattdessen auf einem zentralen Zuteiler, dem *Banker* [2].

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**

↪ **beachte:**

- 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



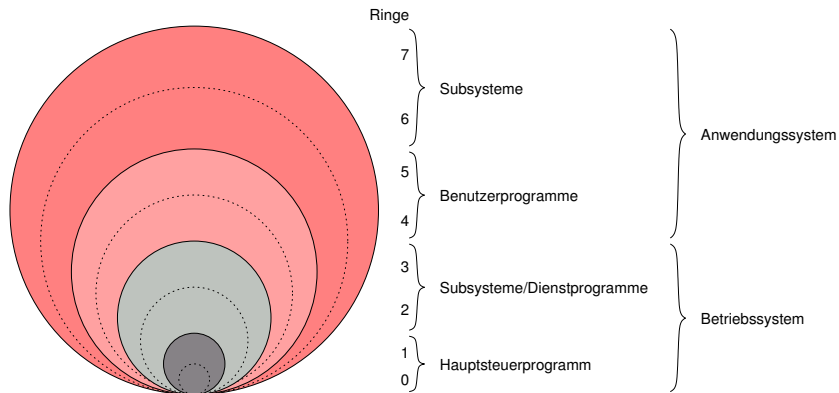
Schutzdomänen, hier ringartig organisiert, ersetzen das konventionelle zweistufige Modell³ [5, 13] \rightsquigarrow **Schutzhierarchie**

- anfangs (mit Multics) nur rein in Software implementiert
 - auf Grundlage des zweistufigen Ansatzes (modifizierte GE 645)
- spätere Hardwarerealisierung (B 6000 [20]) \rightsquigarrow Leistungsgewinn
 - innere Ringe sind mehr sensitiv für Sicherheit, äußere Ringe weniger
- nur untere Ebenen haben uneingeschränkten Zugriff auf höhere

\hookrightarrow **beachte:** Schutzhierarchie \neq Programmhierarchie

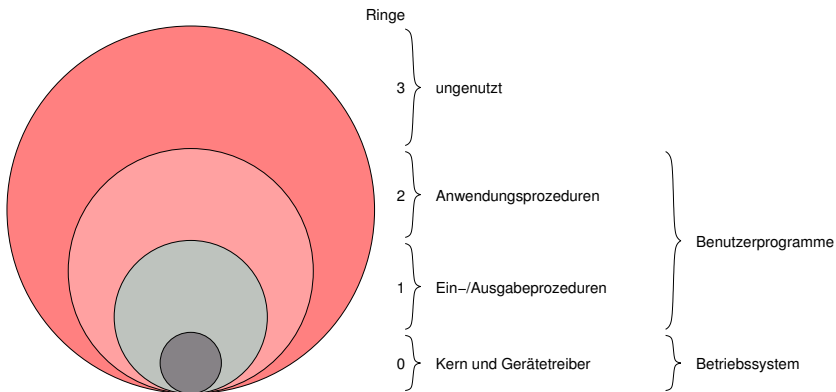
- obwohl die geschützten Objekte (auch) Programme sind:
 - die Programmaufrufe können in beide Richtungen geschehen und
 - Programme tieferer Ebenen können von Programmen höherer Ebenen profitieren, um ihre Funktion(en) zu erfüllen
- es macht jedoch Sinn, dass sich die Schutzhierarchie nach einer Programmhierarchie orientiert

³Hauptsteuerprogramm (*supervisor*) unten, Benutzerprogramm oben.



- *post* Multics: Schutzhierarchie auf Basis von Befähigungen
 - Hydra [22] → C.mmp, Cal [9] → CDC 6400; iAPX 432 [14]





- Schutzhierarchien haben sich (bisher) nicht weitläufig durchgesetzt
 - diese in „unstrukturierte“ Systeme nachträglich einbringen zu wollen, ist alles andere als einfach bzw. scheitert
 - eine Programmhierarchie als „Rückgrat“ kann dabei sehr förderlich sein



Einleitung

Hierarchische Struktur

Arten von Hierarchie

Programmhierarchie

Prozesshierarchie

Mittelvergabehierarchie

Schutzhierarchie

Funktionale Hierarchie

Benutzthierarchie

Hierarchiebildung

Zusammenfassung



[16] Benutzungshierarchie $\stackrel{[4, S. 151]}{\equiv}$ funktionale Hierarchie [7]

- für zwei Programme A und B bedeutet A „benutzt“ B , ...
 - wenn die korrekte Ausführung von B notwendig ist, damit A seine Arbeit entsprechend der Spezifikation vollenden kann
 - 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“
 - ↪ bedingte Aufrufe, z.B. der von KUZU (vgl. S. 7)
 2. Programm A kann B „benutzen“, obwohl es Programm B nie aufruft
 - ↪ asynchrone Programmunterbrechungen, Zuteilung realer Adressbereiche
- „benutzt“ \models „erfordert die Existenz einer korrekten Version von“



- 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 \leadsto vgl. S. 7, **Ausnahme werfen**
 - 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 Prozesse invariant halten**
 - d.h., dass diese *von der Hardware ausgelösten Routinen* terminieren, obwohl ein Aufruf an sie in keinem Programm kodiert ist
- ↪ lässt den Schluss zu, dass Unterbrechungsbehandlungsroutinen die unterste Ebene der Programmhierarchie ausmachen
- diese „benutzen“ jedoch (auch) die Verwaltung des realen Adressraums !



