Betriebssysteme (BS)

VL 3 – Unterbrechungen, Hardware

Daniel Lohmann

Lehrstuhl für Informatik 4
Verteilte Systeme und Betriebssysteme

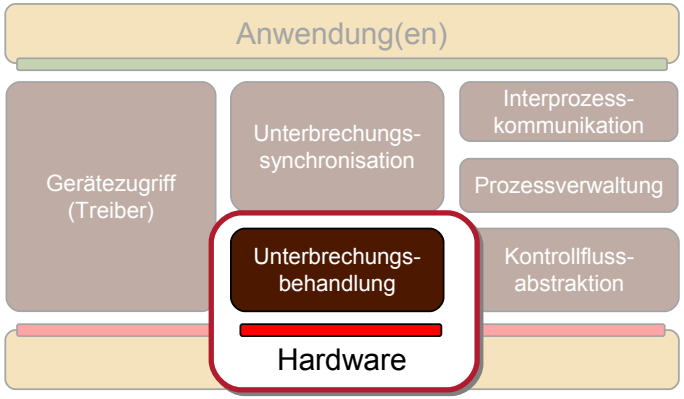
Friedrich-Alexander-Universität
Erlangen Nürnberg

WS 15 – 27. Oktober 2015

https://www4.cs.fau.de/Lehre/WS15/V_BS

Agenda

Überblick: Einordnung dieser VL



Agenda

Sinn und Zweck von Unterbrechungen

ein Blick zurück in die Historie von Betriebssystemen ...

- Überlappte Ein-/Ausgabe
  - Eingaben: Verschwendung von anderweitig nutzbaren Prozessorzyklen bei (oft nicht vorhersagbar langem) **aktivem Warten**
  - Ausgaben: selbständiges Agieren der E/A Geräte (z.B. durch **DMA**) entlastet die CPU
- *Timesharing* Betrieb
  - Zeitgeber Unterbrechungen geben dem Betriebssystem die Möglichkeit ...
    - zur **Verdrängung von Prozessen**
    - Aktivitäten zeitgesteuert zu starten

Priorisierung

- **Problem**:
  - Mehrere Unterbrechungsanforderungen können gleichzeitig signalisiert werden. Welche ist wichtiger?
  - Während die CPU auf die wichtigste Anforderung reagiert, können weitere Anforderungen signalisiert werden.

Priorisierung

- **Problem**:
  - Mehrere Unterbrechungsanforderungen können gleichzeitig signalisiert werden. Welche ist wichtiger?
  - Während die CPU auf die wichtigste Anforderung reagiert, können weitere Anforderungen signalisiert werden.
- **Lösung**: ein **Priorisierungsmechanismus** ...
  - **in Software**: die CPU hat nur einen IRQ (*interrupt request*) Eingang und fragt die Geräte in bestimmter Reihenfolge ab
  - **in Hardware**: eine Priorisierungsschaltung ordnet Geräten eine Priorität zu und leitet immer nur die dringendste Anforderung zur Behandlung weiter
  - **mit festen Prioritäten**: jedem Gerät wird statisch eine Priorität zugeordnet
  - **mit variablen Prioritäten**: Prioritäten sind dynamisch änderbar oder wechseln zum Beispiel zyklisch

Verlust von IRQs

- **Problem**:
  - während der Behandlung oder Sperrung von Unterbrechungen, kann die CPU keine neuen Unterbrechungen behandeln
  - die Speicherkapazität für Unterbrechungsanforderungen ist endlich.
    - i.d.R. ein Bit pro Unterbrechungseingang

Verlust von IRQs

- **Problem**:
  - während der Behandlung oder Sperrung von Unterbrechungen, kann die CPU keine neuen Unterbrechungen behandeln
  - die Speicherkapazität für Unterbrechungsanforderungen ist endlich.
    - i.d.R. ein Bit pro Unterbrechungseingang
- **Lösung**: in Software
  - die Unterbrechungsbehandlungsroutine sollte möglichst kurz sein (zeitlich!), um die Wahrscheinlichkeit von Verlusten zu minimieren
  - Unterbrechungen sollten nicht unnötig lange gesperrt werden
  - jeder Gerätetreiber sollte davon ausgehen, dass **eine** Unterbrechung **mehr als eine** abgeschlossene E/A Operation anzeigen kann

Zuordnung einer Behandlungsroutine

- **Problem**:
  - die Software soll mit möglichst wenig Aufwand herausfinden können, welches Gerät die Unterbrechung ausgelöst hat
    - eine sequentielle Abfrage der Geräte kostet nicht nur Zeit, sondern verändert die Zustände von E/A Bussen und unbeteiligten Geräten
- **Lösung**:
  - jeder Unterbrechung wird eine Nummer zugeordnet, die als Index in eine Vektortabelle verwendet wird
    - die Vektornummer hat nicht zwangsläufig etwas mit der Priorität zu tun
    - es kommt in der Praxis leider vor, dass Geräte sich eine Vektornummer teilen müssen (*interrupt sharing*)
  - der Aufbau der Vektortabelle variiert je nach Prozessortyp
    - meist enthält sie Zeiger auf Funktionen
    - seltener sind die Einträge selbst bereits Instruktionen

Zuordnung einer Behandlungsroutine

- **Problem**:
  - die Software soll mit möglichst wenig Aufwand herausfinden können, welches Gerät die Unterbrechung ausgelöst hat
    - eine sequentielle Abfrage der Geräte kostet nicht nur Zeit, sondern verändert die Zustände von E/A Bussen und unbeteiligten Geräten

Zustandssicherung

- **Problem**:
  - nach der Ausführung der Behandlungsroutine soll zum normalen Kontext zurückgekehrt werden können
  - die Behandlung soll quasi unbemerkt ablaufen (*transparency*)

Zustandssicherung

- **Problem**:
  - nach der Ausführung der Behandlungsroutine soll zum normalen Kontext zurückgekehrt werden können
  - die Behandlung soll quasi unbemerkt ablaufen (*transparency*)
- **Lösung**:
  - Zustandssicherung durch Hardware
    - nur das Notwendigste: z.B. Rücksprungadresse u. Prozessorstatuswort
    - Wiederherstellung durch speziellen Befehl, z.B. iret, rte, ...
  - Zustandssicherung durch Software
    - da Unterbrechungen jederzeit auftreten können, muss auch die Behandlungsroutine Zustände sichern und wiederherstellen

Geschachtelte Behandlung

- **Problem**:
  - um auf sehr wichtige Ereignisse schnell reagieren zu können, soll auch eine Unterbrechungsbehandlung unterbrechbar sein
  - eine unbegrenzte Schachtelungstiefe muss aber vermieden werden

Geschachtelte Behandlung

- **Problem**:
  - um auf sehr wichtige Ereignisse schnell reagieren zu können, soll auch eine Unterbrechungsbehandlung unterbrechbar sein
  - eine unbegrenzte Schachtelungstiefe muss aber vermieden werden
- **Lösung**:
  - die CPU erlaubt immer nur Unterbrechungen mit höherer Priorität
  - die aktuelle Priorität wird im Prozessorstatuswort gespeichert
  - die vorherige Priorität wird auf einem Stapel abgelegt

Multiprozessorsysteme

- **Problem**:
  - Unterbrechungen können immer nur von einer CPU behandelt werden. Aber welche?
  - es gibt eine weitere Kategorie von Unterbrechungen: die Interprozessor-Unterbrechungen
- **Lösung**: die Hardware zur Unterbrechungsbehandlung auf Multiprozessorsystemen muss komplexer ausgelegt sein. Es gibt viele Entwurfsvarianten ...
  - feste Zuordnung
  - zufällige Zuordnung
  - programmierbare Zuordnung
  - Zuordnung unter Berücksichtigung der Prozessorlast

  ... und Kombinationen davon.

Gefahr: „unechte Unterbrechungen“

(*„spurious interrupts“*)

- **Problem**: ein technischer Mechanismus zur Unterbrechungsbehandlung birgt die Gefahr von fehlerhaften Unterbrechungsanforderungen, z.B. durch ...
  - Hardwarefehler
  - fehlerhaft programmierte Geräte
- **Lösung**:
  - Hardware- und Softwarefehler vermeiden ☺
  - Betriebssystem „defensiv“ programmieren
    - mit unechten Unterbrechungen rechnen

Gefahr: „Unterbrechungsstürme“

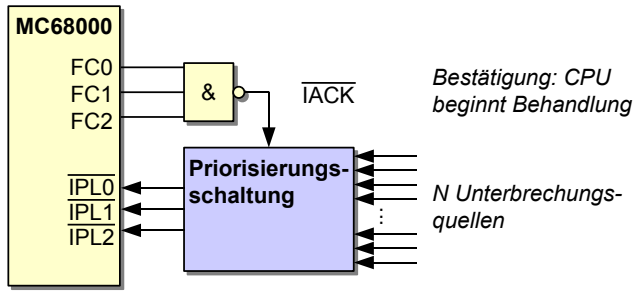
(*„interrupt storms“*)

- **Problem**:
  - hochfrequente Unterbrechungsanforderungen können einen Rechner lahm legen
  - es handelt sich entweder um unechte Unterbrechungen oder der Rechner ist mit der E/A Last überfordert
  - kann leicht mit Seitenflattern (*thrashing*) verwechselt werden
- **Lösung**: durch das Betriebssystem
  - Unterbrechungsstürme erkennen
  - das verursachende Gerät deaktivieren

