Betriebssystemtechnik

Programmunterbrechungen: Ereigniszustellung

Wolfgang Schröder-Preikschat

Lehrstuhl Informatik 4

8. Mai 2012

Gliederung

1 Grundlagen
- Funktionale Hierarchie
- Benutzthierarchie

2 Systementwurf
- Szenario
- Signalumsetzer

3 Asynchroner Systemsprung
- Motivation
- Definition
- Konzept

4 Zusammenfassung

Stufenweiser Maschinenentwurf

Ansatz: Programmhierarchie 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.*

  - 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 Analogie zur Hardware und ihren Programmen (Software)

Hierarchiebildung basierend auf Funktionen

Strukturierung eines Systems in Ebenen sagt noch längst nichts aus über das Zusammenspiel der Ebenen untereinander[1]

- 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 [4]

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

[1] Das Kapitel „Hierarchische Struktur“ wird diesen Punkt später erneut aufgreifen.

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 Schichtanordnung (engl. *sandwich*, [8]) bilden

Benutztbeziehung

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

$$\text{[8] Benutzthierarchie} \overset{[3,\ S.\ 151]}{\equiv} \text{funktionale Hierarchie [4]}$$

- 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"
     ↪ beispielsweise der bedingte Aufruf einer seiteneffektfreien Prozedur
  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"

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. 5: Modul $\neq$ Ebene
- Konflikt, der Neugestaltung der betroffenen Ebenen erfordert
  - lineare Ordnung durchsetzen $\leadsto$ Zyklus aufbrechen

Auflösung

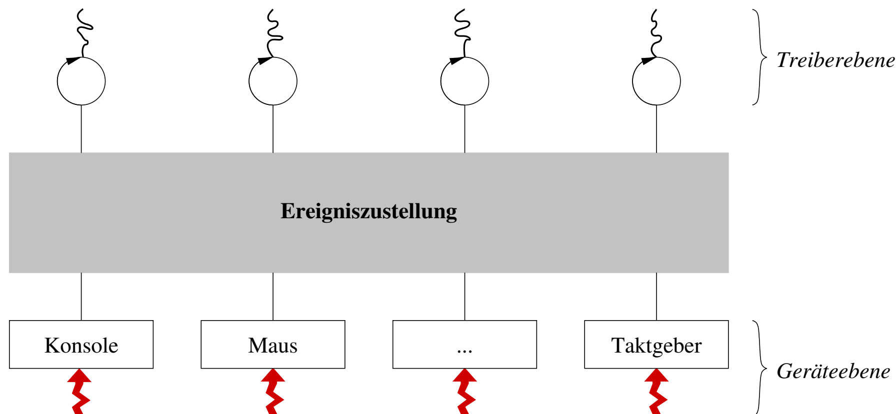
Eines der Programme so in zwei Teile „schneiden", dass sich beide Programme „benutzen" können:

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

Gliederung

1. Grundlagen
   - Funktionale Hierarchie
   - Benutzthierarchie
2. Systementwurf
   - Szenario
   - Signalumsetzer
3. Asynchroner Systemsprung
   - Motivation
   - Definition
   - Konzept
4. Zusammenfassung

