Betriebssystemtechnik

Unterbrechungen

17. Mai 2010

Überblick

Unterbrechungen
- Einleitung
- Hardwareaspekte
- Softwareaspekte
- Propagierung
- Parallelverarbeitung
- Zusammenfassung
- Bibliographie

Motiv: Einbindung peripherer Prozesse

„Unterbrechungsgesteuerte Systeme sind nicht einfach" (J. Nolte, PEACE, um 1990)

Bestandsaufnahme: über 70 % der Betriebssystemabstürze gehen auf Kosten fehlerhafter Gerätetreiber [1]

Protokollverletzungen sind die Ursache allen Übels:
- 38 % bei Interaktion mit dem Betriebssystem
- 20 % bei Zugriff auf gemeinsame Betriebsmittel
- 19 % bei Interaktion mit den Geräten

Nebenläufigkeit in den Griff zu bekommen, ist nach wie vor *die* Herausforderung im Betriebssystembau
- beginnt mit der Unterbrechungsbehandlung
- setzt sich fort in Parallelprozessoren

Lernziel
- Trennung der Belange (engl. *separation of concerns*) bei Entwurf und Entwicklung von Programmen zur Ereignisverarbeitung

Einordnung

Schicht	Funktion	Konzepte
12	Programmverwaltung	Text, Daten, Überlagerung
11	Dateiverwaltung	Dateisystem; Verzeichnis, Verknüpfung
10	Prozessverwaltung	Aktivitätsträger, Kontext, Stapel
9	Adressraumverwaltung	Arbeitsspeicher, Segment, Seite
8	Informationsaustausch	Paket, Nachricht, Kanal, Portal
7	Geräteprogrammierung	Kern; Signal, Zeichen, Block, Datenstrom
6	Platzanweisung	Hauptspeicher, Fragment, Seitenrahmen
5	Zugriffskontrolle	Subjekt, Objekt, Domäne, Befähigung
4	Betriebsmittelzugriff	Verdrängungs-/Vorgangssperre
3	Auftragseinplanung	Ereignis, Priorität, Zeitscheibe, Energie
2	Ablaufsteuerung	Unterbrechungs-/Fortsetzungssperre, Wettlauftoleranz
1	Kontrollflusswechsel	Koroutine, Unterbrechung, Fortsetzung
0	Stammprozessorabstraktion	Stammsystem
−1	Peripherie	MMU, (A)PIC, DMA, UART, ATA, SCSI, USB, ...
−2	Zentraleinheit	ARM, AVR, PowerPC, SPARC, x86, ...

Rekapitulation: SOS 1 [3] bzw. SP [4], BS [5]

Programmunterbrechungen kommen, in Abhängigkeit von ihrer Ursache und in Bezug auf den aktuellen Kontrollfluss, in zwei Ausführungen vor:

synchron (engl. *trap*, dt. *Falle*)
- die Unterbrechungsstelle im Programm ist vorhersagbar
- die Unterbrechung des Prozesses ist reproduzierbar

asynchron (engl. *interrupt*, dt. *Unterbrechung*)
- weder vorhersag- noch reproduzierbar: $\neg$ synchron
- Wettlaufsituationen sind auszuschließen/zu tolerieren
  - Unterbrechungssperre bzw. ✓
  - nichtblockierende Synchronisation oder ✗
  - unterbrechungstransparente Synchronisation ✗

Beachte: Analogie zu Konzepten der Ausnahmebehandlung [2]

Ein *Interrupt* muss nach dem Wiederaufnahmemodell, ein *Trap* kann auch nach dem Beendigungsmodell behandelt werden.

Erkennung asynchroner Programmunterbrechungen

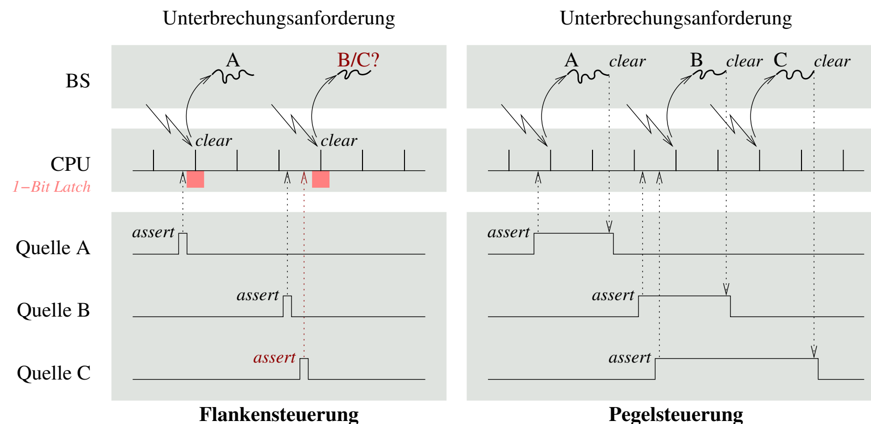
Untersuchungsgegenstand: die Unterbrechungsanforderungsleitung (engl. *interrupt request line*) bzw. ihr jew. Zustand am Eingang der CPU

flankengesteuert (engl. *edge triggered*) $\mapsto$ Taktflanke
- Erkennung der Unterbrechungsanforderung in zwei Phasen:
  1. Pegelwechsel[1] im Auffangregister (engl. *latch*) zwischenspeichern
  2. auf Zustandsänderung am Ende des laufenden Befehls prüfen
- ggf. implizites Löschen der Quelle nach erkannter Anforderung
  - eine Funktion der Hardware: Protokoll CPU $\leftrightarrow$ Gerät/PIC

pegelgesteuert (engl. *level triggered*) $\mapsto$ Logikpegel
- zyklische Zustandsabfrage der Unterbrechungsleitung (engl. *pin*)
  - taktweise oder am Ende des laufenden Befehls
  - Unterbrechung erfolgt je nach definierter Logikebene (0 oder 1)
- explizites Löschen der Quelle bei der Unterbrechungsbehandlung
  - eine Funktion der Software: Protokoll Gerätetreiber $\leftrightarrow$ Gerät

[1] Von logisch „0“ (engl. *low*) zu logisch „1“ (engl. *high*) oder umgekehrt.

Gemeinsame Benutzung derselben Unterbrechungsleitung



Problemfälle

Flankensteuerung	Wiederbehauptung, Unterbrechungssperre
Pegelsteuerung	Nichtbehandlung

Flanken- vs. Pegelsteuerung

Wiederbehauptung (engl. *reassertion*) einer Unterbrechung über eine mehreren Quellen gemeinsame Unterbrechungsanforderungsleitung
- vgl. Beispiel auf S. 5-6

Flankensteuerung
- Unterbrechungsanforderungen bleiben ggf. unerkannt
  - die Mitbenutzung derselben Unterbrechungsanforderungsleitung durch mehrere Geräte muss dem Gerätetreiber bewusst sein
  - er muss alle Geräte an dieser Leitung immer abfragen — und darf am Ende keine (weitere) Unterbrechungsanforderung verlieren
- unerkannte Quellen werden keine neuen Unterbrechungen fordern
  - ein Aufhänger (engl. *hangup*) im System kann die Folge sein

Pegelsteuerung
- Unterbrechungen werden erkannt, wenngleich auch verzögert

Flanken- vs. Pegelsteuerung (Forts.)

