

# Betriebssystemtechnik

## Programmunterbrechungen: Prinzipien

Wolfgang Schröder-Preikschat

Lehrstuhl Informatik 4

24. April 2012

# Gliederung

## 1 Rekapitulation

## 2 Hardwareaspekte

- Signalisierungstechnik
- Unterbrechungssteuerung
- Ausnahmebehandlung

## 3 Softwareaspekte

- Unterbrechungslatenz
- Fehlhandlung
- Kontextsicherung
- Modularisierbarkeit

## 4 Zusammenfassung

# Einbindung peripherer Prozesse

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

Bestandsaufnahme: über 70 % der Betriebssystemabstürze gehen auf Kosten fehlerhafter Gerätetreiber [7]

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

# Stoffrelevante Schichten im Gesamtbild

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	<b>Geräteprogrammierung</b>	Kern; Signal, Zeichen, Block, Datenstrom
6	Elementplatzierung	Hauptspeicher; Seitenrahmen, Segment/Fragment
5	Zugriffskontrolle	Subjekt, Objekt, Domäne, Befähigung
4	Betriebsmittelzugriff	Verdrängungs-/Vorgangssperre
3	Auftragseinplanung	Ereignis, Priorität, Zeitscheibe, Energie
2	Ablaufsteuerung	Unterbrechungs-/Fortsetzungssperre, Transaktion
1	Kontextsicherung	Koroutine, <b>Unterbrechung</b> , <b>Fortsetzung</b> , Region
0	Stammprozessorabstraktion	ADT, Stammsystem
-1	Peripherie	MMU, (A)PIC, DMA, UART, ATA, SCSI, USB, ...
-2	Zentraleinheit	ARM, AVR, PowerPC, SPARC, x86, ...

## Vertiefung: SOS 1 [10] bzw. SP [11], BS [6]

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: **→ 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.

# Gliederung

## 1 Rekapitulation

## 2 Hardwareaspekte

- Signalisierungstechnik
- Unterbrechungssteuerung
- Ausnahmebehandlung

## 3 Softwareaspekte

- Unterbrechungslatenz
- Fehlhandlung
- Kontextsicherung
- Modularisierbarkeit

## 4 Zusammenfassung

# 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<sup>1</sup> 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

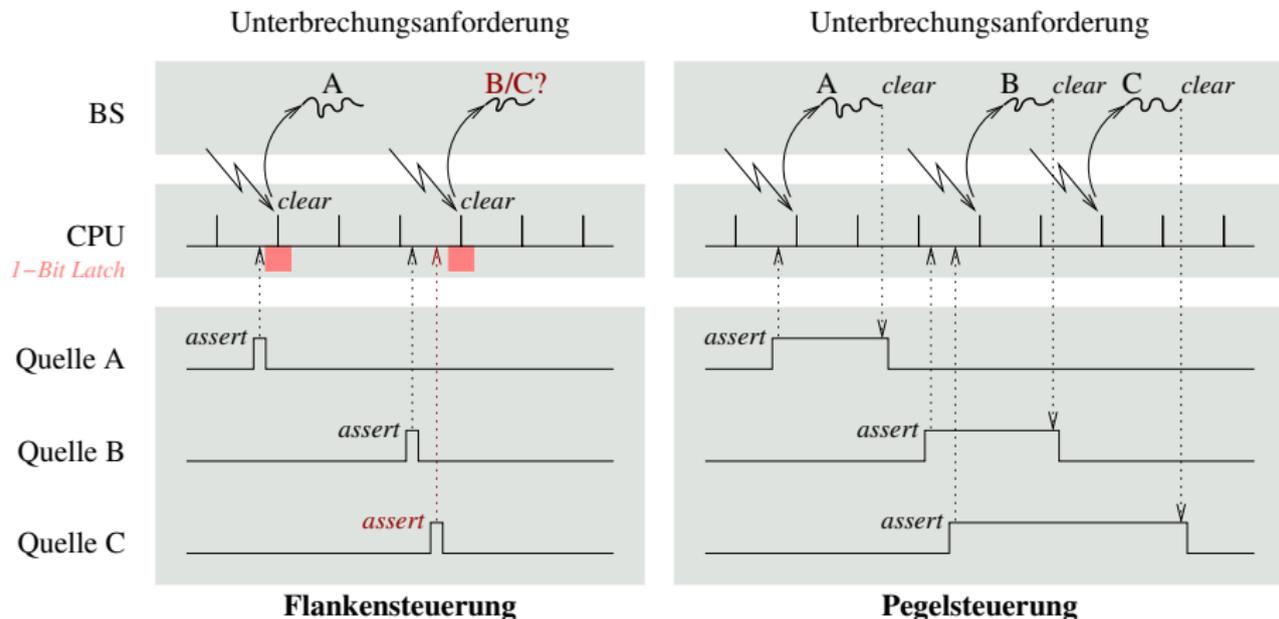
**pegelgesteuert** (engl. *level triggered*)  $\mapsto$  **Logikpegel**