Zustellung von Gerätesignalen an Treiberfäden

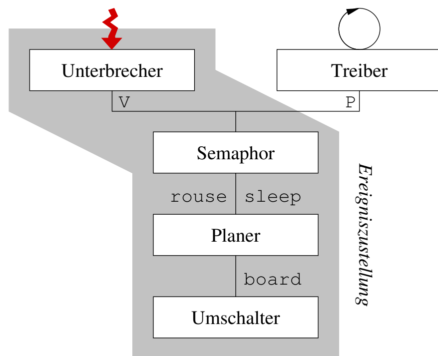


Gerätesignal $\mapsto$ konsumierbares Betriebsmittel

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

Ereigniszustellung $\mapsto$ Produzent/Verteiler von Gerätesignalen

Am Anfang: Skizze einer Aufrufhierarchie

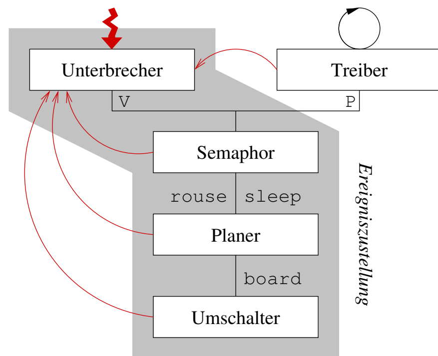


`P`	Signal konsumieren
`V`	Signal produzieren
`sleep`	Faden auf Signal warten lassen
`rouse`	auf Signal wartenden Faden bereitstellen
`board`	Fadenwechsel durchsetzen

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

Aufrufhierarchie $\neq$ Benutztbeziehung: Zyklen



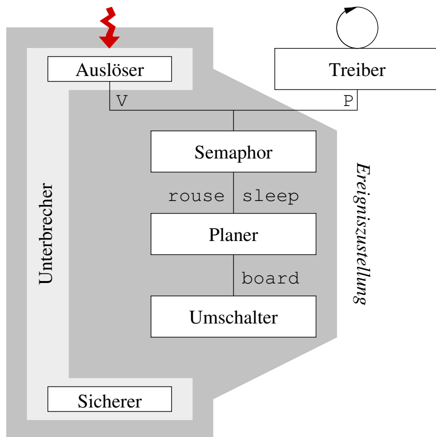
Beachte

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

Schichtanordnung des Unterbrechers

- der Unterbrecher wird aufgetelt 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.

Herleitung der Skizze einer Benutztbeziehung



Unterbrecher

Auslöser stellt Signale zu

Sicherer sichert Integrität zu

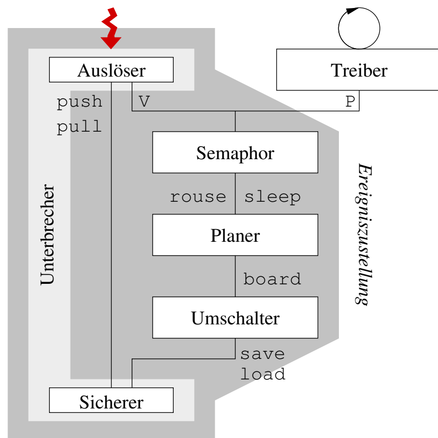
Beachte

- 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?

Verfeinerung zu einer funktionalen Hierarchie



Sicherer

flüchtige Register

- `push` Sicherung
- `pull` Wiederherstellung

nichtflüchtige Register

- `save` Sicherung
- `load` Wiederherstellung

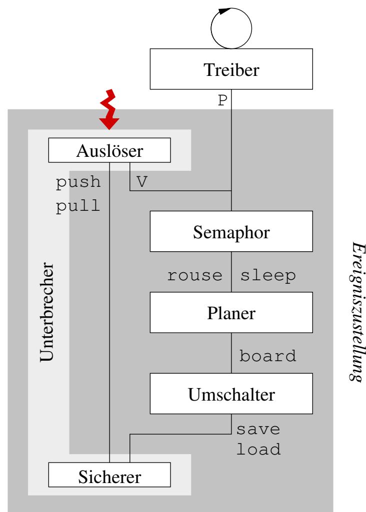
Implementierung

- ist tiefst prozessorabhängig
- muss ablaufinvariant sein

Beachte: „Benutzung“ von Semaphor impliziert Blockadefreiheit

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

Anordnung der Funktionen in einer Benutzthierarchie



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

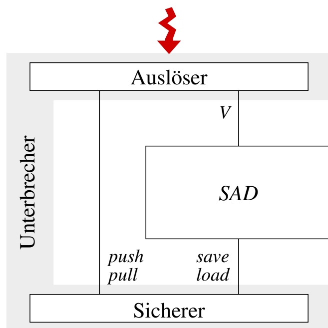
Beachte: Treiber „benutzt“ Auslöser

- ohne ihn jedoch aufzurufen

Gliederung

1. Grundlagen
   - Funktionale Hierarchie
   - Benutzthierarchie
2. Systementwurf
   - Szenario
   - Signalumsetzer
3. Asynchroner Systemsprung
   - Motivation
   - Definition
   - Konzept
4. Zusammenfassung