Nichtbehandlung einer (niedrig priorisierten) Unterbrechung über eine mehreren Quellen gemeinsame Unterbrechungsanforderungleitung

- Problem der „Quellendistanz“ zur CPU bzw. Abfragereihenfolge
  - Abfrage durch die CPU erfolgt typicherweise von „nah“ zu „fern“
- FPFS („*first polled, first served*“) $\models$ FCFS $\overset{?}{\leadsto}$ Prioritätsverletzung

Flankensteuerung

- Unterbrechungsanforderungen werden — *wahrscheinlich* — erkannt
  - da die Unterbrechungsleitung immer nur kurz aktiv gehalten wird
  - problematisch können jedoch gleichzeitige Signalisierungen sein

Pegelsteuerung

- Blockade höher priorisierter („ferner“) Unterbrechungen ist möglich
  - falls die CPU Anforderungen „naher“ Quellen (niedrigerer Priorität) konstruktions-/fehlerbedingt nicht bedienen kann
- nicht erkannte Anforderungen halten die gemeinsame Leitung aktiv

Flanken- vs. Pegelsteuerung (Forts.)

Unterbrechungssperre d.h. zeitweises Abwehren von Unterbechungen

Flankensteuerung

- Unterbrechungsanforderungen bleiben ggf. unerkannt
  1. durch Unterbrechungssperren der Hardware
     - jede CPU startet die Unterbrechungsbehandlung auf einer bestimmten Unterbrechungsprioritätsebene (engl. *interrupt priority level*, IPL)
     - Unterbrechungen mit Prioritäten $P \leq IPL_{act}$ sind solange gesperrt

$$IPL = \begin{cases} 0, 1, 2, \ldots, N & \text{kaskadierbare Unterbrechungsanforderung} \\ 0, 1 & \text{sonst} \end{cases}$$

  2. durch Unterbrechungssperren der Software: kritische Abschnitte
- Unterbrechungsereignisse gehen möglicherweise verloren

Pegelsteuerung

- Unterbrechungen werden erkannt, Ereignisse gehen nicht verloren

Flanken- vs. Pegelsteuerung (Forts.)

Anwendungsfälle geben vor, welche Art der Unterbrechungssteuerung in einem Rechensystem umgesetzt sein sollte

Universalsystem $\leadsto$ egal

- Textverarbeitung, Programmentwicklung, ..., Unterhaltung

Spezialsystem $\leadsto$ nicht egal $\hookrightarrow$ Pegelsteuerung

- Prozesssteuerung, d.h. die Steuerung *externer* Prozesse

Beachte

> *Wird von einem Rechensystem erwartet, Echtzeitfähigkeit, Zuverlässigkeit oder Robustheit zu garantieren, so sollten Unterbrechungsanforderungen pegelgesteuert sein!*

- die Unterschiede zwischen Flanken- und Pegelsteuerung treten an vielen Stellen in der Betriebssystemsoftware zum Vorschein
- sie machen querschneidende Belange in der Systemsoftware aus und sind nur schwer zu modularisieren $\leadsto$ AOP [6]

Arten von Unterbrechungen

maskierbare Unterbrechung (engl. *maskable interrupt*)

- auch: Unterbrechungsanforderung (engl. *interrupt request*, IRQ)
- Ab-/Anschalten möglich durch Spezialbefehle der CPU
  - privilegierte Befehle, je nach CPU
- die CPU dadurch anweisen, einen IRQ zu ignorieren/beachten
  - den Unterbrecher in der CPU steuern

nicht-maskierbare Unterbrechung (engl. *non-maskable interrupt*, NMI)

- Ab-/Anschalten ggf. möglich durch Funktionen des Gerätetreibers
  - sofern das Gerät die Programmierung der Unterbrechung zulässt
- so ggf. die Signalisierung der Unterbrechung unterbinden können
  - ein an der CPU angeschlossenes Gerät steuern

Ansatzpunkte zur Abwehr von Unterbrechungen

CPU immer möglich ↪ IRQ
- Annahme von Unterbrechungsanforderungen verweigern
  - an der *Senke* ansetzen
- Elementaroperation der Befehlssatzebene

PIC bedingt möglich ↪ IRQ
- Weiterleitung von Unterbrechungsanforderungen unterbinden
  - am ggf. vorhandenen *Mittler* ansetzen
- Programm der Befehlssatzebene: Gerätetreiber

Gerät immer möglich ↪ NMI/IRQ
- Auslösen von Unterbrechungsanforderungen abstellen
  - an der *Quelle* ansetzen
- Programm der Befehlssatzebene: Gerätetreiber

Annahme von Unterbrechungsanforderungen verweigern

... und wieder zulassen: CPU mit $N(IPL)$ gleich $2^1 = 2$ (links) bzw. $2^3 = 8$ (rechts)

x86

```
INLINE void irq_avert () {
    asm("cli");
}

INLINE void irq_admit () {
    asm("sti");
}
```

m68k

```
INLINE void irq_avert () {
    asm("or #$0700, sr");
}

INLINE void irq_admit () {
    asm("and #$f8ff, sr");
}
```

Beachte: Mögliche Ereignisverluste bei Flankensteuerung
- wenn in der Sperrphase Unterbrechungsanforderungen eingehen *und* diese in der Hardware nicht gespeichert werden können
- die Speicherkapazität der Auffangregister ist stark begrenzt: ein Bit!

Auslösung der Unterbrechungsbehandlungsroutine

ungerichtete Unterbrechung (engl. *non-vectored interrupt*) ↪ RISC
- die CPU nimmt die Ausführung eines Abfrageprogramms auf
  - das dann die Hardware nach der Art der Unterbrechung abfragt ↪ Unterbrechungsabfrage (engl. *polled interrupt*)
  - um zur gewünschten Behandlungsroutine verzweigen zu können
- das Abfrageprogramm hat eine fest vorgegebene Anfangsadresse

gerichtete Unterbrechung (engl. *vectored interrupt*) ↪ CISC
- die CPU nimmt die Ausführung einer Behandlungsroutine auf
  - jede Unterbrechungsart erhält eine solche Routine zugewiesen ↪ Ausnahmevektor (engl. *exception vector*)
  - alle Ausnahmevektoren sind in einer Vektortabelle zusammengefasst
- die Vektortabelle hat eine feste oder variable Anfangsadresse

Anmerkung

Häufig sind im Falle eines RISC beide Konzepte anzufinden, indem das Abfrageprogramm eine gerichtete Unterbrechung *emuliert*.