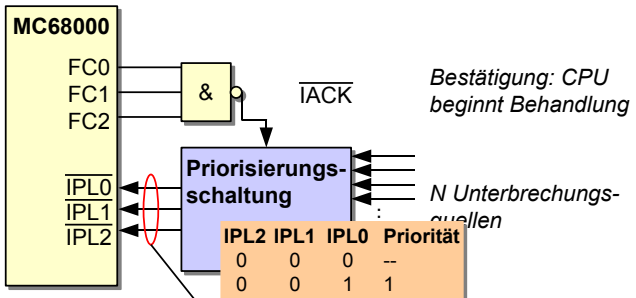
Agenda

Unterbrechungen beim MC68000

Unterbrechungen beim MC68000



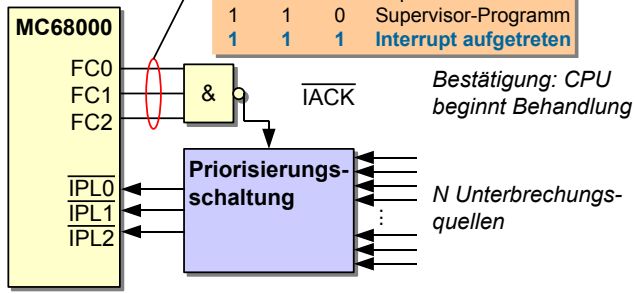
Unterbrechungen beim MC68000



IPL2	IPL1	IPL0	Priorität
0	0	0	--
0	0	1	1
0	1	0	2
0	1	1	3
1	0	0	4
1	0	1	5
1	1	0	6
1	1	1	7

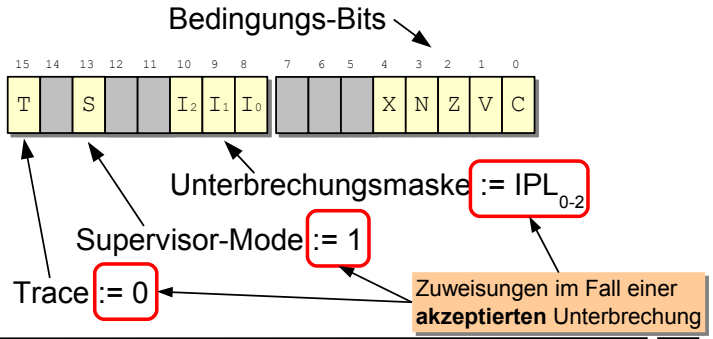
Unterbrechungen

FC2	FC1	FC0	Zyklustyp
0	0	0	reserviert
0	0	1	Anwender-Daten
0	1	0	Anwender-Program
0	1	1	reserviert
1	0	0	reserviert
1	0	1	Supervisor-Daten
1	1	0	Supervisor-Programm
1	**1**	**1**	**Interrupt aufgetreten**



Das Statusregister (SR) des MC68000

- enthält u.A. die aktuelle Unterbrechungsmaske
  - bei einer Unterbrechung wird geprüft, ob $IPL_{0-2} > I_{0-2}$ ist. Wenn nein, wird der Anforderung (noch) nicht stattgegeben.
  - eine Unterbrechung mit $IPL_{0-2} = 7$ wird aber immer bearbeitet (NMI)



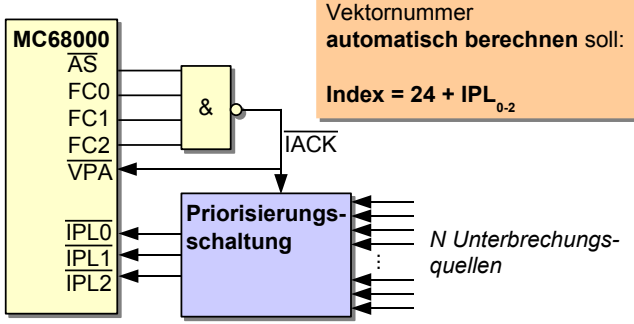
Vektortabelle des MC68000

Index	Adresse	Bedeutung
0	0x000	Reset: Supervisor-Stapelzeiger
1	0x004	Reset: PC
2	0x008	Busfehler
3	0x00c	Adressfehler
4	0x010	Illegaler Befehl
5	0x014	Division durch Null
...		
24	0x060	unechte Unterbrechung
25	0x064	**autovektorielle Unterbrechung, Ebene 1**
26	0x068	**autovektorielle Unterbrechung, Ebene 2**
...		
30	0x078	**autovektorielle Unterbrechung, Ebene 6**
31	0x07c	**autovektorielle Unterbrechung, Ebene 7 (NMI)**
32-47	0x080	TRAP-Befehlsvektoren
48-63	0x0c0	reserviert
64-255	0x100	**Anwender-Unterbrechungsvektoren**

Autovektorielle Unterbrechungen

über **VPA** signalisiert die externe Schaltung, dass die CPU die Vektornummer **automatisch berechnen** soll:

Index = 24 + IPL_{0-2}

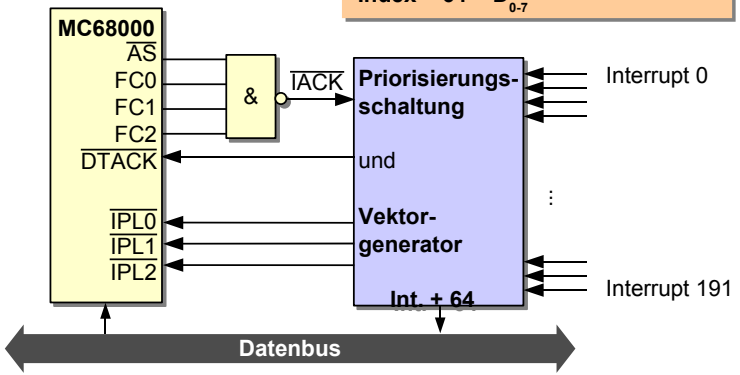


Problem: Es stehen nur 6 Vektoren für Geräte bereit. Bei mehr Geräten ist *„sharing“* nicht zu vermeiden.

Nicht-autovektorielle Unterbrechungen

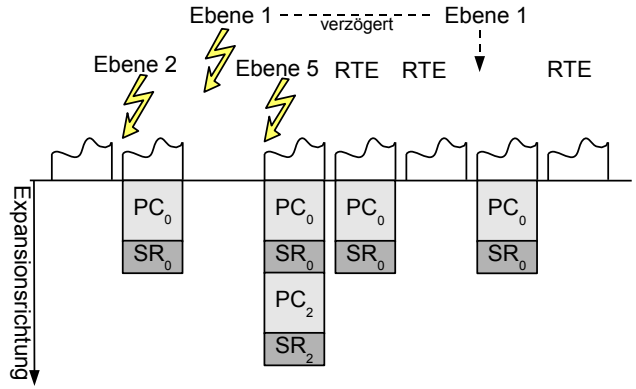
über **DTACK** signalisiert die externe Schaltung, dass die CPU die Vektornummer **über den Datenbus lesen** soll.

Index = 64 + D_{0-7}



Zustandssicherung beim MC68000

- der vorherige SR Inhalt und der PC werden bei einer Unterbrechung auf dem Supervisor-Stapel gesichert
- der RTE Befehl macht den Vorgang rückgängig



MC68000 - Zusammenfassung

- 6 Prioritätsebenen für Hardware-Unterbrechungen + NMI
  - Unterbrechungsebene 1-6, NMI Ebene 7
  - „Maskierung“ über I_{0-2} im Statusregister möglich
- nur Unterbrechungen höherer Priorität und der NMI können eine laufende Behandlung unterbrechen
  - Statusregister wird automatisch angepasst
- automatische Zustandssicherung auf dem *Supervisor*-Stapel, geschachtelte Behandlung möglich.
- die Vektornummernerzeugung erfolgt entweder ...
  - autovektoriell: Index = Priorität + 24
  - nicht-autovektoriell (durch externe Hardware): Index = 64 ... 255
- keine Multiprozessorunterstützung

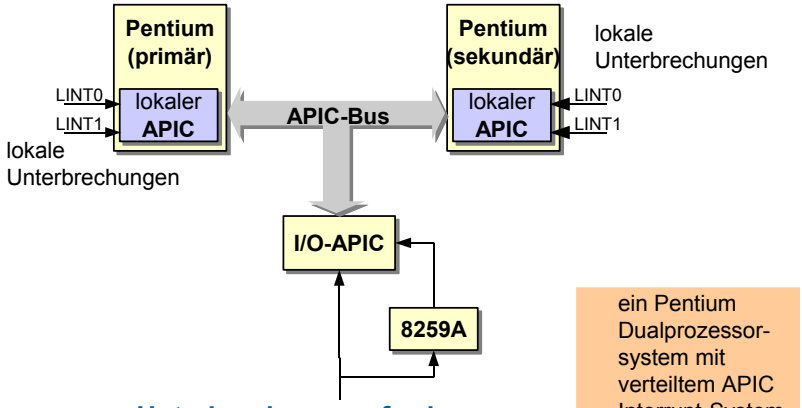
Unterbrechungen bei x86 CPUs

Unterbrechungen bei x86 CPUs

- bis einschließlich i486 hatten x86 CPUs nur einen IRQ und einen NMI Eingang
- externe Hardware sorgte für die Priorisierung und Vektornummerngenerierung
  - durch einen Chip namens **PIC 8259A**
    - 8 Interrupt-Eingänge
    - 15 Eingänge bei Kaskadierung von zwei PICs
    - keine Multiprozessorunterstützung
- heutige x86 CPUs enthalten den weit leistungsfähigeren **„Advanced Programmable Interrupt Controller“ (APIC)**
  - notwendig für **Multiprozessorsysteme**
  - inzwischen aber auch in allen Einprozessorsystemen aktiv
    - natürlich gibt es den PIC 8259A noch immer ☺

Die APIC Architektur

- ein APIC *Interrupt*-System besteht aus lokalen APICs auf jeder CPU und einem I/O APIC



Der I/O APIC

- heute typischerweise in der *Southbridge* von PC Chipsätzen integriert
- normalerweise 24 *Interrupt*-Eingänge
  - zyklische Abfrage (Round-Robin Priorisierung)
- für jeden Eingang gibt es einen 64 Bit Eintrag in der ***Interrupt Redirection Table***
  - beschreibt das Unterbrechungssignal
  - dient der Generierung der APIC-Bus Nachricht