Hardwaresignal $\mapsto$ Softwaresignal



Aufgabe des Auslösers:

1. Annahme und Quittierung einer Unterbrechungsanforderung
2. Generierung eines Gerätesignals: `V`

Beachte

1. läuft auf $IPL \geq 0$
2. sollte/muss auf $IPL = 0$ laufen

Anforderungen

zu 1. Laufzeitminimierung der Unterbrechungssperre ($IPL > 0$)

- konstruktiv, durch Zweiteilung der Behandlungsroutine

zu 2. sichere Unterbrechungsentsperrung: $IPL > 0 \mapsto IPL = 0$

- Emulation eines asynchronen Systemsprungs

Kontextfreier (ablaufinvarianter) Systemaufruf

AST (Abk. engl. *asynchronous system trap*, [1])
Mechanismus, der einen Prozess vor Rückkehr zum Benutzermodus erneut in den Systemmodus zwingt[2]

- zur nachträglichen Abarbeitung im Systemmodus ausgelöster, jedoch vorerst zurückgestellter Systemaufträge
- als wenn diese Aufträge per Systemaufruf ausgeführt werden

> *On the VAX, a software-initiated interrupt to a service routine. ASTs enable a process to be notified of the occurrence of a specific event asynchronously with respect to its execution. In 4.3 BSD, ASTs are used to initiate process rescheduling. [5]*

[2] Im Zusammenhang mit asynchronen Programmunterbrechungen ist der Moment der Rückkehr in den Benutzermodus genaugenommen bereits etwas zu spät, wenn die Unterbrechung nämlich den Systemmodus betraf. Als frühester Zeitpunkt ergibt sich, wenn keine Inkarnation einer asynchronen Unterbrechungen mehr aktiv ist.

Implementierungsansätze

Hardware

- bei Rückkehr aus der Unterbrechungsbehandlung prüft die CPU, ob ein AST möglich/erlaubt ist
  möglich $\mapsto$ AST-Bitschalter $= 1$ $\hookrightarrow$ Prozessorstatus
  erlaubt $\mapsto$ Rückkehr zum Benutzermodus

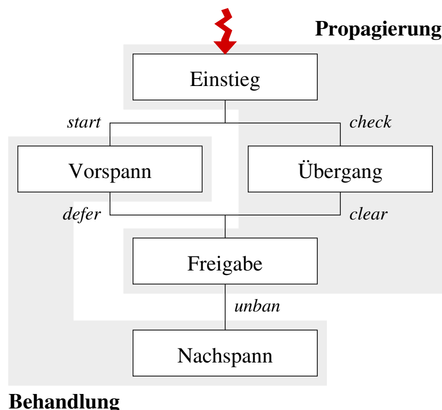
Hard-/Software

- das Betriebssystem weist dem AST die niedrigste Prioritätsebene zu: $0 \leq IPL_{off} < IPL_{ast} < IPL_{max}$
- bei Rückkehr aus der Unterbrechungsbehandlung prüft die CPU, ob ein AST möglich/erlaubt ist
  möglich $\mapsto$ IPL_{ast} aktiv geschaltet
  erlaubt $\mapsto$ Abstieg auf IPL_{ast}