Typen von Ausnahmevektoren

Programmabschnitt variabler aber maximaler Länge
- wird im Regelfall statisch ausgerichtet

Anfangsadresse einer Behandlungsroutine
- kann statisch oder dynamisch bestimmt werden

Deskriptor eines Behandlungsobjektes
- kann statisch oder dynamisch aufgesetzt werden
- beschreibt die Umgebung eines passiven oder aktiven Objekts
  - mit ggf. eigenem Adressraum und Laufzeitkontext

Beachte

Instanzenbildung ist möglich zur Programmier-, Übersetzungs-, Binde-, Lade- oder Ausführungszeit eines Betriebssystems — wenngleich auch mit typbedingten Schwierigkeiten.

Mehrstufige Unterbrechungen

Voraussetzung: kaskadierbare Unterbrechungsanforderungen direkt unterstützt durch die CPU

- typischerweise hat die CPU hierzu mehr als eine IRQ-Leitung
  - z.B. m68k-Familie: drei Leitungen $\Rightarrow 2^3 = 8$ IRQ-Ebenen
  - die IRQ-Ebene entspricht einer Unterbrechungsprioritätsebene
- nur Unterbrechungen mit Prioritäten $P > IPL$ werden zugelassen

Beachte

- bei nur einer IRQ-Leitung müsste ein komplexes Protokoll greifen
  - die IRQ-Ebene, nicht die -Nummer, geht über den Daten-/Adressbus
  - x86-Familie: der PIC schickt eine IRQ-Nummer, *keine* -Ebene
- ohne diese Fähigkeiten ist Kaskadierung nur ansatzweise möglich:
  1. ein Gerät steuert die Durchleitung von Unterbrechungsanforderungen
     - Funktion des PIC
  2. ein Programm der Befehlssatzebene lässt sofort Unterbrechungen zu
     - Funktion der Unterbrechungsbehandlungsroutine

Kaskadierte Unterbrechung

Programme zur Unterbrechungsbehandlung sind grundsätzlich mit der Tatsache konfrontiert, selbst unterbrochen werden zu können[2]

- wenn Behandlungsroutinen virtuelle Adressen verwenden $\mapsto$ *Trap*
  - ggf. bei E/A oder der Behandlung von Segment-/Seitenfehlern
- wenn höher priorisierte Ereignisse auftreten $\mapsto$ *Interrupt*
  - ein NMI unterbricht einen NMI oder einen IRQ
  - ein IRQ höherer Priorität unterbricht einen IRQ niedrigerer Priorität

Unterbrechungslatenzen erhöhen sich ggf. nicht deterministisch

- der Aufwand multipliziert sich mit der Anzahl der Prioritätsebenen
  - Parameter der Hardware/Software des Rechensystems ✓
- kritisch: Frequenz/Zeitspanne von Unterbrechungsanforderungen
  - Parameter der Umgebung des Rechensystems ?

[2] Programmierfehler nicht betrachtet.

Unterbrechungslatenz

Latenz (von lat. *latens*, verborgen) steht für Latenzzeit und bezeichnet den Zeitraum zwischen einem verborgenen Ereignis und dem Eintreten einer sichtbaren Reaktion darauf [7]

verborgenes Ereignis ist die Auslösung der Unterbrechungsanforderung
- hervorgerufen durch ein Gerät (Quelle)

sichtbare Reaktion ist die Aufnahme einer Programmausführung
- bewerkstelligt durch die CPU (Senke)

Beachte

Unterbrechungslatenz betrifft zwei eng zusammenhängende Bereiche:

1. Hardware: Operationsprinzip der CPU (inkl. PIC, ggf.)
   - Abfragemodell zur Annahme von Unterbrechungsanforderungen
   - Auslösemoment der Unterbrechung ab Annahme der Anforderung
2. Software: Operationsprinzip von Gerätetreiber/Betriebssystem(kern)
   - Dauer, Abstand und Häufigkeit von *Elementaroperationen*
     - erzwungen durch Unterbrechungssperren bzw. gegeben von der CPU

Latenzverbergung (engl. *latency hiding*)

Konzept zur Entlastung einer CPU u.a. von unterbrechungsbedingten Aktivitäten im Betriebssystem oder Maschinenprogramm

- vom Prinzip her ein spezialisiertes Satellitenrechnerkonzept
- je nach Anwendungsfall wünschenswert[3] oder gar notwendig[4]

Unterbrechungsanforderungen gehen an einen Koprozessor („Satellit"), der die Programme zur Unterbrechungsbehandlung ausführt

- je nach Fähigkeiten der Hardware und des Betriebssystems ist die Koprozessorzuordnung statisch oder dynamisch

  statisch $\mapsto$ Prozessorverwaltung im Betriebssystem
  dynamisch $\mapsto$ APIC
- Rechnerarchitekturmodell: asymmetrisches Mehrprozessorsystem
  - Asymmetrie reflektiert sich im Betriebssystem: $BS_{cop} \subset BS_{\neg cop}$
  - paralleles Betriebssystem: BS_{cop} greift in Abläufe von $BS_{\neg cop}$ ein u.u.

[3] z.B. Hochleistungsrechnen [8]
[4] z.B. Echtzeitverarbeitung [9, 10]

Latenzminimierung

Aufgabe einer jeden Unterbrechungsbehandlungsroutine !!!

Faustregeln:

1. Verkürzung der Unterbrechungsbehandlungsdauer ✓
   - Laufzeitoptimierung der in dem Kontext relevanten Systemsoftware, zur Programmier- oder Kompilierzeit
   - ggf. auch, als letzten Ausweg, Rückgriff auf Assembliersprache
2. Aufteilung der Unterbrechungsbehandlungsroutine ✗
   - in ein erstes Unterprogramm, dessen Ausführung zum Zeitpunkt der Unterbrechung aufgenommen werden soll
   - in ein zweites Unterprogramm, dessen Ausführung erst zu einem späteren Zeitpunkt sicherzustellen ist
3. Vermeidung von Unterbrechungssperren im Betriebssystem ✗
   - ideal ist unterbrechungstransparente Systemsoftware [11]

Vorsicht

4. Entsperrung von Unterbrechungen in den Behandlungsroutinen ✗
   - es muss unbedingt der „richtige Zeitpunkt“ abgepasst werden

Latenzminimierung (Forts.)

zu 1. die Unterbrechungsbehandlung läuft auf $IPL_{act} > 0$ (vgl. S. 5-9)
- Unterbrechungen mit $IPL \leq IPL_{act}$ sind solange gesperrt
- je schneller die Behandlung durch ist, umso kürzer die Latenz

zu 2. eine Maßnahme zur weiteren Verkürzung der impliziten Sperrzeit
- nur die erste Phase der Behandlung läuft auf $IPL_{act} > 0$
- die zweite Phase läuft auf $IPL_{act} = 0$ (voll unterbrechbar)
- Entwurfsziel: die erste Phase so kurz wie möglich gestalten

zu 3. die Unterbrechungslatenz hängt ausschließlich von der Hardware ab

zu 4. eine Fehlerquelle, die Systemstillstand/-abstürze verursachen kann
- (Pegelsteuerung) Unterbrechungsanforderungen niedriger IPL müssen zuvor beim Gerät quittiert worden sein
  - Entsperrung $\Longleftrightarrow$ kein Pegel liegt mehr an !
- (Flanken-/Pegelsteuerung) der unbestimmten Ausdehnung des Stapelspeichers muss vorgebeugt werden
  - Entsperrung $\Longleftrightarrow$ softwarebedingter Stapelplatz ist abgebaut !