Der I/O APIC

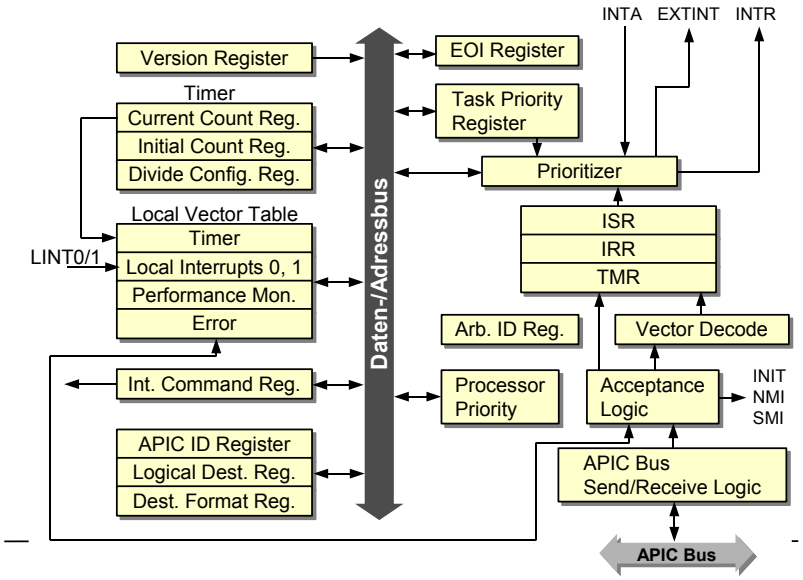
Aufbau (Bits) eines Eintrags in der *Interrupt Redirection Table*

63:56	**Destination Field**	– R/W. 8 Bit Zieladresse.
	je nach Bit 11:	APIC ID der CPU (*Physical Mode*) oder CPU Gruppe (*Logical Mode*)
55:17	<reserviert>	
16	**Interrupt-Mask**	– R/W. Unterbrechungssperre.
15	**Trigger Mode**	– R/W. *Edge*- oder *Level-Triggered*
14	**Remote IRR**	– RO. Art der erhaltenen Bestätigung
13	**Interrupt Pin Polarity**	– R/W. Signalpolarität
12	**Delivery Status**	– RO. Interrupt-Nachricht unterwegs?
11	**Destination Mode**	– R/W. *Logical Mode* oder *Physical Mode*
10:8	**Delivery Mode**	– R/W. Wirkung bei Ziel-APIC
	000 – *Fixed*:	Signal an alle Zielprozessoren ausliefern
	001 – *Lowest Priority*:	Liefern an CPU mit aktuell niedrigster Prio.
	010 – SMI:	*System Management Interrupt*
	100 – NMI:	*Non-Maskable Interrupt*
	101 – INIT:	Ziel-CPUs initialisieren (Reset)
	111 – ExtINT:	Antwort an PIC 8259A
7:0	**Interrupt Vector**	– R/W. **8 Bit Vektornummer (16 – 254)**

Local APICs

- empfangen Unterbrechungsanforderungen vom APIC Bus
- führen die Auswahl und Priorisierung durch
- können zwei lokale Unterbrechungen direkt verarbeiten
- enthalten weitere Funktionseinheiten
  - Eingebauten *Timer*, *Performance Counter*
  - *Command*-Register
    - um selber APIC-Nachrichten zu verschicken
    - insbesondere Inter-Prozessor-Interrupt (IPI)
- programmierbar über 32 Bit Register ab 0xfee00000
  - memory mapped (ohne externe Buszyklen)
  - jede CPU programmiert „ihren“ *Local APIC*

Local APICs - Register



APIC Architektur - Zusammenfassung

- flexible Verteilung an CPUs im x86 Multiprozessorsystem
  - fest, Gruppen, an die CPU mit der geringsten Priorität
  - Liegen mehrere IRQs an, so wird nach Vektornummer priorisiert
- Vektornummer 16-254 können frei zugeordnet werden
  - sollte (an sich) reichen, um „sharing“ zu vermeiden
- *Local* APIC erwartet explizites EOI
  - dafür muss die Software sorgen
- Mit APIC unterstützt x86 prinzipiell auch Prioritätsebenen
  - Systemsoftware muss jedoch entsprechend agieren (Unterbrechungen freigeben, evtl. *Task-Priority-Register* verwenden)

Agenda

Zusammenfassung und Ausblick

- Unterbrechungsbehandlungshardware befasst sich mit ...
  - Priorisierung
  - Zuordnung und Ausführung einer Behandlungsroutine
  - Zustandssicherung und geschachtelter Ausführung
- moderne Unterbrechungsbehandlungshardware kann ...
  - Unterbrechungsvektoren frei zuordnen
  - *„sharing“* von Unterbrechungsvektoren vermeiden
  - Unterbrechungen im Multiprozessorsystem flexibel zuordnen
- das Betriebssystem muss ...
  - Probleme wie *„spurious interrupts“* und *„interrupt storms“* einkalkulieren.
  - das eingetretene Ereignis aus der Behandlungsroutine an die höheren Ebenen und letztendlich zum Anwendungsprozess weiterleiten.

Betriebssysteme (BS)

VL 4 – Unterbrechungen, Software

Daniel Lohmann

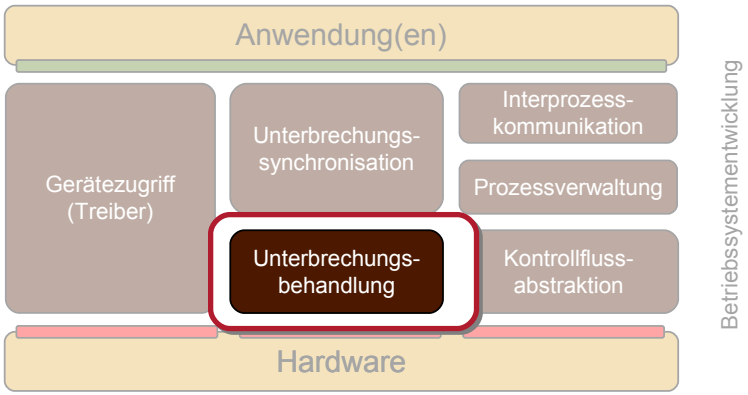
Lehrstuhl für Informatik 4
Verteilte Systeme und Betriebssysteme

Friedrich-Alexander-Universität
Erlangen Nürnberg

WS 15 – 03. November 2015

`https://www4.cs.fau.de/Lehre/WS15/V_BS`

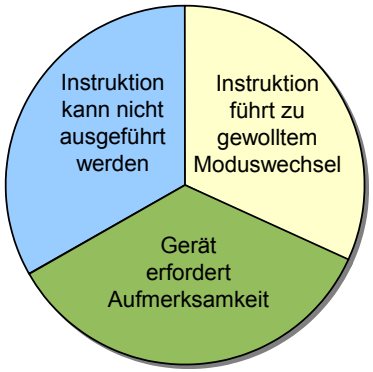
Überblick: Einordnung dieser VL



Agenda

Begriffe

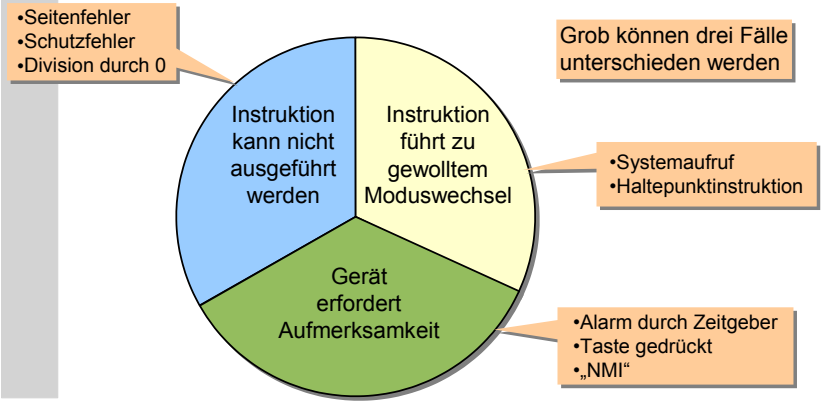
- das Verständnis der Begriffe ist unterschiedlich ...
  - zwecks Klärung begeben wir uns auf die technische Ebene



Grob können drei Fälle unterschieden werden

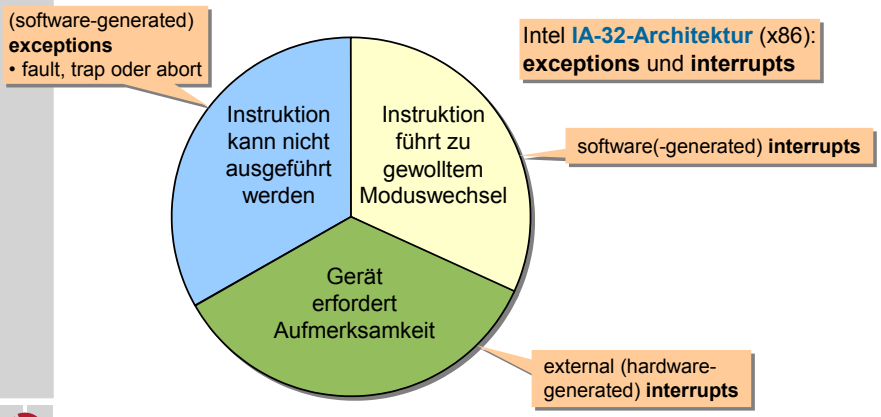
Begriffe

- das Verständnis der Begriffe ist unterschiedlich ...
  - zwecks Klärung begeben wir uns auf die technische Ebene