- 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

---

<sup>1</sup>Von logisch „0“ (engl. *low*) zu logisch „1“ (engl. *high*) oder umgekehrt.

# Gemeinsame Benutzung derselben Unterbrechungsleitung



## Problemfälle

**Flankensteuerung** Wiederbehauptung, Unterbrechungssperre

**Pegelsteuerung** Nichtbehandlung

## „Gänseblümchenkette“ (engl. *daisy chain*)

**Wiederbehauptung** (engl. *reassertion*) einer Unterbrechung über eine mehreren Quellen gemeinsame Unterbrechungsanforderungsleitung

- vgl. Beispiel auf S. 8

### 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

# Abfragereihenfolge

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

- „Quellendistanz“ zur CPU: physikalische Abfragepriorität
  - Abfrage durch die CPU erfolgt typischerweise von „nah“ zu „fern“
- FPFS („*first polled, first served*“)  $\equiv$  FCFS  $\overset{?}{\rightsquigarrow}$  **Prioritätsverletzung**

## 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

## Flankensteuerung

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

# Signalisierung bei abgewehrten Unterbrechungen

Unterbrechungssperre d.h. zeitweises Abwehren von Unterbrechungen

## 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

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

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

## Pegelsteuerung

- Unterbrechungen werden erkannt, Ereignisse gehen nicht verloren

# Anwendungsbezug

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

**Universalsystem**  $\leadsto$  egal

- Textverarbeitung, Programmentwicklung, . . . , Unterhaltung

**Spezialsystem**  $\leadsto$  nicht egal  $\leftrightarrow$  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 **querschnittende Belange** in der Systemsoftware aus und sind nur schwer zu modularisieren  $\leadsto$  AOP [3]

# 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  $\leftrightarrow$  IRQ

- **Annahme** von Unterbrechungsanforderungen **verweigern**
  - an der *Senke* ansetzen
- Elementaroperation der Befehlssatzebene

PIC bedingt möglich  $\leftrightarrow$  IRQ

- **Weiterleitung** von Unterbrechungsanforderungen **unterbinden**
  - am ggf. vorhandenen *Mittler* ansetzen
- Programm der Befehlssatzebene: Gerätetreiber

Gerät immer möglich  $\leftrightarrow$  NMI/IRQ

- **Auslösen** von Unterbrechungsanforderungen **abstellen**
  - an der *Quelle* ansetzen
- Programm der Befehlssatzebene: Gerätetreiber

# Schutz kritischer Abschnitte durch Unterbrechungsabwehr

CPU mit  $N(IPL)$  gleich  $2^1 = 2$  (x86) bzw.  $2^3 = 8$  (m68k)