Ereignisverluste und Pseudoereignisse

Beachte

Unterbrechungslatenzen sind hardwarebedingt ($IPL > 0$), ihre effektive Länge hängt jedoch vielmehr von der Software ab:
- die Zeiträume für Unterbrechungsbehandlung und -sperren

Flankensteuerung kann Ereignisverluste bedingen
- je länger die Unterbrechungslatenz...
  - egal ob Hard- oder Softwarevorgänge den Zeitraum ausmachen
- desto wahrscheinlicher der Ereignisverlust

Pegelsteuerung kann Pseudoereignisse hervorbringen
- bei zu frühzeitiger Entsperrung von Unterbrechungen...
  - liegt der Pegel noch an, kommt es sofort wieder zur Unterbrechung
- sind Pseudoereignisse mehr als nur wahrscheinlich
  - eine softwarebedingte „unechte Unterbrechung“ ist die Folge[5]

[5] engl. *spurious interrupt*, durch Fremdeinfluss verursachte falsche Unterbrechung.

Quittierung einer Unterbrechungsanforderung

Bestätigung (engl. *acknowledgement*) des Empfangs eines Signals zur Programmunterbrechung beim Absender: dem Gerät
- auch Abnahmequittung (engl. *acceptor handshake*)
  - erzeugt/versandt durch Hardware oder Software (Gerätetreiber)
- Regelfall: Gerätetreiber quittiert die Anforderung beim Gerät[6]
  - bei Pegelsteuerung nimmt das Gerät erst dann das Signal zurück
- stellt sicher, dass ein Kommunikationswunsch sein Ziel erreichte [12]

Beachte

Sollte eine Unterbrechungsbehandlung in den zuvor beschriebenen zwei Phasen (zur Latenzminimierung) verlaufen, dann muss die Unterbrechungsanforderung in der ersten Phase quittiert werden.

[6] Je nach Gerät muss hierzu entweder explizit ein Bestätigungskommando in ein Geräteregister geschrieben werden oder es genügt ein normaler Lese-/Schreibzugriff auf eines der Geräteregister, um die Quittung zu generieren.

Integritätswahrung unterbrochener Progammabläufe

Integrität (lat. *integritas*, Echt-/Unversehrtheit, Lauter-/Redlichkeit)
Datenrichtigkeit, Datensicherheit, Fehlerfreiheit, Vollständigkeit

- Prozessorstatus unterbrochener „Prozesse“ ist invariant zu halten

  Sicherung bei Annahme einer Unterbrechungsanforderung
  - Arbeitsteilung CPU, BS, Kompilierer
  - d.h. (vgl. auch [3, 4]): Ebene $_{2,3,5}$

  Wiederherstellung bei Abschluss der Unterbrechungsbehandlung
- Programmabläufe bleiben determiniert, nicht deterministisch **!!!**

Beachte: Determiniertheit und Determinismus

1. bei ein und derselben Eingabe sind verschiedene Abläufe zulässig, alle Abläufe liefern jedoch stets das gleiche Resultat
2. zu jedem Zeitpunkt ist bestimmt, wie weitergefahren wird

Softwarebelange querschneidender Art

Fälle im Problemraum, die einer Modularisierung im Lösungsraum oft entgegenstehen bzw. diese erschweren

Auslösemoment $\hookrightarrow$ Flanke, Pegel
- Latenz, Treiberfunktion, Synchronisation

Unterbrechungslatenz $\hookrightarrow$ Verbergung, Minimierung
- Operationsprinzip (BS), Treiberstruktur

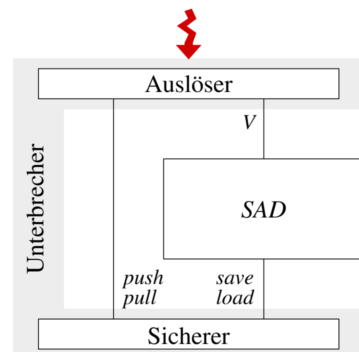
Integritätswahrung $\hookrightarrow$ CPU, BS, Kompilierer
- Prozessorstatus je nach Anwendung/Ebene

Unterbrechungsabwehr $\hookrightarrow$ CPU, Gerät, ggf. auch PIC
- an Quelle, Senke oder (ggf.) Mittler ansetzen

Beachte

Theoretisch lässt sich im Lösungsraum für jeden möglichen Fall eine Funktion definieren, die passend konfigurierbar ausgelegt ist. Jedoch ist dieser Ansatz wegen der hohen Entwurfskomplexität nicht immer praktikabel. $\rightsquigarrow$ AOP [6]

Ereigniszustellung: wieder aufgenommen...



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

Asynchroner Systemsprung

AST (Abk. engl. *asynchronous system trap*, [13])
Mechanismus, der einen Prozess vor Rückkehr zum Benutzermodus erneut in den Systemmodus zwingt[7]
- 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. [14]

[7] 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.

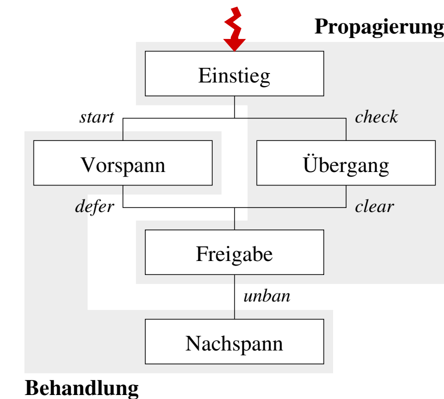
Asynchroner Systemsprung: Implementierungsansätze

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 $\mapsto$ Rückkehr zum „Benutzermodus“

Asynchroner Systemsprung: Entwurfsskizze

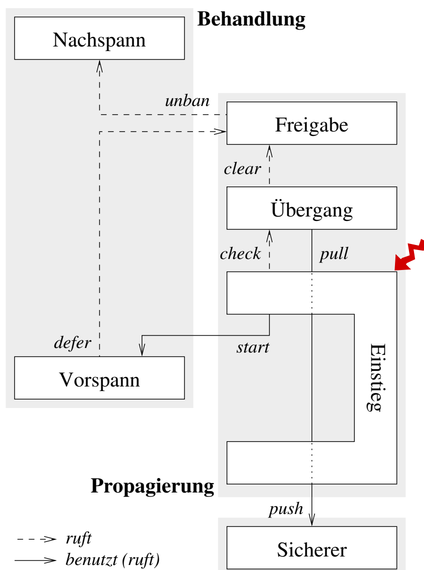


- **Einstieg** Inkarnationsebene verbuchen
- **Vorspann** erster Abschnitt der Behandlungsroutine
- **Übergang** Inkarnationsebene prüfen
- **Freigabe** AST-Warteschlange auf- bzw. abbauen
- **Nachspann** zweiter Abschnitt der Behandlungsroutine