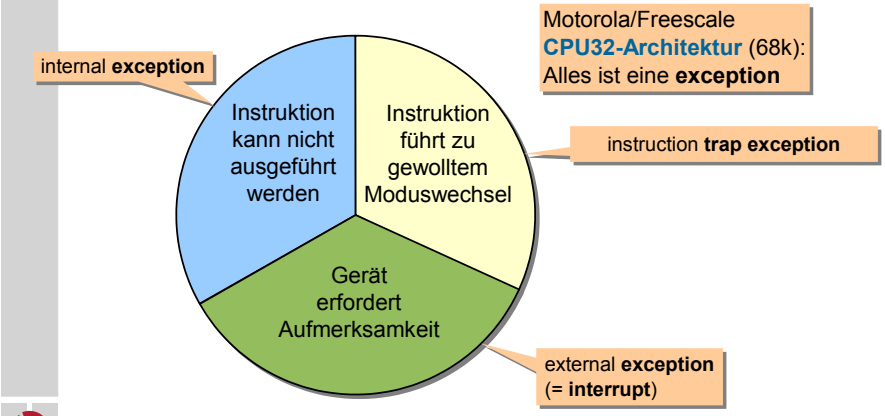
Begriffe: Intel IA-32

- das Verständnis der Begriffe ist unterschiedlich ...
  - zwecks Klärung begeben wir uns auf die technische Ebene



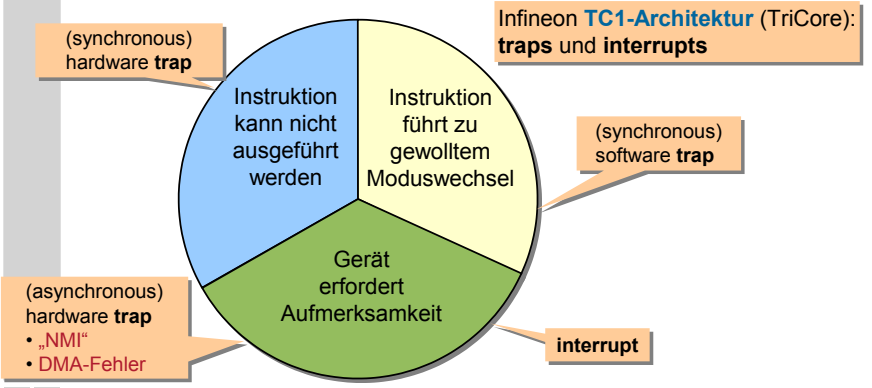
Begriffe: Motorola/Freescale CPU32

- das Verständnis der Begriffe ist unterschiedlich ...
  - zwecks Klärung begeben wir uns auf die technische Ebene



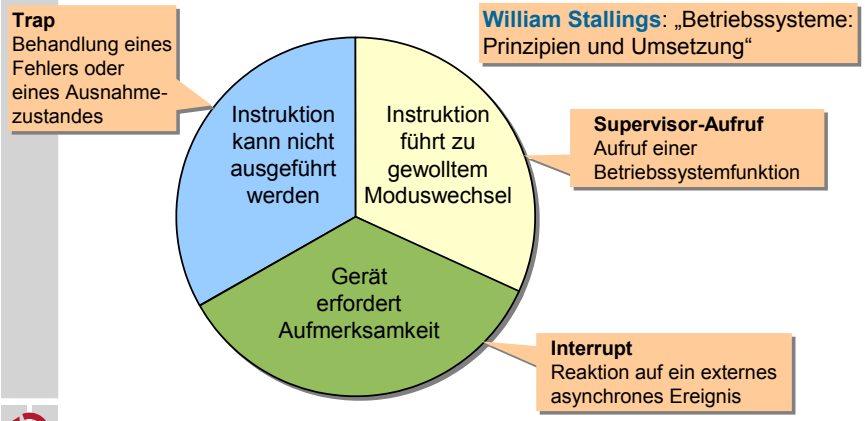
Begriffe: Infineon TC1

- das Verständnis der Begriffe ist unterschiedlich ...
  - zwecks Klärung begeben wir uns auf die technische Ebene



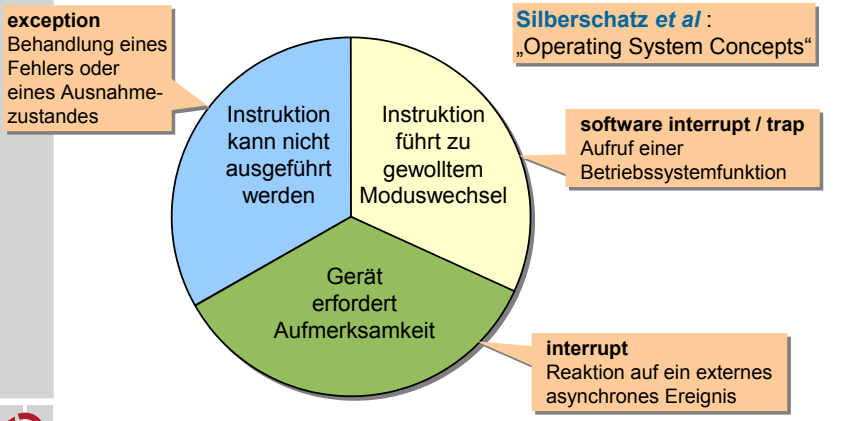
Begriffe: Literatur (Stallings)

- das Verständnis der Begriffe ist unterschiedlich ...
  - zwecks Klärung begeben wir uns auf die technische Ebene



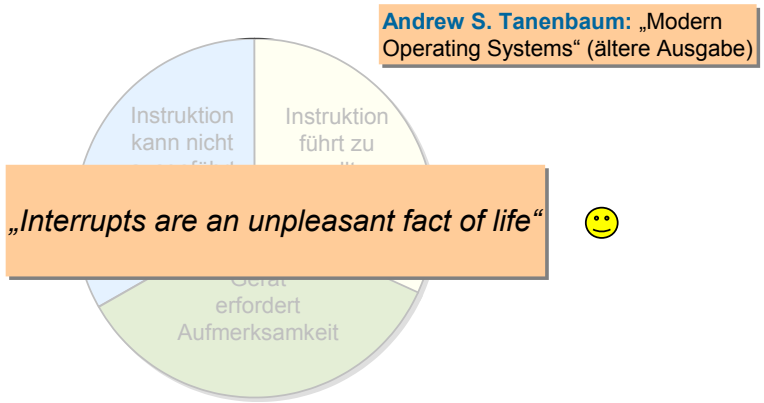
Begriffe: Literatur (Silberschatz)

- das Verständnis der Begriffe ist unterschiedlich ...
  - zwecks Klärung begeben wir uns auf die technische Ebene



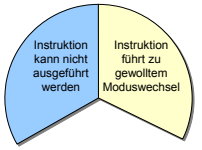
Begriffe: Literatur (Tanenbaum)

- das Verständnis der Begriffe ist unterschiedlich ...
  - zwecks Klärung begeben wir uns auf die technische Ebene



Begriffe: Unser Verständnis in BS

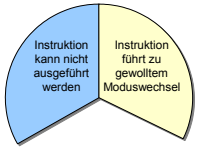
- ***„Trap“***
  - durch **Instruktion** ausgelöst
    - auch die „trap“ oder „int“ Instruktion für Systemaufrufe
    - nicht definiertes Ergebnis (z.B. Division durch 0)
    - Hardware-Problem (z.B. Busfehler)
    - Betriebssystem muss eingreifen (z.B. Seitenfehler)
    - ungültige Instruktion (z.B. bei Programmfehler)
  - Eigenschaften
    - oft vorhersagbar, oft reproduzierbar
    - Wiederaufnahme **oder Abbruch** der auslösenden Aktivität



- **„Unterbrechung“** (engl. *Interrupt*):
  - durch **Hardware** ausgelöst
    - Hardware verlangt die Aufmerksamkeit der Software (Zeitgeber, Tastatursteuereinheit, Festplattensteuereinheit, ...)
  - Eigenschaften:
    - nicht vorhersagbar, nicht reproduzierbar
    - in der Regel Wiederaufnahme der unterbrochenen Aktivität

Begriffe: Unser Verständnis in BS

- **„*Trap*“**
  - durch **Instruktion** ausgelöst
    - auch die „trap“ oder „int“ Instruktion für Systemaufrufe
    - nicht definiertes Ergebnis (z.B. Division durch 0)
    - Hardware-Problem (z.B. Busfehler)
    - Betriebssystem muss eingreifen (z.B. Seitenfehler)
    - ungültige Instruktion (z.B. bei Programmfehler)
  - Eigenschaften
    - oft vorhersagbar, oft reproduzierbar
    - Wiederaufnahme **oder Abbruch** der auslösenden Aktivität



- **„Unterbrechung“** (engl. *Interrupt*):
  - durch **Hardware** ausgelöst
    - Hardware verlangt die Aufmerksamkeit der Software (Zeitgeber, Tastatursteuereinheit, Festplattensteuereinheit, ...)
  - Eigenschaften:
    - nicht vorhersagbar, nicht reproduzierbar
    - in der Regel Wiederaufnahme der unterbrochenen Aktivität

Grundannahmen

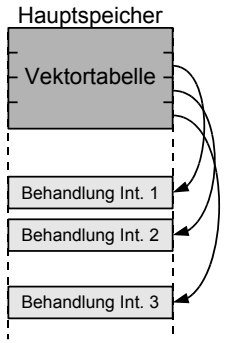
Wir betrachten die Behandlung von Unterbrechungen unter den folgenden Grundannahmen

1. Die CPU startet die Behandlungsroutine automatisch.
2. Die Unterbrechnungsbehandlung erfolgt im Systemmodus.
3. Das unterbrochene Programm kann fortgesetzt werden.
4. Die Maschineninstruktionen verhalten sich atomar.
5. Die Unterbrechungsbehandlung kann unterdrückt werden.

