Betriebssystemtechnik

Hierarchische Struktur

3. Mai 2010

Überblick

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“ [1]

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

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.
```

- sei $p_i = FNUZ$ und $p_j = KUZA$, dann gilt für „benutzt“:

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

- daraus folgt [1]: *FNUZ* „benutzt“ *KUZA* nicht
  - Aufgabe von *FNUZ* ist es, *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 [4] bezeichnet, nicht funktionale Hierarchie.

Hierarchie sequentieller Prozesse (Forts.)

Relation „beauftragt“ legt eine Herarchie „stimmiger Kooperation“ [4] 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 [4]) ist dagegen eine Einschränkung auf das Laufzeitverhalten des Systems; darüberhinaus [1]:

> *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 [5] allein impliziert keine Prozesshierarchie!

- der Mikrokern kann Ebene $_0$ einer Programmhierarchie darstellen
- ggf. mit Prozesshierarchie ab Ebene $_{i,\, i>0}$: Thoth [6], AX [7]

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 [9]

Betriebsmittel werden dabei allerdings nicht immer direkt den Prozessen zugeteilt, 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, die ringartig organisiert sind, ersetzen das konventionelle zweistufige Modell[3] [10, 11] $\models$ Schutzhierarchie

- anfangs (mit Multics) nur rein in Software implementiert
  - auf Grundlage des zweistufigen Ansatzes (modifizierte GE 645)
- spätere Hardwarerealisierung (B 6000 [12]) $\rightsquigarrow$ 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 Programme höherer Ebenen benutzen, 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 6	Subsysteme	Anwendungssystem
5 4	Benutzerprogramme	
3 2	Subsysteme/Dienstprogramme	Betriebssystem
1 0	Hauptsteuerprogramm	

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

- Hydra [13] $\mapsto$ C.mmp, Cal [14] $\mapsto$ CDC 6400; iAPX 432 [15]

Hierarchie spezifischer Schutzzonen (Forts.)

Verwendung der Schutzringe (Intel 80386) in OS/2

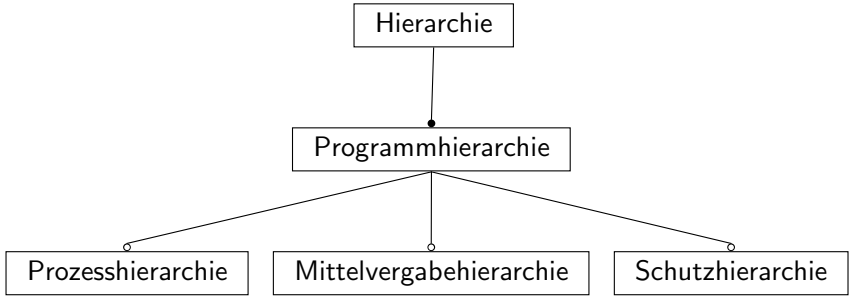
Ring	Funktion	Bereich
3	keine	ungenutzt
2	Anwendungsprozeduren	Benutzerprogramme
1	Ein-/Ausgabeprozeduren	
0	Kern und Gerätetreiber	Betriebssystem

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!!!

☞ entsprechend liegt der Fokus im weiteren Verlauf der Veranstaltung

Stufenweiser Maschinenentwurf

Ansatz: Programmhierarchie bzw. Hierarchie abstrakter Maschinen, wie z.B. mit THE [2], Venus [16] oder FAMOS [17] 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.*

  - tiefere (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 Analgie zur Hardware und ihren Programmen (Software)[4]

[4] Ein für diese Maschine geschriebenes Programm ist auf höherer Ebene, es kann oder kann nicht die Hardware korrekt nutzen. Verletzung der Ordnung ist unmöglich.

Hierarchiebildung basierend auf Funktionen

Strukturierung eines Systems in Ebenen sagt noch längst nichts aus über das Zusammenspiel der Ebenen untereinander[5]

- 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 $L_0, L_1, \ldots, L_n$ 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 Befehlssatzebene der Zielmaschine
  - jede Ebene stellt der nächst höheren Ebene neue „Hardware“ bereit

Der Systementwurf ist hierarchisch, nicht seine Implementierung [17]

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

[5] Siehe die verschiedenen, im vorigen Abschnitt behandelten Relationsarten.