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

Asynchroner Systemsprung: 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

Unterbrechungsbehandlung: Gewaltenteilung

Vorspann $\hookrightarrow$ FLIH (Abk. für engl. *first-level interrupt handler*)
- ist bedingt unterbrechbar:
  - Flankensteuerung $\iff IPL_{irq} \geq IPL_{act}$
  - Pegelsteuerung $\iff IPL_{irq} > IPL_{act}$
- erledigt nur die dringend nötigen Geräteaufgaben:
  - Flankensteuerung $\Longrightarrow$ ggf. nichts
  - Pegelsteuerung $\Longrightarrow$ Abnahmequittung zustellen
- löst bei Bedarf einen Nachspann aus

Nachspann $\hookrightarrow$ SLIH (Abk. für engl. *second-level interrupt handler*)
- ist grundsätzlich unterbrechbar: $IPL = 0$
- erledigt ausstehende/angefallene Geräteaufgaben

Beachte: Nachspann $\neq$ Epilog [5]
- läuft zwar mit $IPL = 0$, jedoch asynchron im Betriebssystemkontext
- darf weiterhin nur wiedereintrittsinvariante Funktionen verwenden!

Weitergabe von Unterbrechungen *de*LUXE

TIP (Abk. engl. *trap/interrupt propagation*)

```
#include "luxe/queue.h"
#include "luxe/remit.h"

typedef remit_t tip_t;              /* deferrable SLIH abstraction */

extern void tip_entry ();           /* come here upon trap/interrupt */
extern void tip_start ();           /* start of FLIH, also: prototye */
extern void tip_defer (tip_t *);    /* provision of SLIH */
extern void tip_unban (tip_t *);    /* release of deferred SLIH */
extern void tip_check ();           /* propagate SLIH, if allowed */
extern void tip_clear ();           /* propagate SLIH, if any */

extern queue_t *tip_store ();       /* return pointer to SLIH queue */
```


Einstieg in die Unterbrechungsweitergabe *de*LUXE

Pegelsteuerung

```
__GLOBAL(tip_entry):
    TIP_PUSH                      /* save relevant CPU state */
    call __GLOBAL(tip_start)      /* invoke associated FLIH */
                                  /* fall into SLIH check... */
```

Flankensteuerung

```
__GLOBAL(tip_entry):
    incl tip_level                /* register this interrupt */
    sti                           /* approve interruptibility */
    TIP_PUSH                      /* save relevant CPU state */
    call __GLOBAL(tip_start)      /* invoke associated FLIH */
                                  /* fall into SLIH check... */
```

Beachte

Verschiedene Ausnahmevektoren verweisen auf verschiedene Einstiegsroutinen, die dann mit `jmp __GLOBAL(tip_check)` enden.

Hilfsmittel der Assembliersprachenebene *de*LUXE

Externe Symbolnamen

```
#ifdef __luxe_leading_underscore__
#define __GLOBAL(name) _ ## name
#else
#define __GLOBAL(name) name
#endif
```

Beachte die traditionelle Weise

Einige Kompilierer für C erzeugen Symbolnamen mit einem führenden Unterstrich (engl. *underscore*). So etwa gcc für früheres Linux sowie für Win32 und die BSD-Familie.

Sicherer: Flüchtige Register

```
#ifndef TIP_PUSH
#define TIP_PUSH \
    pushl %eax; \
    pushl %ecx; \
    pushl %edx
#endif

#ifndef TIP_PULL
#define TIP_PULL \
    popl %edx; \
    popl %ecx; \
    popl %eax
#endif
```


Übergang zur Unterbrechungsweitergabe (1) *de*LUXE

Pegelsteuerung

```
__GLOBAL(tip_check):
    cmpl $0, __GLOBAL(tip_queue)  /* any pending SLIH? */
    je 1f                         /* no, done */
    cmpb $0, tip_level            /* TIP critical section active? */
    jne 1f                        /* yes, do not reenter! */
    movb $1, tip_level            /* enter TIP critical section */
2:  sti                           /* approve interruptibility */
    call __GLOBAL(tip_clear)      /* propagate pending SLIH */
    cli                           /* prevent interruptibility */
    cmpl $0, __GLOBAL(tip_queue)  /* any SLIH pending again? */
    jne 2b                        /* yes, clear */
    movb $0, tip_level            /* leave TIP critical section */
1:  TIP_PULL                      /* restore relevant CPU state */
    iret
```

Beachte: Wettlaufsituation

- es muss sichergestellt sein, dass kein Systemsprung vergessen wurde

Übergang zur Unterbrechungsweitergabe (2) *de*LUXE

Flankensteuerung

```
__GLOBAL(tip_check):
    cmpl $1, tip_level          /* at bottom of interrupt stack? */
    jne 1f                      /* no, do not propagate! */
2:  cli                         /* prevent interruptibility */
    cmpl $0, __GLOBAL(tip_queue)  /* any pending SLIH? */
    je 1f                       /* no, done */
    sti                         /* re-approve interruptibility */
    call __GLOBAL(tip_clear)    /* propagate pending SLIH */
    jmp 2b                      /* re-check for sporadic SLIH  */
1:  TIP_PULL                    /* restore relevant CPU state */
    cli                         /* prevent interruptibility */
    decl tip_level              /* deregister this interrupt */
    iret
```

Beachte: Wettlaufsituation

- wie zuvor (S. 5-35) darf auch hier kein Systemsprung verloren gehen
  - kritisch: Abfrage der Warteschlange und Auswertung der Abfrage

Richtiger Zeitpunkt der Unterbrechungsweitergabe

Durchsetzung eines (emulierten) asynchronen Systemsprungs erfordert, dass keine Inkarnation einer Unterbrechungsbehandlung mehr aktiv ist

deLUXE

```
.lcomm tip_level,1,0
```

- je nach Steuerungsart ein Bitschalter oder Inkarnationszähler

- richtiger Zeitpunkt $\Longleftrightarrow$ $level = \begin{cases} 0 & \text{bei Pegelsteuerung} \\ 1 & \text{bei Flankensteuerung} \end{cases}$

Beachte

- in der Situation ist kein Vorspann mehr aktiv, auch kein Nachspann
  - nur der gesicherte Zustand einer Inkarnation liegt auf dem Stapel
  - dieser würde auch bei einem echten AST wieder oben rauf kommen
- die Stapelausdehnung bleibt trotz Unterbrechungsfreigabe begrenzt
  - festgelegt durch die Anzahl unterstützter Unterbrechungsebenen
  - hinzu kommt noch Stapelbedarf des „tiefsten" Nachspanns

Richtiger Zeitpunkt der Unterbrechungsweitergabe (Forts.)