Grundannahmen: Behandlungsroutine

1. Die CPU startet die Behandlungsroutine automatisch.

- erfordert die Zuordnung einer Behandlungsroutine
- Ermittlung der Unterbrechungsursache nötig



Varianten:
- Register enthält Startadresse der Vektortabelle
- Tabelleneinträge enthalten Code
- Programmierbarer „Event Controller“ behandelt die Unterbrechung in Hardware
- Tabelle enthält Deskriptoren
- Behandlungsroutine hat eigenen Prozesskontext

Grundannahmen: Systemmodus

2. Die Unterbrechungsbehandlung erfolgt im Systemmodus.

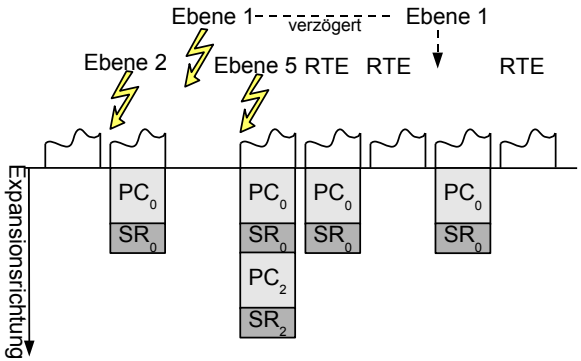
- Unterbrechungen sind der einzige Mechanismus, um nicht-kooperativen Anwendungen die CPU zu entziehen
- nur das BS darf uneingeschränkt auf Geräte zugreifen
- die CPU schaltet daher vor der Unterbrechungs-behandlung in den privilegierten Systemmodus

Varianten:
- bei **16-Bit-CPUs** ist eine Aufteilung in Benutzer-/Systemmodus eher die Ausnahme
- bei **8-Bit-CPUs** (oder kleiner) gibt es diese Aufteilung nicht

Grundannahmen: Kontextsicherung

3. Das unterbrochene Programm kann fortgesetzt werden.

- notwendiger Zustand wird automatisch gesichert
- ggf. auch geschachtelt, erfordert Stapel



Varianten:
- weitere Informationen über die Ursache auf dem Stapel
- keine Prioritäten
- spezieller „Interrupt-Stack“
- Kontextsicherung in Registern

Grundannahmen: Atomares Verhalten

4. Die Maschineninstruktionen verhalten sich atomar.

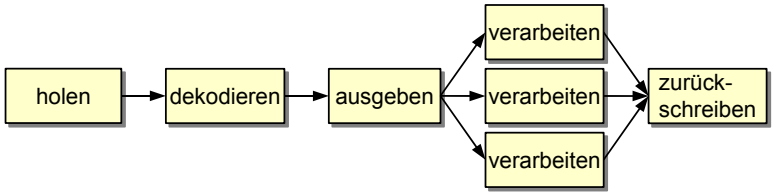
- definierter CPU-Zustand zu Beginn der Behandlungsroutine
- Wiederherstellbarkeit des Zustands

- trivial bei CPUs mit klassischem von-Neumann-Zyklus
- nicht-trivial bei modernen CPUs:
  - Fließbandverarbeitung: Befehle müssen annuliert werden
  - Superskalare CPUs: zusätzlich Befehlsreihenfolge merken

Grundannahmen: Atomares Verhalten

4. Die Maschineninstruktionen verhalten sich atomar.

Befehlsverarbeitung bei superskalaren Prozessoren:
(stark vereinfacht!)



Im Idealfall werden alle Stufen immer benutzt, d.h. mehrere Befehle werden parallel ausgeführt. Wann soll geprüft werden, ob eine Unterbrechungsanforderung anliegt?

Grundannahmen: Atomares Verhalten

4. Die Maschineninstruktionen verhalten sich atomar.

Trotz der Schwierigkeiten liefern die meisten CPUs „**präzise Unterbrechungen**“:

- *„All instructions preceding the instruction indicated by the saved program counter have been executed and have modified the process state correctly.“*
- *„All instructions following the instruction indicated by the saved program counter are unexecuted and have not modified the process state.“*
- *„If the interrupt is caused by an exception condition raised by an instruction in the program, the saved program counter points to the interrupted instruction. The interrupted instruction may or may not have been executed, depending on the definition of the architecture and the cause of the interrupt. Whichever is the case, the interrupted instruction has either completed, or has not started execution.“*

J. E. Smith and A. R. Pleszkun,
„Implementing Precise Interrupts in Pipelined Processors“,
IEEE Transactions on Computers, Vol. 37, No. 5, 1988

Grundannahmen: Unterdrückung

5. Die Unterbrechungsbehandlung kann unterdrückt werden.

- Beispiele:
  - Motorola 680x0: entsprechend der Priorität
  - Intel x86: global mit `sti`, `cli`
  - *Interrupt Controller*: jede Quelle einzeln
- automatische Unterdrückung erfolgt auch durch die CPU vor Betreten der Behandlungsroutine

Grundannahmen: Unterdrückung

5. Die Unterbrechungsbehandlung kann unterdrückt werden.

- automatische Unterdrückung erfolgt auch durch die CPU vor Betreten der Behandlungsroutine
  - Unterbrechungen nicht vorhersagbar, theoretisch beliebig häufig
  - Stapelüberlauf könnte nicht ausgeschlossen werden
- unterdrückt wird (durch die Hardware) die Behandlung...
  - pauschal aller Unterbrechungen (sehr restriktiv)
  - Unterbrechungen niedriger oder gleicher Priorität (weniger restriktiv)
    - bestimmte Geräte werden bevorzugt
- bessere Modelle mit Hilfe von Software (z.B. in Linux):
  - Unterbrechungen, die bereits behandelt werden, werden unterdrückt
    - hohe Reaktivität ohne Bevorzugung einzelner Geräte

Zustandssicherung

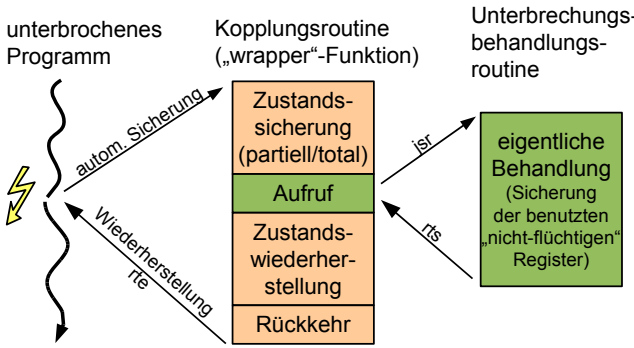
- der Zustand eines Rechners ist enorm groß
  - alle Prozessorregister
    - Instruktionszeiger, Stapelzeiger, Vielzweckregister, Statusregister, ...
  - der komplette Hauptspeicherinhalt, Caches
  - der Inhalt von E/A-Registern bzw. *Ports*, Festplatteninhalte, ...
- jeglicher benutzter Zustand, dessen asynchrone Änderung das unterbrochene Programm nicht erwartet, ...
  - darf während der Unterbrechungsbehandlung nicht modifiziert werden
  - muss gesichert und später wiederhergestellt werden
- die CPU sichert (je nach Typ) automatisch ...
  - minimal wenige Bytes (nur Instruktionszeiger und Statusregister)
  - alle Register

Zustandssicherungskonzepte

- **totale Sicherung**
  - die Behandlungsroutine sichert alle Register, die nicht automatisch gesichert wurden
  - Nachteil: eventuell wird zu viel gesichert
  - Vorteil: gesicherter Zustand leicht „zugreifbar“
- **partielle Sicherung**
  - die Behandlungsroutine sichert nur die Register, die im weiteren Verlauf geändert werden bzw. nicht gesichert und wieder hergestellt werden
  - machbar, wenn die eigentliche Behandlung in einer Hochsprache wie C oder C++ implementiert ist
  - Vorteile:
    - nur veränderter Zustand wird auch gesichert
    - evtl. weniger Instruktionen zum Sichern und Wiederherstellen nötig
  - Nachteil: gesicherter Zustand „verstreut“

Übergang auf die Hochsprachenebene

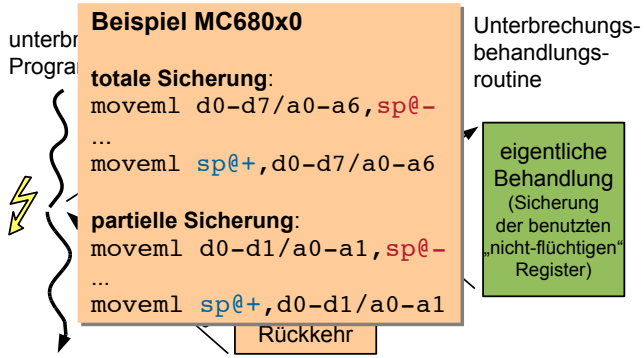
- nicht-portabler Maschinencode sollte minimiert werden
- die eigentliche Unterbrechungsbehandlung erfolgt in einer Hochsprachenfunktion



Sprache: beliebig Assembler! Hochsprache

Übergang auf die Hochsprachenebene

- nicht-portabler Maschinencode sollte minimiert werden
- die eigentliche Unterbrechungsbehandlung erfolgt in einer Hochsprachenfunktion



Sprache: beliebig Assembler! Hochsprache

Flüchtige und nicht-flüchtige Register

- eine Aufteilung, die **der (C/C++) Übersetzer** vornimmt
  - **nicht-flüchtig**
    - der Übersetzer garantiert, dass der Wert dieser Register über Funktionsaufrufe hinweg erhalten bleibt
    - ggf. in der aufgerufenen Funktion gesichert und wiederhergestellt
  - **flüchtig** (engl. *scratch registers*)
    - wenn die aufrufende Funktion den Wert auch nach dem Aufruf noch benötigt, muss das Register selbst (beim Aufrufer) gesichert werden
    - normalerweise für Zwischenergebnisse verwendet
- üblicherweise gibt es jedoch einen Standard
  - an den sich alle Übersetzer halten
  - Beispiel x86:
    - `eax`, `ecx`, `edx` und `eflags` gelten als flüchtig

Wiederherstellung

- die Kopplungsroutine muss alle gesicherten Registerinhalte am Ende wieder laden
  - ... und dann nicht mehr verändern!
- mit einer speziellen Instruktion (z.B. `rte` oder `iret`) wird der vorherige Zustand wiederhergestellt
  - Lesen des automatisch gesicherten Zustands von *Supervisor-Stack*
  - Setzen des gesicherten Arbeitsmodus (Benutzer-/Systemmodus) und Sprung an die gesicherte Adresse

Das BS kann den Zustand auch vor dem `rte`/`iret` ändern. Dies wird gerne ausgenutzt, um BS-Code im Benutzermodus auszuführen.