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*, [18])

Beachte ↪ [18]
- eine Ebene kann in mehrere verschiedene Module aufgeteilt sein
- ein einzelnes Modul kann selektiv mehrere Ebenen überspannen
  - d.h. eine Schichtanordnung (engl. *sandwich*, [19]) bilden

Benutztbeziehung

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

[19] Benutzthierarchie $\overset{[20,\ S.\ 151]}{\equiv}$ funktionale Hierarchie [17]

- 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. 3-5)
  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. 3-5, 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 Routinen Unterbrechungen behandeln, obwohl ein Aufruf dazu an sie in keinem Programm kodiert ist

Beachte

zu 2. lässt den Schluss zu, dass Unterbrechungsbehandlungsroutinen die unterste Ebene der Programmhierarchie ausmachen
↪ 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 (vgl. auch S. 3-17):

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 ↪ [19, 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 (engl. *sandwich*)

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

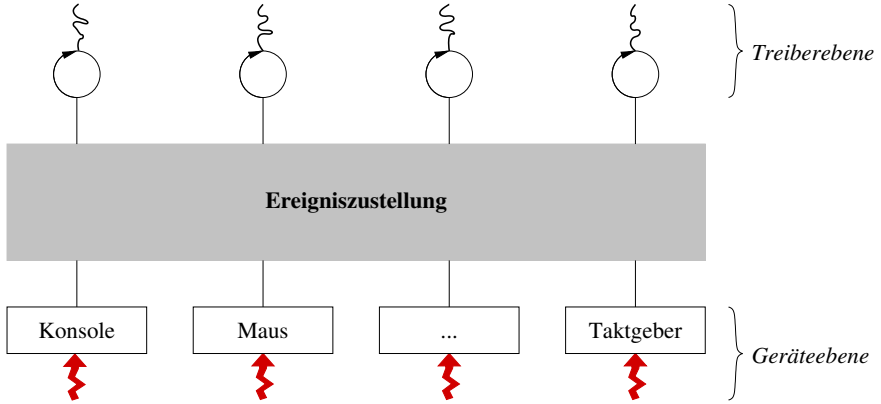
- typisch, falls Modul als "Ebene der Abstraktion" verstanden wird
  - vgl. auch S. 3-18: Modul $\neq$ Ebene
- Konflikt, der Neugestaltung der betroffenen Ebenen erfordert
  - lineare Ordnung durchsetzen $\rightsquigarrow$ Zyklus aufbrechen

Auflösung

Eines der Programme so in zwei Teile „schneiden", dass sich beide Programme „benutzen" können und dennoch die vier Bedingungen (S. 3-23) erfüllen

- $A \mapsto A_1, A_2 : A_1$ „benutzt" B „benutzt" A_2
- $B \mapsto B_1, B_2 : B_1$ „benutzt" A „benutzt" B_2

Zustellung von Gerätesignalen an Treiberfäden

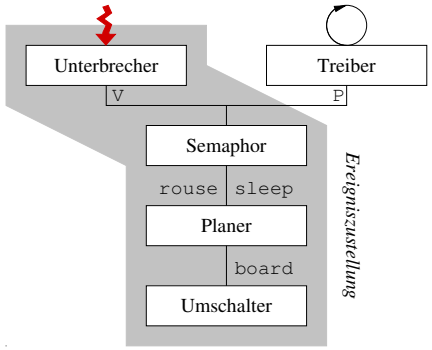


Gerätesignal $\mapsto$ konsumierbares Betriebsmittel

Treiberfaden $\mapsto$ Konsument von Signalen „seines" Gerätes

Ereigniszustellung $\mapsto$ Produzent/Verteiler von Gerätesignalen

Ereigniszustellung: Aufrufhierarchie



`P` Signal konsumieren

`V` Signal produzieren

`sleep` Faden auf Signal warten lassen

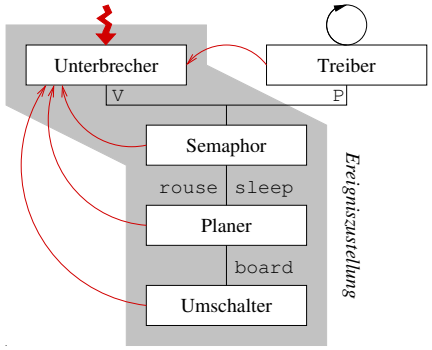
`rouse` auf Signal wartenden Faden bereitstellen

`board` Fadenwechsel durchsetzen

Beachte: Diese Struktur repräsentiert *keine* funktionale Hierarchie!