„benutzt“ \models „erfordert die Existenz einer korrekten Version von“

- „erfordert die Existenz“ \mapsto das Vorhandensein des Namens eines Artefaktes im Entwurf *oder* in der Implementierung
- „korrekte Version“ \mapsto 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!



- Grundregeln:
 1. Ebene 0 umfasst die Menge aller Programme, die kein anderes Programm (der Software) „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** [16, 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

- A ist wesentlich einfacher, da es B benutzt
 - B wurde bereitgestellt, um A zu unterstützen
- 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
 - der Fall ist ggf. Grund zur Neugestaltung oder Schichtanordnung
- 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
- es gibt keine vorstellbare Teilmenge, die A aber nicht B enthält
 - denn dann würde A die von B bereitgestellte Funktion nicht erfordern
 - beachte hier auch die *leere Implementierung*: `inline void B() {}`



Betriebssystemprogramme 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 [19]
- 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



- 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 in $OOStuBS_{BST}$
- Ebene₀ umfasst Programme verschiedener Abstraktionen
 - Teile von Funktionen zur Prozessor- und Adressraumverwaltung
 - unter diesen Programmen lässt sich keine Benutzrelation mehr finden



- Programme zur Unterbrechungsbehandlung, sofern vorhanden⁴
 - den **FLIH** (Abk. für *first-level interrupt handler*) „benutzen“ alle
 - er muss terminieren, damit der unterbrochene Prozess weiterlaufen kann
 - er muss den Prozessorzustand invariant halten, um Integrität zu wahren
 - ein ausgelöster, asynchron dazu laufender Nachspann ist eher unkritisch
 - z.B. ein Epilog [17] oder asynchroner Systemsprung (AST [10])
 - Programme zur Verwaltung des realen Adressraums
 - die **Zuordnung realer Adressen** an Prozessinkarnationen „benutzen“ alle
 - im Zuge der Vergabe, Freigabe oder des Entzugs von Hauptspeicher
 - das alles muss korrekt erfolgen, um systemweite Integrität zu wahren
 - ein Programm, das auch nicht von jedem anderen aufgerufen wird **!!!**
- ↪ **beachte:** gegenseitige Benutzung von Programmen
- Schichtanordnung (*sandwich*, [16, S. 5]) der Programme hilft nicht
 - für Programme der Ebene₀ gilt, dass sie sich ggf. gegenseitig „benutzen“

⁴Die geforderte Betriebsart bestimmt die Notwendigkeit von *Interrupts*!

Einleitung

- Hierarchische Struktur

Arten von Hierarchie

- Programmhierarchie

- Prozesshierarchie

- Mittelvergabehierarchie

- Schutzhierarchie

Funktionale Hierarchie

- Benutzthierarchie

- Hierarchiebildung

Zusammenfassung



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 unterscheidet sich von Aufrufbeziehung:
 - aufruf** \models erfordert die Existenz einer Version von
 \models nicht alles was „aufgerufen“ wird, wird auch „benutzt“
 - benutzt** \models erfordert die Existenz einer korrekten Version von
 \models nicht alles was „benutzt“ wird, wird auch „aufgerufen“
- die Rechnerbetriebsart gibt Hierarchiebildung von Betriebssystemen vor
- Schichtanordnung heißt nicht, dass alle Schichten immer ausgefüllt sind



- [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
- [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] LEFFLER, S. J. ; MCKUSICK, M. K. ; KARELS, M. J. ; QUARTERMAN, J. S.:
The Design and Implementation of the 4.3BSD UNIX Operating System.
Addison-Wesley, 1989. –
ISBN 0–201–06196–1



- [11] LIEDTKE, J. :
Towards Real Microkernels.
In: *Communications of the ACM* (1996), Sept., S. 70–77
- [12] 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)
- [13] ORGANICK, E. I.:
The Multics System: An Examination of its Structure.
MIT Press, 1972. –
ISBN 0-262-15012-3
- [14] 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



- [15] 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
- [16] PARNAS, D. L.:
Some Hypothesis About the "Uses" Hierarchy for Operating Systems / TH
Darmstadt, Fachbereich Informatik.
1976 (BSI 76/1). –
Forschungsbericht
- [17] SCHÖN, F. ; SCHRÖDER-PREIKSCHAT, W. ; SPINCZYK, O. ; SPINCZYK, U. :
On Interrupt-Transparent Synchronization in an Embedded Object-Oriented
Operating System.
In: LEE, I. (Hrsg.) ; KAISER, J. (Hrsg.) ; KIKUNO, T. (Hrsg.) ; SELIC, B. (Hrsg.):
*Proceedings of the 3rd IEEE International Symposium on Object-Oriented
Real-Time Distributed Computing (ISORC '00)*.
Washington, DC, USA : IEEE Computer Society, 2000, S. 270-277



- [18] SCHRÖDER, W. :
Eine Familie von UNIX-ähnlichen Betriebssystemen – Anwendung von Prozessen und des Nachrichtenübermittlungskonzeptes beim strukturierten Betriebssystementwurf, Technische Universität Berlin, Diss., Dez. 1986
- [19] SCHRÖDER-PREIKSCHAT, W. ; KLEINÖDER, J. :
Systemprogrammierung.
http://www4.informatik.uni-erlangen.de/Lehre/WS08/V_SP, 2008 ff.
- [20] SCHROEDER, M. D. ; SALTZER, J. H.:
A Hardware Architecture for Implementing Protection Ring.
In: [12], S. 42–54
- [21] VARNEY, R. C.:
Process Selection in a Hierarchical Operating System.
In: [12], S. 106–108
- [22] 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