Wettlaufsituation

```
__GLOBAL(tip_entry):
    incl tip_level
    sti
```

. . . im Falle kaskadierbarer Unterbrechungen!!!

- bei Unterbrechung vor `incl tip_level`
- auch hier (x86 und ggf. PIC): NMI

Lösungsansätze:

- NMI ausnehmen, d.h. nicht auf asynchronen Systemsprung abbilden
- anstatt Zählen, spezielle Fähigkeiten der CPU nutzen, z.B.:
  - wenn der 1. Befehl noch garantiert ausgeführt wird (TMS 9900)
  - wenn das Statusregister Informationen zum Arbeitsmodus enthält
    - im Unterbrechungsfall wurde der Inhalt auf dem Stapel gesichert
    - mit Informationen aus dem Zeitraum direkt *vor* der Unterbrechung
    - richtiger Zeitpunkt zum Systemsprung: voriger Modus war „Benutzer"
    - Überprüfung: indirekt indizierte Adressierung mittels Stapelzeiger

Beachte

Wenn der NMI nicht ausgenommen werden soll, dann muss eine Lösung allein auf Basis der CPU gefunden werden!

Unterbrechungsfreigabe

Aufhebung der Unterbrechungssperre, wenn dem Konzept entsprechend eine unbestimmte Ausdehnung des Stapels nicht möglich ist[8]

- *zuerst* Unterbrechungen zulassen, um sie *danach* wieder abzuwehren
- ist nicht mit einer Unterbrechungssperre zu verwechseln
  - die umgekehrt funktioniert, zum Schutz kritischer Abschnitte

Beachte

- Unterbrechungsabwehr bringt das System nur in den Normalzustand zurück, der nämlich dafür sorgt, dass
  1. kein zurückgestellter Systemsprung vergessen wird und
  2. der Stapel nicht unbestimmt wachsen kann
     - bei Unterbrechung zwischen `decl tip_level` und `iret` (S. 5-36)

[8] Nicht dem Konzept entsprechend ist die Verwendung von Einstieg und Übergang zur Unterbrechungsfreigabe bei Flankensteuerung für den pegelgesteuerten Betrieb. Umgekehrt genauso, unbestimmtes Anwachsen des Stapels geht dann jedoch nicht.

Freigabe zurückgestellter Systemsprünge *de*LUXE

Nachspann zurückstellen

```
INLINE void tip_defer (tip_t *item) {
    assert(item);
    its_aback(tip_store(), (chain_t*)&item->next);
}
```

Nachspann freistellen

```
INLINE void tip_unban (tip_t *item) {
    (item->work)();
}

void tip_clear () {
    tip_t *next;
    do if ((next = (tip_t*)its_fetch(tip_store()))) tip_unban(next);
    while (next != 0);
}
```

- ITS steht für (Abk. engl.) *interrupt transparent synchronization*

Abstraktion zurückgestellter Systemsprünge *de*LUXE

Nachspann ↦ Prozeduradresse in einer Kette

```
#include "luxe/chain.h"

struct remit {
    chain_t next;       /* next remit in sequence, if any */
    void (*work)();     /* deferred procedure */
};

typedef struct remit remit_t;
```

- ein Auftrag (engl. *remit*), um einen Systemsprung zu leisten
- zur Buchführung/Abrechnung bieten sich Spezialisierungen an
  - z.B. Verbuchung der pro Nachspann verbrauchten Systemzeit

Beachte

- Zurückstellung eines Nachspanns trifft eine Planungsentscheidung
- diese muss nicht zwingend auf FCFS hinauslaufen...

Wiederbehauptung asynchroner Systemsprünge

Wettlaufsituation im Falle der Zurückstellung eines Nachspanns, der sich noch auf der Nachspannwarteschlange befindet

- erneute Einreihung in die Warteschlange ist zu unterbinden
  - **Vorbeugung** heißt, die Unterbrechungsfrequenz kann die WCET[9] *aller* möglichen Nachspänne nicht unterschreiten **?**
  - **Vermeidung** zieht algorithmische Lösungen in Erwägung ✗
- wobei das Nachspannereignis ggf. nicht verloren gehen darf

Lösungsansätze

- eine Freiliste (engl. *free list*) von Auftragsdeskriptoren verwalten
  - *defer* leert und *clear* füllt diese Liste in Kooperation
  - ist die Liste leer, gehen Nachspannereignisse verloren
- im Auftragsdeskriptor einen Ereigniszähler führen/auswerten ✗
  - *defer* erhöht und *clear* erniedrigt den Zähler in Kooperation
  - bei Zählerüberlauf, gehen Nachspannereignisse verloren

[9] Abk. engl. *worst case execution time*

Wiederbehauptung asynchroner Systemsprünge (Forts.)

Ernüchterung

- Propagierung schließt Ereignisverlust nicht ganz aus
  - kann bestenfalls unwahrscheinlicher gemacht werden
- betrifft Gerätetreiber mehr oder weniger stark
  - lässt sich nur End-zu-End [12] entscheiden
- ggf. werden zus. Vor-/Nachspannprotokolle benötigt

TIP Spezialisierung: *safe interrupt propagation*, SIP *de*LUXE

```
struct apart {          /* NOTE: this is a tip_t variant */
    remit_t item;       /* remit abstraction controlled */
    char    busy;       /* processing state */
};
```

```
void sip_defer (tip_t *item) {
    assert(item);
    if (!ami_apply(&item->busy))
        tip_defer(item);
}
```

```
void sip_unban (tip_t *item) {
    item->busy = 0;
    tip_unban(item);
}
```

apply ⊨ atomares *test and set*, TAS

Lastverteilung bei der Unterbrechungsbehandlung

Mehrprozessortechnik gibt dem Betriebssystem Möglichkeiten zu einer Entlastung der Recheneinheiten von der Unterbrechungsbehandlung

Idee *de* LUXE

```
__GLOBAL(tip_entry):
    TIP_PUSH
    call __GLOBAL(tip_start)
    TIP_PULL
    iret
```

- jeder Prozessor verarbeitet nur noch den für ihn bestimmten Vorspann
- die Verarbeitung des Nachspanns übernimmt ein anderer Prozessor

- neben *entry* zeichnen sich Änderungen in *defer* und *clear* ab: sie sind
  1. multiprozessorsicher auszulegen und müssen
  2. prozessorübergreifenden Abfrage- oder Anzeigebetrieb unterstützen
- als Modelle bieten sich an: Fließband, Zusteller und Zustellergruppe

Beachte

- Mehrprozessortechnik umfasst (natürlich) mehrkernige Prozessoren

Lastverteilung bei der Unterbrechungsbehandlung (Forts.)