## 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");  
}
```

## Verwendung

```
#define ENTER irq_avert  
#define LEAVE irq_admit  
  
/* critical section */  
ENTER();  
...  
LEAVE();
```

## Beachte: Mögliche Ereignisverluste bei Flankensteuerung

- wenn in der Sperrphase Unterbrechungsanforderungen eingeht *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-vectorized interrupt*)  $\leftrightarrow$  RISC

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

**gerichtete Unterbrechung** (engl. *vectorized interrupt*)  $\leftrightarrow$  CISC

- die CPU nimmt die Ausführung einer Behandlungsroutine auf
  - jede Unterbrechungsart erhält eine solche Routine zugewiesen
    - $\leftrightarrow$  **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: Erzeugung von Vektorexemplaren

- ist möglich zur Programmier-, Übersetzungs-, Binde-, Lade- oder Ausführungszeit eines Betriebssystems
- wengleich 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

# Gliederung

- 1 Rekapitulation
- 2 Hardwareaspekte
  - Signalisierungstechnik
  - Unterbrechungssteuerung
  - Ausnahmebehandlung
- 3 Softwareaspekte
  - Unterbrechungslatenz
  - Fehlhandlung
  - Kontextsicherung
  - Modularisierbarkeit
- 4 Zusammenfassung

# Kaskadierte Unterbrechung

Programme zur Unterbrechungsbehandlung sind grundsätzlich mit der Tatsache konfrontiert, selbst unterbrochen werden zu können<sup>2</sup>

- 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 ?

---

<sup>2</sup>Programmierfehler nicht betrachtet.

# Verzögerung bis zur Programmunterbrechung

**Latenz** (von lat. *latens*, verborgen) steht für **Latenzzeit** und bezeichnet den Zeitraum zwischen einem verborgenen Ereignis und dem Eintreten einer sichtbaren Reaktion darauf [12]

**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<sup>3</sup> oder gar notwendig<sup>4</sup>

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:  $BS_{cop} \subset BS_{-cop}$
  - **paralleles Betriebssystem**:  $BS_{cop}$  greift in Abläufe von  $BS_{-cop}$  ein u.u.

---

<sup>3</sup>z.B. Hochleistungsrechnen [1]

<sup>4</sup>z.B. Echtzeitverarbeitung [4, 5]

# 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 Assemblersprache
- 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** [9]

## 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. 11)
- 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  $\iff$  **kein Pegel liegt mehr an** !
  - (Flanken-/Pegelsteuerung) der unbestimmten Ausdehnung des Stapelspeichers muss vorgebeugt werden
    - Entsperrung  $\iff$  **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 -sperrern