- Unterbrechungsbehandlung impliziert Abhängigkeiten, die in der hierarchischen Struktur nicht reflektiert worden sind
- hier: die Abhängigkeit von der Integrität des Prozessorzustands

Ereigniszustellung: Zyklus in der Benutztbeziehung



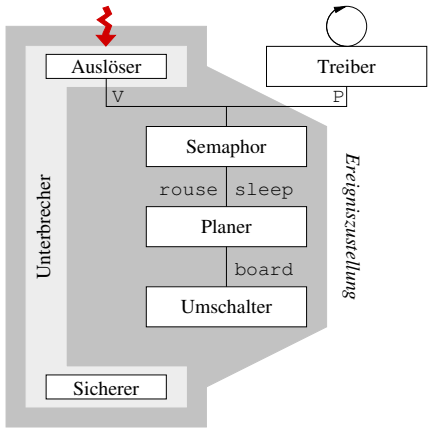
Beachte

Die korrekte Ausführung des Unterbrechers ist notwendig, damit Treiber, Semaphor, Planer und Umschalter korrekt funktionieren können!

Schichtanordnung des Unterbrechers

- der Unterbrecher wird aufgeteilt in zwei Funktionskomplexe:
  1. der Komplex zur Erzeugung des Signals (Aufruf von `V`)
  2. der Komplex zur Sicherstellung der Integrität des Prozessorzustands
- Funktion 1. „benutzt" Semaphor, der Rest „benutzt" Funktion 2.

Ereigniszustellung: Benutztbeziehung



Unterbrecher

Auslöser stellt Signale zu

Sicherer sichert Integrität zu

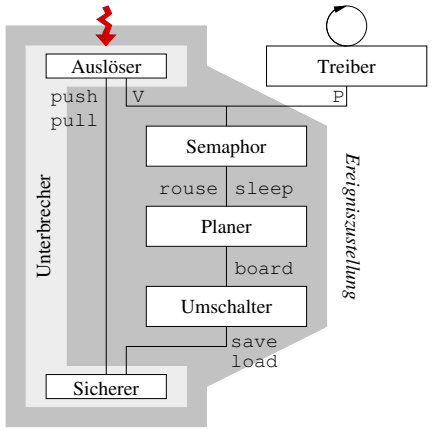
Beachte

- der Umschalter muss ebenfalls Integrität wahren
  - beim Fadenwechsel
  - Prozessorzustand sichern
- der Sicherer übernimmt die diesbezügliche Funktion

Frage

- Wie drückt sich die funktionale Beziehung zum Sicherer aus?

Ereigniszustellung: Funktionale Hierarchie



Sicherer

flüchtige Register

- `push` Sicherung
- `pull` Wiederherstellung

nichtflüchtige Register

- `save` Sicherung
- `load` Wiederherstellung

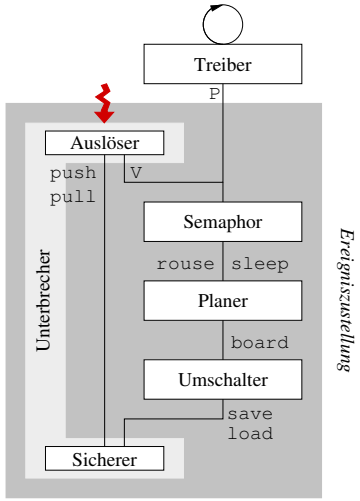
Implementierung

- ist tiefst prozessorabhängig
- muss ablaufinvariant sein

Beachte: „Benutzung“ von Semaphor impliziert Blockadefreiheit

- genauer: `V` *darf nicht* blockierend synchronisiert sein!

Ereigniszustellung: Funktionale Hierarchie (Forts.)



Treiber

- hängt von korrektem Auslöser ab
  - direkt $\mapsto$ Aufruf von `V`
    - Signalerzeugung
    - Prozessdeblockade
  - indirekt $\mapsto$ Implementierung von `V`
    - keine Prozessblockade
    - wartefrei synchronisieren
- ist dem Auslöser überzuordnen

Beachte: Treiber „benutzt“ Auslöser

- ohne ihn jedoch aufzurufen

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

- Ereigniszustellung: von Aufruf- zur funktionalen Hierarchie
- Schichtanordnung der Unterbrechungsbehandlung

Literaturverzeichnis

[1] David Lorge Parnas.