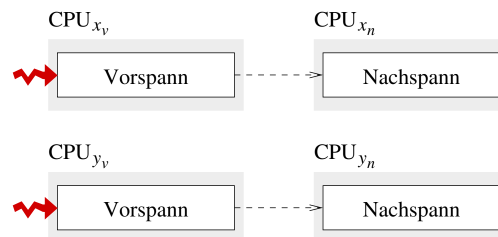
Abfragebetrieb
- aktives Warten (engl. *busy waiting*)
- *clear* stellt den nächsten verfügbaren Nachspann frei, überprüft dabei anhaltend die Warteschlange
- *defer* reiht einen Nachspann in die Warteschlange ein

Anzeigebetrieb
- passives Warten
- *clear* stellt den nächsten verfügbaren Nachspann frei und suspendiert seine CPU bei leerer Warteschlange
- *defer* reiht einen Nachspann in die Warteschlange ein und signalisiert ggf. die beauftragte CPU

Beachte

- welche CPU im Anzeigebetrieb zu signalisieren ist, hängt ganz vom Modell der Lastverteilung ab:
  - statisch im Falle von Fließband und Zusteller, für gewöhnlich
  - dynamisch im Falle von Zustellergruppe

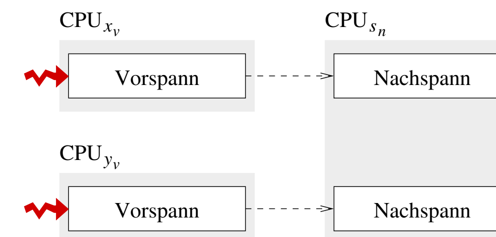
Fließband



Prinzip
- CPU_{x_v} & CPU_{y_v} sind Vorspannstufen
- CPU_{x_n} & CPU_{y_n} sind feste Nachspannstufen

- jede Nachspannstufe führt die Nachspänne ihrer Vorspannstufe aus
  - zeitversetzt, parallel zu nachfolgenden Unterbrechungen/Vorspännen
- 1:1-Wettlaufsituation der gemeinsamen Nachspannwarteschlange
  - *defer* ausgeführt auf CPU_{x_v} konkurriert mit *clear* auf CPU_{x_n}
  - entsprechendes Muster der Kooperation gilt für CPU_{y_v} und CPU_{y_n}
- Nachspannverarbeitung geringer Durchlaufzeit steht im Vordergrund
  - nicht aber Maximierung der Auslastung von CPU_{x_n} & CPU_{y_n}

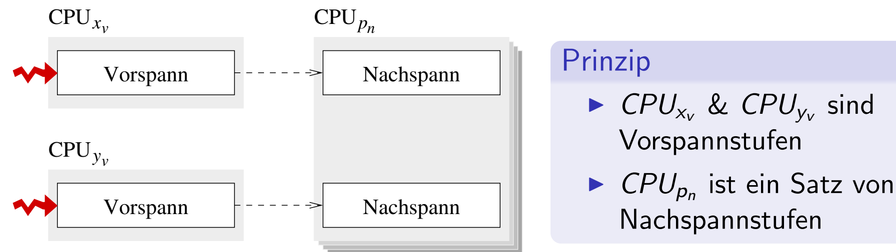
Zusteller



Prinzip
- CPU_{x_v} & CPU_{y_v} sind Vorspannstufen
- CPU_{s_n} ist zentrale, feste Nachspannstufe

- eine Nachspannstufe führt die Nachspänne aller Vorspannstufen aus
  - CPU_{x_v} & CPU_{y_v} sind Klienten, CPU_{s_n} ist Zusteller (engl. *server*)
- N:1-Wettlaufsituation der gemeinsamen Nachspannwarteschlange
  - *defer* auf CPU_{x_v} & CPU_{y_v} konkurrieren miteinander
  - *clear* auf CPU_{s_n} konkurriert mit *defer* auf CPU_{x_v} & CPU_{y_v}
- Auslastungsmaximierung von CPU_{s_n} steht im Vordergrund
  - weniger eine geringe Durchlaufzeit der Nachspannverarbeitung

Zustellergruppe



Prinzip

- CPU_{x_v} & CPU_{y_v} sind Vorspannstufen
- CPU_{p_n} ist ein Satz von Nachspannstufen

- $M > 1$ Nachspannstufen führen Nachspänne der Vorspannstufen aus
  - CPU_{x_v} & CPU_{y_v} Klienten, CPU_{p_n} Zustellergruppe (engl. *server pool*)
- N:M-Wettlaufsituation der gemeinsamen Nachspannwarteschlange
  - *defer* auf CPU_{x_v} & CPU_{y_v} konkurrieren miteinander
  - *clear* auf CPU_{p_n} konkurrieren miteinander
  - *clear* auf CPU_{p_n} konkurriert mit *defer* auf CPU_{x_v} & CPU_{y_v}
- Auslastungsmaximierung & Durchlaufzeitminimierung als Ziel

Asymmetrisches Mehrprozessorsystem

Koprozessoren entlasten Prozessoren von der Unterbrechungsbehandlung

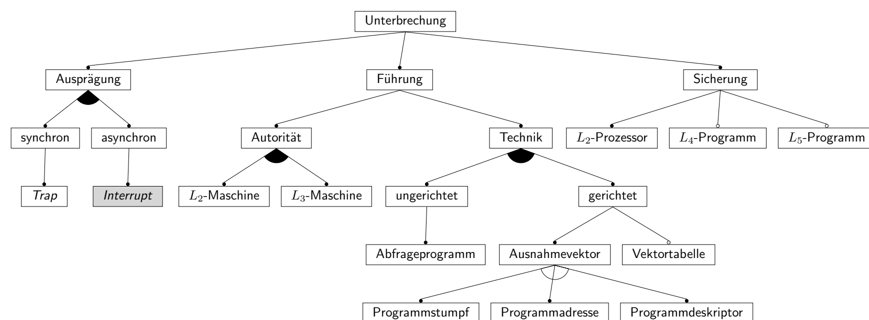
- CPU_{x_n}, CPU_{y_n}, CPU_{s_n} und CPU_{p_n} wären solche Koprozessoren
- nur diese sind für die Nachspannverarbeitung verantwortlich
  - sie erledigen die „schwere“ Arbeit zur Unterbrechungsbehandlung
  - sie werden auch evtl. Fortsetzungen ins Betriebssystem vorantreiben
- die anderen, CPU_{x_v} & CPU_{y_v}, machen nur Vorspannverarbeitung
  - sie erledigen die „leichte“ Arbeit zur Unterbrechungsbehandlung
- das BS ist *funktional dediziert verteilt* über die Prozessoren

Beachte

Vollständige Entlastung eines Prozessors sogar von der Verarbeitung eines Vorspanns der Unterbrechungsbehandlung setzt spezielle Hardware voraus, die Unterbrechungsanforderungen (statisch oder dynamisch) an andere Prozessoren weiter- bzw. umleitet.