Zustandsänderungen ...

- sind Sinn und Zweck der Unterbrechungsbehandlung
  - Gerätetreiber müssen über den Abschluss einer E/A Operation informiert werden
  - der Scheduler muss erfahren, dass eine Zeitscheibe abgelaufen ist
- müssen mit Vorsicht durchgeführt werden
  - Unterbrechungen können zu jeder Zeit auftreten
  - kritisch sind Daten/Datenstrukturen, die der normale Kontrollfluss und die Unterbrechungsbehandlung sich teilen

Agenda

Beispiel 1: Systemzeit

- per Zeitgeberunterbrechung wird die globale Systemzeit inkrementiert
  - z.B. einmal pro Sekunde
- mit Hilfe einer Betriebssystemfunktion `time()` kann die Systemzeit abgefragt werden

```
/* globale Zeitvariable */
extern volatile time_t global_time;
```

```
/* Systemzeit abfragen */

time_t time () {
  return global_time;
}
```

```
/* Unterbrechungs- *
 * behandlung      */
void timerHandler () {
  global_time++;
}
```

Beispiel 1: Systemzeit

- hier schlummert möglicherweise ein Fehler ...
  - das Lesen von global_time erfolgt nicht notwendigerweise atomar!

```
32-Bit-CPU:
mov global_time, %eax
```

```
16-Bit-CPU (little endian):
mov global_time, %r0; lo
mov global_time+2, %r1; hi
```

- kritisch ist eine Unterbrechung zwischen den beiden Leseinstruktionen bei der 16-Bit-CPU

Instruktion	global_time hi / lo	Resultat r1 / r0
?	002A FFFF	? ?
mov global_time, %r0	002A FFFF	? FFFF
/* Inkrementierung */	002B 0000	? FFFF
mov global_time+2, %r1	002B 0000	**002B FFFF**

Beispiel 1: Systemzeit

- hier schlummert möglicherweise ein Fehler ...
  - das Lesen von global_time erfolgt nicht notwendigerweise atomar!

```
32 Bit C                                  le endian):
mov glob                                   %r0; lo
                                          2, %r1; hi
```

Problem:

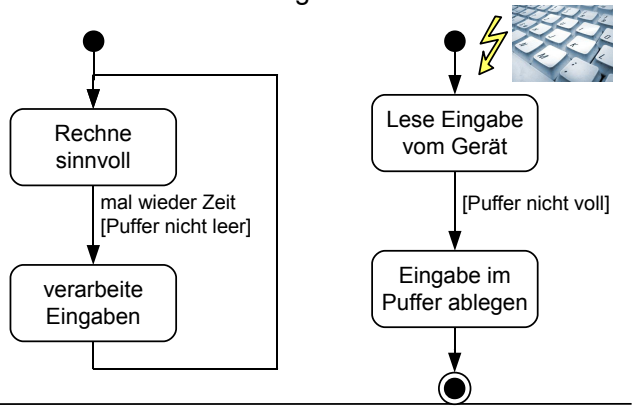
Alle 18,2 Stunden kann die Systemzeit (kurz) um etwa die gleiche Zeit vorgehen. Leider ist das Problem nicht verlässlich reproduzierbar.

- kritisch ist eine Unterbrechung zwischen den beiden Leseinstruktionen bei der 16 Bit CPU

Instruktion	global_time hi / lo	Resultat r1 / r0
?	002A | FFFF	? | ?
mov global_time, %r0	002A | FFFF	? | FFFF
/* Inkrementierung */	002B | 0000	? | FFFF
mov global_time+2, %r1	002B | 0000	002B | FFFF

Beispiel 2: Ringpuffer

- Unterbrechungen wurden eingeführt, damit das System **nicht aktiv** auf Eingaben warten muss
  - während gerechnet wird, kann die Unterbrechungsbehandlung Eingaben in einem Puffer ablegen

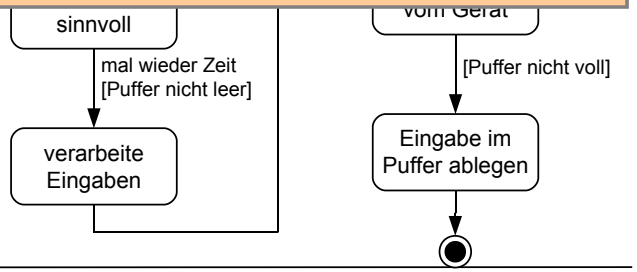


Beispiel 2: Ringpuffer

- Unterbrechungen wurden eingeführt, damit das System **nicht aktiv** auf Eingaben warten muss
  - wäh… …dlung Ein…

Problem:

Wenn die Eingabe nicht schnell genug verarbeitet werden, kann der Puffer voll werden. Die Behandlungsroutine kann die Eingabe dann nicht im Puffer ablegen. In diesem Fall geht die Eingabe verloren.



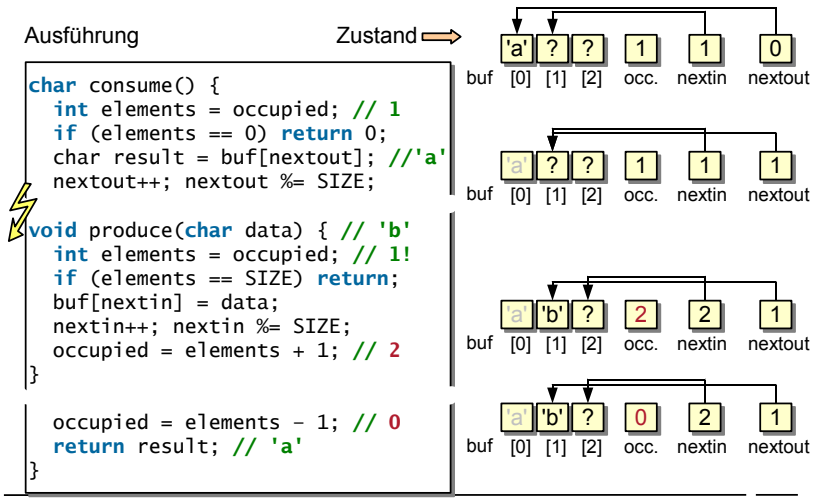
Beispiel 2: Ringpuffer

auch die Pufferimplementierung ist kritisch ...

```
// Pufferklasse in C++
class BoundedBuffer {
  char buf[SIZE]; int occupied; int nextin, nextout;
public:
  BoundedBuffer(): occupied(0), nextin(0), nextout(0) {}
  void produce(char data) {          // Unterbrechungsbehandlung:
    int elements = occupied;         // Elementzähler merken
    if (elements == SIZE) return;    // Element verloren
    buf[nextin] = data;              // Element schreiben
    nextin++; nextin %= SIZE;        // Zeiger weitersetzen
    occupied = elements + 1;         // Zähler erhöhen
  }
  char consume() {                   // normaler Kontrollfluss:
    int elements = occupied;         // Elementzähler merken
    if (elements == 0) return 0;     // Puffer leer, kein Ergebnis
    char result = buf[nextout];      // Element lesen
    nextout++; nextout %= SIZE;      // Lesezeiger weitersetzen
    occupied = elements - 1;         // Zähler erniedrigen
    return result;                   // Ergebnis zurückliefern
} };
```

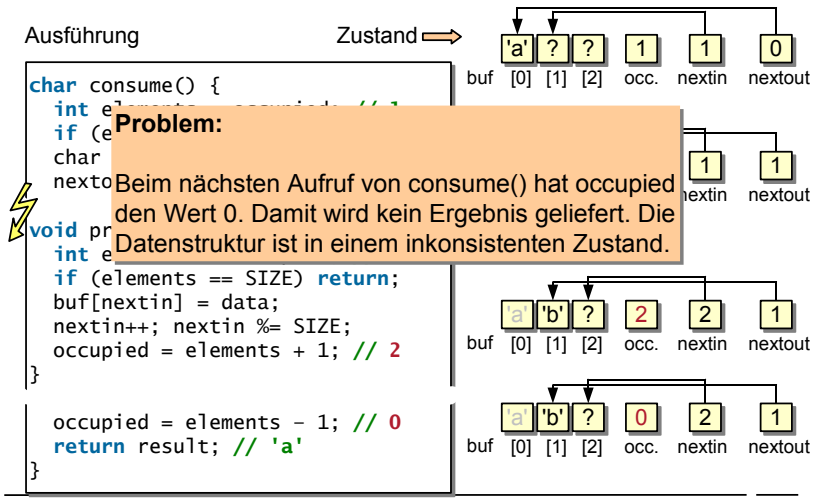
Beispiel 2: Ringpuffer

auch die Pufferimplementierung ist kritisch ...



Beispiel 2: Ringpuffer

auch die Pufferimplementierung ist kritisch ...