Software

- CPU/Betriebssystem verbuchen Inkarnationen von Unterbrechungsbehandlungen
- bei Rückkehr aus der Unterbrechungsbehandlung prüft das BS, ob ein AST möglich/erlaubt ist
  möglich $\mapsto$ AST-Warteschlange $\neq \emptyset$
  erlaubt $\mapsto$ genau eine (akt.) Inkarnation verbucht

Entwurfsskizze zur Nachbildung eines AST in Software



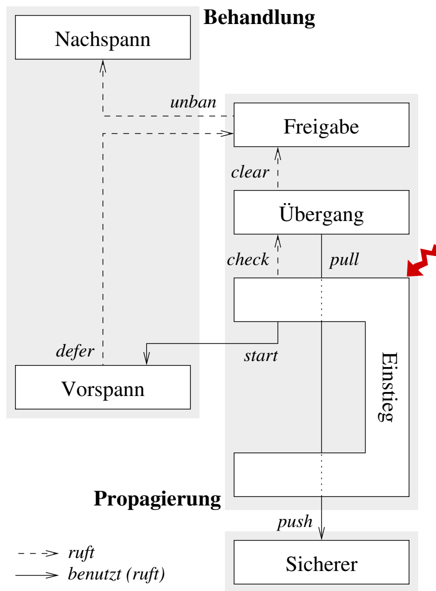
Einstieg	Inkarnationsebene verbuchen
Vorspann	erster Abschnitt der Behandlungsroutine
Übergang	Inkarnationsebene prüfen
Freigabe	AST-Warteschlange auf- bzw. abbauen
Nachspann	zweiter Abschnitt der Behandlungsroutine

Beachte: Einstieg startet auf $IPL > 0$, Nachspann läuft auf $IPL = 0$

Flankensteuerung $\mapsto$ sichere Entsperrung bereits im Einstieg möglich

Pegelsteuerung $\mapsto$ sichere Entsperrung erst im Übergang möglich

Repräsentation des Entwurfs in einer Benutzthierarchie



Nachspann	erfordert erfolgreiche Propagierung des AST
Freigabe	erfordert die Entsperrung von Unterbrechungen
Übergang	erfordert die Verbuchung der Inkarnationsebene
Einstieg	erfordert die Rückkehr aus dem Vorspann
Vorspann	erfordert den Einstieg in die Propagierung

Gliederung

1. Grundlagen
   - Funktionale Hierarchie
   - Benutzthierarchie
2. Systementwurf
   - Szenario
   - Signalumsetzer
3. Asynchroner Systemsprung
   - Motivation
   - Definition
   - Konzept
4. Zusammenfassung

Resümee

Grundlagen $\hookleftarrow$ funktionale Hierarchie, Benutzthierarchie

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

Systementwurf $\hookleftarrow$ Signalumsetzer

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

asynchroner Systemsprung $\hookleftarrow$ kontextfreier Systemaufruf

- Umwandlung eines Hardwaresignals in ein Softwaresignal
- Implementierungsansätze: Hardware, Hard-/Software, Software
- Benutzthierarchie des Softwareansatzes

Literaturverzeichnis

[1] Digital Equipment Corporation (Hrsg.):
VAX-11 Architecture Reference Manual.
Maynard, MA, USA: Digital Equipment Corporation, Mai 1982. –
Document Number EK-VAXAR-RM-001

[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] 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

[4] 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

[5] Leffler, S. J. ; McKusick, M. K. ; Karels, M. J. ; Quarterman, J. S.:
The Design and Implementation of the 4.3 BSD UNIX Operating System.
Addison-Wesley, 1989. –
ISBN 0–201–06196–1

Literaturverzeichnis (Forts.)

[6] Liskov, B. J. H.:
The Design of the Venus Operating Systeme.
In: 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), S. 11–16

[7] Parnas, D. L.:
On the Criteria to be used in Decomposing Systems into Modules.
In: *Communications of the ACM* 15 (1972), Dez., Nr. 12, S. 1053–1058

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