- APIC, Abk. für engl. *advanced programmable interrupt controller*

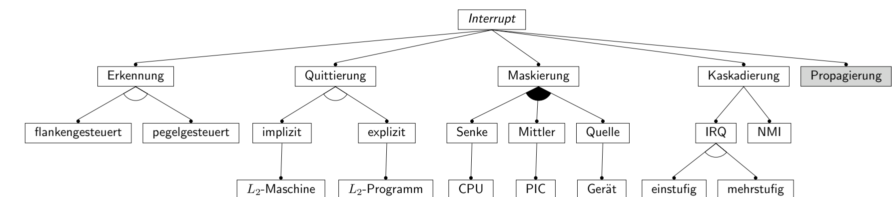
Merkmaldiagramm



L_2-Maschine	▶ CPU und ggf. PIC
L_3-Maschine	▶ Emulation gerichteter Unterbrechungen im BS
L_2-Prozessor	▶ Sicherung des min. Prozessorstatus' durch die CPU
L_4-Programm	▶ Sicherung der flüchtigen Register
L_5-Programm	▶ Sicherung der nicht-flüchtigen Register

Merkmaldiagramm (Forts.)

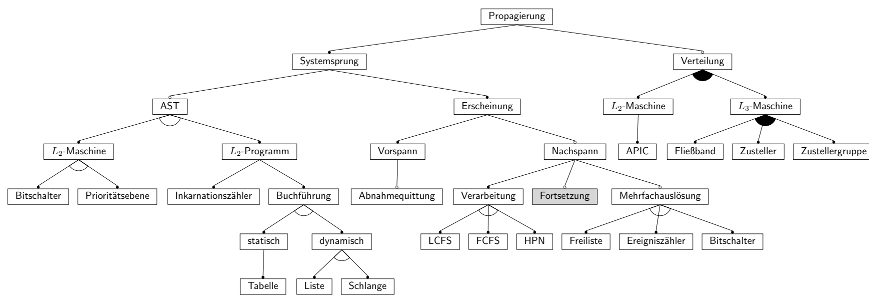
Asynchrone Programmunterbrechung



L_2-Maschine	▶ Kombination von CPU, PIC und Gerät
L_2-Programm	▶ FLIH, Vorspann eines asynchronen Systemsprungs

Merkmaldiagramm (Forts.)

Unterbrechungsweitergabe



Fortsetzung folgt. . .

- Synchronisierung mit den Abläufen im Betriebssystem
  - ↪ Epilog [5], wobei ein Vorspann den Prolog repräsentiert

Resümee

Hardwareaspekte ←→ Erkennung und Arten von Unterbrechungen

- Flanken-/Pegelsteuerung, Maskierung, Führung
- Ausnahmevektoren, Mehrstufigkeit

Softwareaspekte ←→ Unterbrechungslatenz

- Latenzverbergung/-minimierung, Ereignisverluste
- Pseudoereignisse, Quittierung, Integritätswahrung

Propagierung ←→ asynchroner Systemsprung, Vor-/Nachspann

- Weiter-/Freigabe von Unterbrechungen
- Freigabe und Wiederbehauptung von Systemsprüngen

Parallelverarbeitung ←→ Mehrprozessortechnik zur Lastverteilung

- Abfrage- oder Anzeigebetrieb
- Fließband, einzelner Zusteller oder Zustellergruppe

Literaturverzeichnis

[1] Leonid Ryzhyk, Peter Chubb, Ihor Kuz, and Gernot Heiser.
Dingo: Taming device drivers.
In *Proceedings of the fourth ACM European Conference on Computer Systems (EuroSys'09), Nuremberg, Germany, March 31–April 3, 2009*, pages 275–288, New York, NY, USA, April 2009. ACM Press.

[2] John Bannister Goodenough.
Exception handling: Issues and a proposed notation.
Communications of the ACM, 18(12):683–696, 1975.

[3] Wolfgang Schröder-Preikschat.
Softwaresysteme 1.
http://www4.informatik.uni-erlangen.de/Lehre/SOS1, 2004.

Literaturverzeichnis (Forts.)

[4] Wolfgang Schröder-Preikschat.
Systemprogrammierung.
http://www4.informatik.uni-erlangen.de/Lehre/SP, 2008.

[5] Daniel Lohmann.
Betriebssysteme.
http://www4.informatik.uni-erlangen.de/Lehre/BS, 2007.

[6] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin.
Aspect-oriented programming.
In Mehmet Aksit and Satoshi Matsuoka, editors, *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP '97)*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer-Verlag, June 1997.

Literaturverzeichnis (Forts.)

[7] Wikipedia Foundation Inc.
Wikipedia, Die freie Enzyklopädie.
http://de.wikipedia.org.

[8] Ulrich Brüning, Wolfgang K. Giloi, and Wolfgang Schröder-Preikschat.
Latency hiding in message-passing architectures.
In *Proceedings of the 8th International Symposium on Parallel Processing (IPPS '94)*, pages 704–709, Washington, DC, USA, 1994. IEEE Computer Society Press.

Literaturverzeichnis (Forts.)

[9] Luis E. Leyva del Foyo, Pedro Mejia-Alvarez, and Dionisio de Niz.
Predictable interrupt management for real time kernels over conventional PC hardware.
In *Proceedings of the 12th IEEE International Symposium on Real-Time and Embedded Technology and Applications (RTAS '06)*, pages 14–23, Los Alamitos, CA, USA, 2006. IEEE Computer Society Press.

[10] Luis E. Leyva del Foyo, Pedro Mejia-Alvarez, and Dionisio de Niz.
Predictable interrupt scheduling with low overhead for real-time kernels.
In *Proceedings of the 12th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA '06)*, pages 385–394, Washington, DC, USA, 2006. IEEE Computer Society Press.

Literaturverzeichnis (Forts.)

[11] Friedrich Schön, Wolfgang Schröder-Preikschat, Olaf Spinczyk, and Ute Spinczyk.
On interrupt-transparent synchronization in an embedded object-oriented operating system.
In *Proceedings of the 3rd IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC '00)*, pages 270–277. IEEE Computer Society Press, March 2000.

[12] Jerome H. Saltzer, David P. Reed, and David D. Clark.
End-to-end arguments in system design.
ACM Transaction on Computer Systems, 2(4):277–288, November 1984.

Literaturverzeichnis (Forts.)

[13] Digital Equipment Corporation.
VAX-11 Architecture Reference Manual.
Digital Equipment Corporation, Maynard, MA, USA, May 1982. Document Number EK-VAXAR-RM-001.

[14] Samuel J. Leffler, Marshall Kirk McKusick, Michael J. Karels, and John S. Quarterman.
The Design and Implementation of the 4.3 BSD UNIX Operating System.
Addison-Wesley, May 1989.