Zustandsänderung: Analyse

- selbst einzelne Zuweisungen müssen nicht atomar sein
  - Abhängigkeit vom CPU-Typ, Übersetzer und Codeoptimierung
- Pufferspeicher ist endlich
  - Behandlungsroutine kann nicht warten
  - Daten können verloren gehen
- Pufferdatenstruktur kann kaputt gehen aufgrund von ...
  - inkonsistenten Zwischenzuständen bei Änderungen durch den normalen Kontrollfluss
  - Zustandsänderungen während des Lesens (inkonsistente Kopie!)
  - Änderungen mit Hilfe einer Kopie, die nicht mehr dem Original entspricht
- das Problem ist nicht symmetrisch
  - der normale Kontrollfluss „unterbricht“ nicht die Unterbrechungsbehandlung
  - kann ausgenutzt werden!

„Harte“ Synchronisation

- Durch Unterdrückung von Unterbrechungen können *race conditions* vermieden werden:

```
char consume() {                    // normaler Kontrollfluss:
  disable_interrupts();             // Unterbrechungen verbieten
  int elements = occupied;          // Elementzähler merken
  if (elements == 0) {              // Puffer leer, kein Ergebnis
    enable_interrupts();            // Unterbrechungen zulassen
    return 0;
  }
  char result = buf[nextout];       // Element lesen
  nextout++; nextout %= SIZE;       // Lesezeiger weitersetzen
  occupied = elements - 1;          // Zähler erniedrigen
  enable_interrupts();              // Unterbrechungen zulassen
  return result;                    // Ergebnis zurückliefern
}
```

- Probleme:
  - Gefahr des Verlusts von Unterbrechungsanforderungen
  - hohe und schwer vorherzusagende **„Unterbrechungslatenzen“**

Weitere Techniken in der nächsten VL

- „schlaue“ (optimistische) Verfahren
  - Datenstruktur geschickt wählen
    - möglichst wenige geteilte Elemente
    - mit weichen Konsistenzbedingungen arbeiten
  - optimistisch herangehen
    - i.d.R. tritt keine Unterbrechung im kritischen Abschnitt auf
    - falls doch, wird der Schaden festgestellt und repariert
    - ggf. wird die Operation auch wiederholt
- Pro-/Epilogmodell
  - Aufteilung der Unterbrechungsbehandlung in zwei Phasen
    - der kritische Teil wird durch einen Softwaremechanismus verzögert
    - schnelle Reaktion weiterhin möglich

Agenda

Zusammenfassung

- die korrekte Behandlung von Unterbrechungen gehört zu den schwierigsten Aufgaben im Betriebssystembau
  - Quelle der Asynchronität
    - gleichzeitig Segen und Fluch
  - Zustandssicherung auf Registerebene
    - Assemblerprogrammierung!
    - Abhängigkeit vom Übersetzer (z.B. flüchtige/nicht-flüchtige Register)
  - unterschiedliche Modelle (Prioritäten, u.s.w.)
- Zustandsänderungen in der Unterbrechungsbehandlung müssen wohl überlegt sein
  - kritische Abschnitte schützen
  - Fehler schwer zu finden (nicht verlässlich reproduzierbar!)

Betriebssysteme (BS)

VL 5 – Unterbrechungen, Synchronisation

Daniel Lohmann

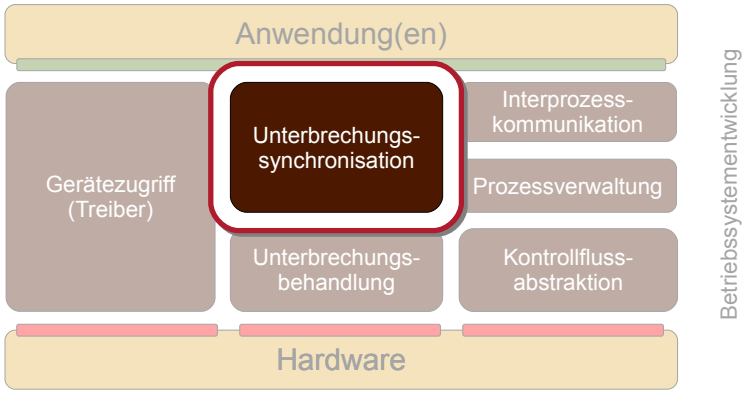
Lehrstuhl für Informatik 4
Verteilte Systeme und Betriebssysteme

Friedrich-Alexander-Universität
Erlangen Nürnberg

WS 15 – 10. November 2015

https://www4.cs.fau.de/Lehre/WS15/V_BS

Überblick: Einordnung dieser VL



Agenda

Agenda

Motivation: Konsistenzprobleme

Beispiel 1: Systemzeit

- hier schlummert möglicherweise ein Fehler ...
  - das Lesen von global_time erfolgt nicht notwendigerweise atomar!

```
32-Bit-CPU:
mov global_time, %eax
```

```
16-Bit-CPU (little endian):
mov global_time, %r0; lo
mov global_time+2, %r1; hi
```

- kritisch ist eine Unterbrechung zwischen den beiden Leseinstruktionen bei der 16-Bit-CPU

Instruktion	global_time hi / lo	Resultat r1 / r0
?	002A FFFF	? ?
mov global_time, %r0	002A FFFF	? FFFF
/* Inkrementierung */	002B 0000	? FFFF
mov global_time+2, %r1	002B 0000	002B FFFF

Beispiele aus der letzten Vorlesung

Beispiel 2: Ringpuffer

auch die Pufferimplementierung ist kritisch ...

Ausführung | Zustand

```
char consume() {
  int e
  if (e
  char
  nexto
void pr
  int e
  if (elements == SIZE) return;
  buf[nextin] = data;
  nextin++; nextin %= SIZE;
  occupied = elements + 1; // 2
}

  occupied = elements - 1; // 0
  return result; // 'a'
}
```

Problem:
Beim nächsten Aufruf von consume() hat occupied den Wert 0. Damit wird kein Ergebnis geliefert. Die Datenstruktur ist in einem inkonsistenten Zustand.

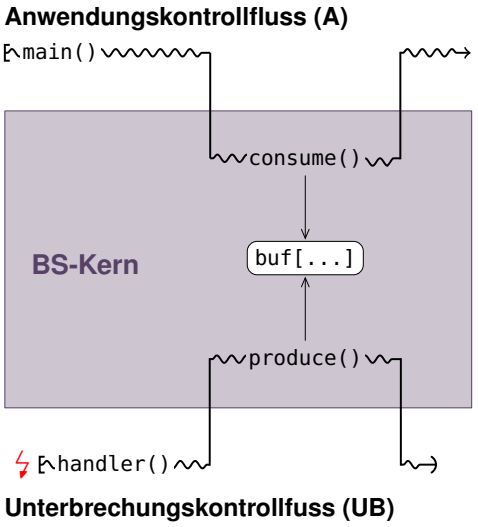
buf [0] [1] [2] occ. nextin nextout

Motivation: Ursache

Kontrollflüsse "von oben"

"begegnen" sich im Kern

und "von unten"



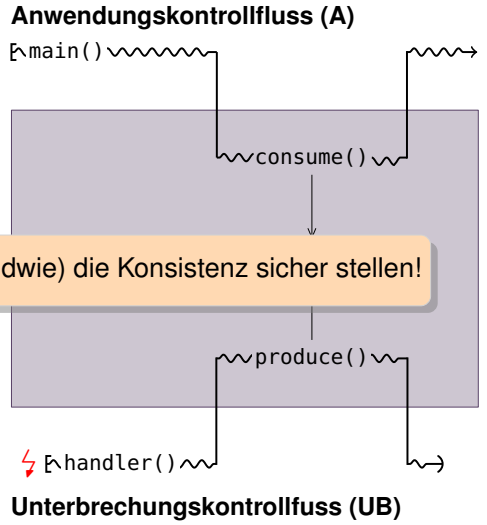
Motivation: Ursache

Kontrollflüsse "von oben"

"b
si

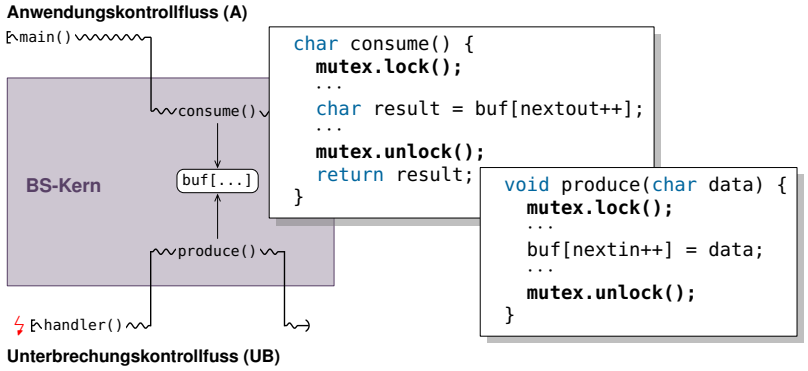
Wir müssen (irgendwie) die Konsistenz sicher stellen!

und "von unten"



Naiver Lösungsansatz

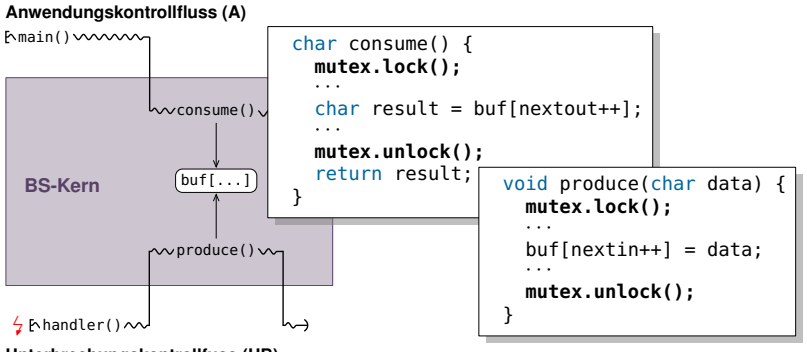
- Zweiseitige Synchronisation
  - gegenseitiger Ausschluss durch Mutex, *Spin-Lock*, ... (vgl. [SP])
  - wie zwischen zwei Prozessen