**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<sup>5</sup>

---

<sup>5</sup>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<sup>6</sup>
  - bei Pegelsteuerung nimmt das Gerät erst dann das Signal zurück
- stellt sicher, dass ein Kommunikationswunsch sein Ziel erreichte [8]

## Beachte

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

---

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

# Integritätswahrung unterbrochener Programmabläufe

**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 [10, 11]): Ebene <sub>2,3,5</sub>

**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

- Flanke, Pegel
- Latenz, Treiberfunktion, Synchronisation

## Unterbrechungslatenz

- Verbergung, Minimierung
- Operationsprinzip (BS), Treiberstruktur

## Integritätswahrung

- CPU, BS, Kompilierer
- Prozessorstatus je nach Anwendung/Ebene

## Unterbrechungsabwehr

- 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.

~ AOP [3]

# Gliederung

## 1 Rekapitulation

## 2 Hardwareaspekte

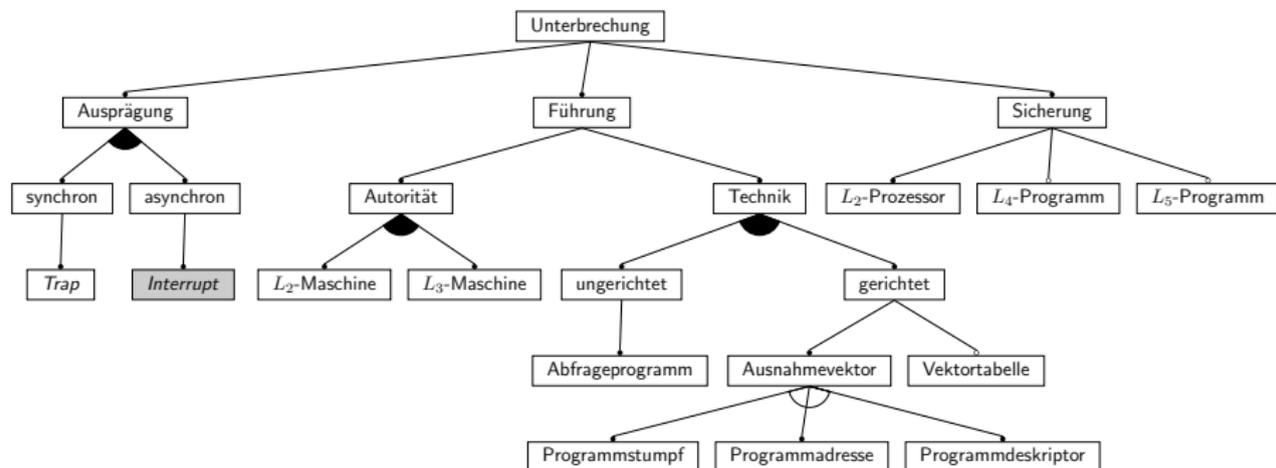
- Signalisierungstechnik
- Unterbrechungssteuerung
- Ausnahmebehandlung

## 3 Softwareaspekte

- Unterbrechungslatenz
- Fehlhandlung
- Kontextsicherung
- Modularisierbarkeit

## 4 Zusammenfassung

# Artefakte von Programmunterbrechungen



*L*<sub>2</sub>-Maschine

- CPU und ggf. PIC

*L*<sub>3</sub>-Maschine

- Emulation gerichteter Unterbrechungen im BS

*L*<sub>2</sub>-Prozessor

- Sicherung des min. Prozessorstatus' durch die CPU

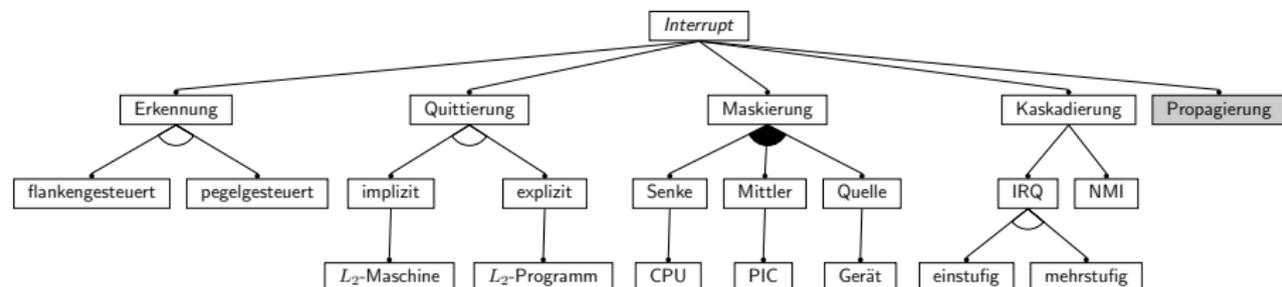
*L*<sub>4</sub>-Programm

- Sicherung der flüchtigen Register

*L*<sub>5</sub>-Programm

- Sicherung der nicht-flüchtigen Register

# Artefakte asynchroner Programmunterbrechungen



*L<sub>2</sub>-Maschine* • Kombination von CPU, PIC und Gerät

*L<sub>2</sub>-Programm* • FLIH, Vorspann eines asynchronen Systemsprungs

*Propagierung: Thema der nächsten Vorlesung...*

- Weiterleitung von Unterbrechungsanforderungen ans Betriebssystem
- Umwandlung eines Hardwaresignals in ein Softwaresignal
- Abbildung des Softwaresignals auf Entitäten des Betriebssystems

# Resümee

**Rekapitulation** ↔ Einbindung peripherer Prozesse

- Geräteprogrammierung, Kontextsicherung
- synchrone vs. asynchrone Programmunterbrechung

**Hardwareaspekte** ↔ Erkennung und Arten von Unterbrechungen

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

**Softwareaspekte** ↔ Unterbrechungslatenz

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

# Literaturverzeichnis

- [1] BRÜNING, U. ; GILOI, W. K. ; SCHRÖDER-PREIKSCHAT, W. :  