On a 'buzzword': Hierarchical structure.
In Jack L. Rosenfeld, editor, *Information Processing 74, Proceedings of the IFIP Congress 74, Stockholm, Sweden, August 5–10, 1974*, pages 336–339, New York, NY, USA, 1977. North-Holland Publishing Company.

[2] Edsger Wybe Dijkstra.
The structure of the THE-multiprogramming system.
Communications of the ACM, 11(5):341–346, May 1968.

Literaturverzeichnis (Forts.)

[3] Edsger Wybe Dijkstra.
Complexity controlled by hierarchical ordering of functions and variability.
In Peter Naur and Bruce Randell, editors, *Software Engineering, Report on the Conference of the NATO Science Committee, Garmisch, Germany, 7–11 October 1968*, pages 181–186, Brussels, Belgium, October 1969. Science Affairs Division NATO.

[4] Arie Nicolaas Habermann.
On the Harmonious Co-Operation of Abstract Machines.
PhD thesis, Technische Hogeschool Eindhoven, Eindhoven, The Netherlands, October 1967.

[5] Jochen Liedtke.
Torwards real microkernels.
Communications of the ACM, pages 70–77, September 1996.

Literaturverzeichnis (Forts.)

[6] David Ross Cheriton.
Multi-Process Structuring and the Thoth Operating System.
PhD thesis, University of Waterloo, Ontario, Canada, 1978.

[7] Wolfgang Schröder.
Eine Familie von UNIX-ähnlichen Betriebssystemen – Anwendung von Prozessen und des Nachrichtenübermittlungskonzeptes beim strukturierten Betriebssystementwurf.
Dissertation, Technische Universität Berlin, December 1986.

[8] Peer Brinch Hansen.
The nucleus of a multiprogramming system.
Communications of the ACM, 13(4):238–241/250, April 1970.

Literaturverzeichnis (Forts.)

[9] R. C. Varney.
Process selection in a hierarchical operating system.
In *Proceedings of the 3rd ACM Symposium on Operating Systems Principles (SOSP '71)* [21], pages 106–108.

[10] Robert M. Graham.
Protection in an information processing utility.
Communications of the ACM, 11(5):365–369, May 1968.

[11] Elliot I. Organick.
The Multics System: An Examination of its Structure.
MIT Press, 1972.

[12] Michael D. Schroeder and Jerome H. Saltzer.
A hardware architecture for implementing protection rings.
In *Proceedings of the 3rd ACM Symposium on Operating Systems Principles (SOSP '71)* [21], pages 42–54.

Literaturverzeichnis (Forts.)

[13] William Allan Wulf, Ellis S. Cohen, William M. Corwin, Anita K. Jones, Roy Levin, Charles Pierson, and Fred J. Pollack.
HYDRA: The kernel of a multiprocessor operating system.
Communications of the ACM, 17(6):337–345, June 1974.

[14] Butler W. Lampson and Howard E. Sturgis.
Reflections on an operating system design.
Communications of the ACM, 19(5):251–265, May 1976.

[15] Elliot I. Organick.
A programmer's view of the Intel 432 system.
McGraw-Hill, Inc., New York, NY, USA, 1983.

[16] Barbara H. Liskov.
The design of the Venus operating system.
In *Proceedings of the 3rd ACM Symposium on Operating Systems Principles (SOSP '71)* [21], pages 11–16.

Literaturverzeichnis (Forts.)

[17] Arie Nicolaas Habermann, Lawrence Flon, and Lee W. Cooprider.
Modularization and hierarchy in a family of operating systems.
Communications of the ACM, 19(5):266–272, 1976.

[18] David Lorge Parnas.
On the criteria to be used in decomposing systems into modules.
Communications of the ACM, pages 1053–1058, December 1972.

[19] David Lorge Parnas.
Some hypothesis about the "uses" hierarchy for operating systems.
Technical report, TH Darmstadt, Fachbereich Informatik, 1976.

[20] David Garlan, J. Frits Habermann, and David Notkin.
Nico Habermann's research: A brief retrospective.
In *Proceedings of the 16th International Conference on Software Engineering (ICSE '94), May 16–21, 1994, Sorrento, Italy*, pages 149–153, New York, NY, USA, 1994. ACM Press.

Literaturverzeichnis (Forts.)

[21] volume 6 of *ACM SIGOPS Operating Systems Review*, New York, NY, USA, June 1972. ACM Press.