**Latency Hiding in Message-Passing Architectures.**  
In: SIEGEL, H. J. (Hrsg.): *Proceedings of the 8th Internation Symposium on Parallel Processing (IPPS '94)*.  
Washington, DC, USA : IEEE Computer Society, 1994. –  
ISBN 0-8186-5602-6, S. 704-709
- [2] GOODENOUGH, J. B.:  
**Exception Handling: Issues and a Proposed Notation.**  
In: *Communications of the ACM* 18 (1975), Nr. 12, S. 683-696
- [3] KICZALES, G. ; LAMPING, J. ; MENDHEKAR, A. ; MAEDA, C. ; LOPES, C. V. ; LOINGTIER, L.-M. ; IRWIN, J. :  
**Aspect-Oriented Programming.**  
In: AKSIT, M. (Hrsg.) ; MATSUOKA, S. (Hrsg.): *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP '97)* Bd. 1241, Springer-Verlag, 1997 (Lecture Notes in Computer Science). –  
ISBN 3-540-63089-9, S. 220-242

# Literaturverzeichnis (Forts.)

- [4] LEYVA-DEL-FOYO, L. E. ; MEJIA-ALVAREZ, P. ; DE NIZ, D. :  
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)*.  
Los Alamitos, CA, USA : IEEE Computer Society, 2006. –  
ISBN 0-7695-2516-4, S. 14-23
- [5] LEYVA-DEL-FOYO, L. E. ; MEJIA-ALVAREZ, P. ; DE NIZ, D. :  
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 2006)*.  
Los Alamitos, CA, USA : IEEE Computer Society, 2006. –  
ISBN 0-7695-2676-4, S. 385-394
- [6] LOHMANN, D. ; SCHRÖDER-PREIKSCHAT, W. :  
*Betriebssysteme*.  
[http://www4.informatik.uni-erlangen.de/Lehre/WS08/V\\_BS](http://www4.informatik.uni-erlangen.de/Lehre/WS08/V_BS), 2008 ff.

# Literaturverzeichnis (Forts.)

- [7] RYZHYK, L. ; CHUBB, P. ; KUZ, I. ; HEISER, G. :  
Dingo: Taming Device Drivers.  
In: SCHRÖDER-PREIKSCHAT, W. (Hrsg.) ; WILKES, J. (Hrsg.) ; ISAACS, R. (Hrsg.):  
*Proceedings of the Fourth ACM European Conference on Computer Systems (EuroSys '09)*.  
New York, NY, USA : ACM Press, 2009. –  
ISBN 978-1-60558-482-9, S. 275-288
- [8] SALTZER, J. H. ; REED, D. P. ; CLARK, D. D.:  
End-To-End Arguments in System Design.  
In: *ACM Transactions on Computer Systems* 2 (1984), Nov., Nr. 4, S. 277-288
- [9] 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

# Literaturverzeichnis (Forts.)

- [10] SCHRÖDER-PREIKSCHAT, W. ; KLEINÖDER, J. :  
*Softwaresysteme 1.*  
[http://www4.informatik.uni-erlangen.de/Lehre/SS04/V\\_S0S1](http://www4.informatik.uni-erlangen.de/Lehre/SS04/V_S0S1), 2004–2007
- [11] SCHRÖDER-PREIKSCHAT, W. ; KLEINÖDER, J. :  
*Systemprogrammierung.*  
[http://www4.informatik.uni-erlangen.de/Lehre/WS08/V\\_SP](http://www4.informatik.uni-erlangen.de/Lehre/WS08/V_SP), 2008 ff.
- [12] WIKIPEDIA FOUNDATION INC.:  
*Wikipedia, Die freie Enzyklopädie.*  
San Francisco, CA, USA : <http://de.wikipedia.org>,