Naiver Lösungsansatz

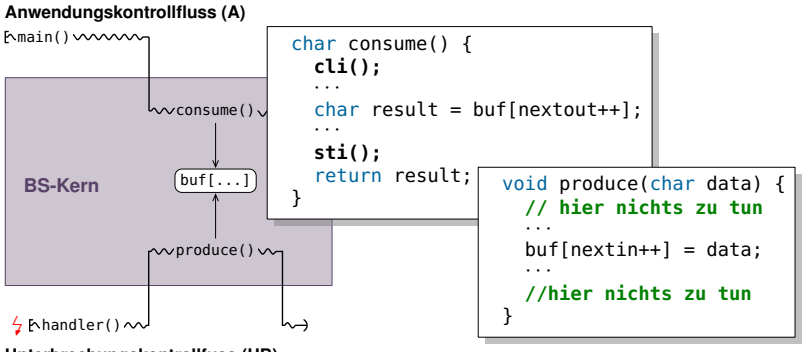
- Zweiseitige Synchronisation
  - gegenseitiger Ausschluss durch M
  - wie zwischen zwei Prozessen

Zweiseitige Synchronisation funktioniert **natürlich nicht!**



Besserer Lösungsansatz

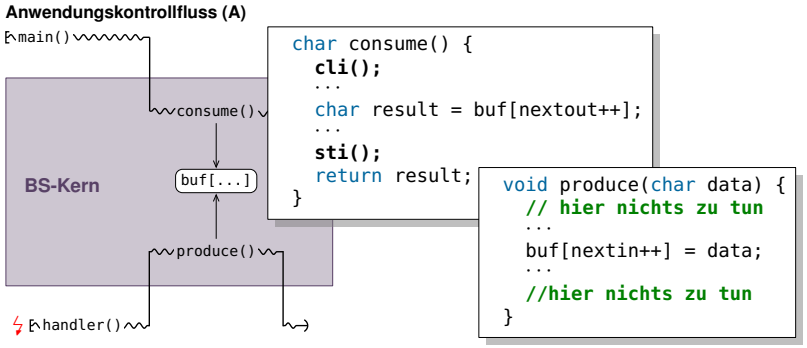
- Einseitige Synchronisation
  - Unterdrückung der Unterbrechungsbehandlung im Verbraucher
  - Operationen `disable_interrupts()` `enable_interrupts()`
    (im Folgenden o. B. d. A. in „Intel“-Schreibweise: `cli()` / `sti()`)



Besserer Lösungsansatz

- Einseitige Synchronisation
  - Unterdrückung der Unterbrechung
  - Operationen `disable_interrupts(`
    (im Folgenden o. B. d. A. in „Intel“-Schreibw

Einseitige Synchronisation funktioniert. [Warum?]



Erstes Fazit

- Konsistenzsicherung zwischen
  - Anwendungskontrollfluss (A) und
  - Unterbrechungsbehandlung (UB)

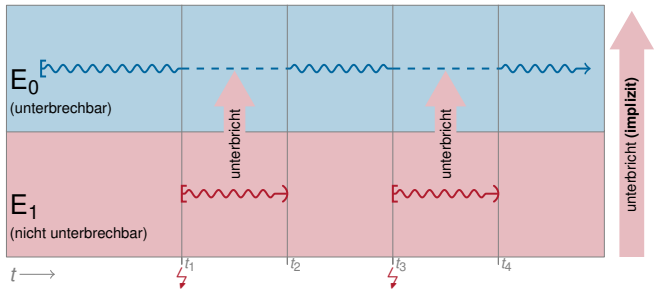
  muss anders erfolgen als zwischen Prozessen
- Die Beziehung zwischen A und UB ist **asymmetrisch**
  - Es handelt sich um „verschiedene Arten“ von Kontrollflüssen
  - UB *unterbricht* Anwendungskontrollfluss
    - implizit, an beliebiger Stelle
    - hat immer Priorität, läuft durch (*run-to-completion*)
  - A kann UB *unterdrücken* (besser: *verzögern*)
    - explizit, mit `cli` / `sti` (Grundannahme 5 aus VL 4)
- Synchronisation / Konsistenzsicherung erfolgt **einseitig**

Diese Tatsachen müssen wir **beachten**!
(Das heißt aber auch: Wir können sie **ausnutzen**)

Agenda

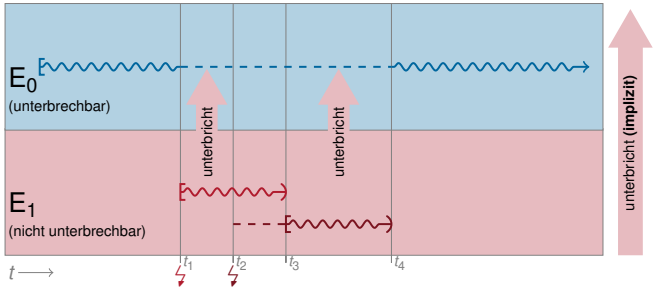
Prioritätsebenenmodell

- E_0 sei die Anwendungskontrollfluss-Ebene (A)
  - Kontrollflüsse dieser Ebene sind jederzeit unterbrechbar (durch E_1-Kontrollflüsse, implizit)
- E_1 sei die Unterbrechungsbehandlungs-Ebene (UB)
  - Kontrollflüsse dieser Ebene sind nicht unterbrechbar (durch $E_{0/1}$-Kontrollflüsse, implizit)



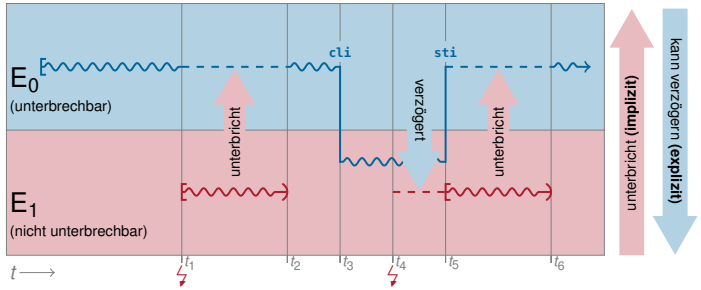
Prioritätsebenenmodell

- Kontrollflüsse derselben Ebene werden sequentialisiert
  - Sind mehrere Kontrollflüsse in einer Ebene anhängig, so werden diese nacheinander abgearbeitet (*run-to-completion*)
    - damit ist auf jeder Ebene höchstens ein Kontrollfluss aktiv
  - Die Sequentialisierungsstrategie selber ist dabei beliebig
    - FIFO, LIFO, nach Priorität, zufällig, ...
    - Für E_1-Kontrollflüsse auf dem PC implementiert der (A)PIC die Strategie



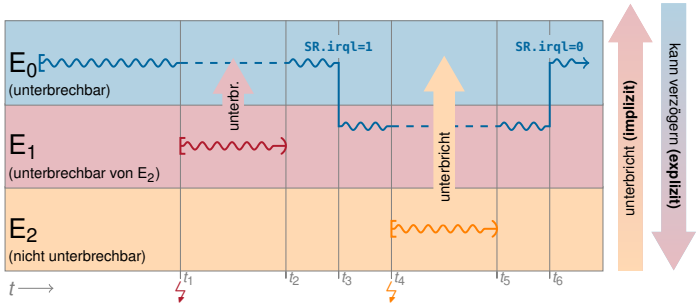
Prioritätsebenenmodell

- Kontrollflüsse können die Ebene wechseln
  - Mit `cli` wechselt ein E_0-Kontrollfluss explizit auf E_1
    - er ist ab dann nicht mehr unterbrechbar
    - andere E_1-Kontrollflüsse werden verzögert (⟷ Sequentialisierung)
  - Mit `sti` wechselt ein E_1-Kontrollfluss explizit auf E_0
    - er ist ab dann (wieder) unterbrechbar
    - anhängige E_1-Kontrollflüsse „schlagen durch“ (⟷ Sequentialisierung)



Prioritätsebenenmodell

- Verallgemeinerung für mehrere Unterbrechungsebenen:
  - Kontrollflüsse auf E_l werden
    1. jederzeit unterbrochen durch Kontrollflüsse von E_m (für $m > l$)
    2. nie unterbrochen durch Kontrollflüsse von E_k (für $k \leq l$)
    3. sequentialisiert mit weiteren Kontrollflüssen von E_l
  - Kontrollflüsse können die Ebene wechseln
    - durche spezielle Operationen (hier: Modifizieren des Statusregisters)



Prioritätsebenenmodell: Konsistenzsicherung

- Jede Zustandsvariable ist (logisch) genau einer Ebene E_l zugeordnet
  - Zugriffe aus E_l sind implizit konsistent (↔ Sequentialisierung)
  - Konsistenz bei Zugriff aus höheren / tieferen Ebenen muss explizit sichergestellt werden
- Maßnahmen zur Konsistenzsicherung bei Zugriffen:
  - „von oben“ (aus E_k mit $k < l$) durch harte Synchronisation
    - **explizit die Ebene auf E_l wechseln** beim Zugriff (Verzögerung)
    - damit erfolgt der Zugriff aus derselben Ebene (↔ Sequentialisierung)
  - „von unten“ (aus E_m mit $m > l$) durch weiche Synchronisation
    - **algorithmisch sicherstellen**, dass Unterbrechungen nicht stören
    - erfordert unterbrechungstransparente Algorithmen

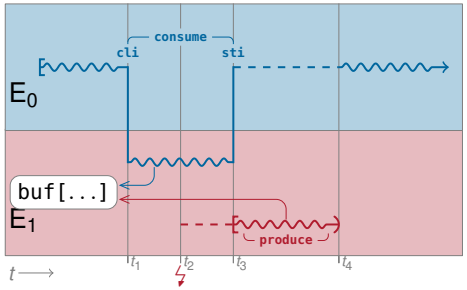
Agenda

Bounded Buffer – Lösung mit harter Synchronisation

Zugriff „von oben“ wird hart synchronisiert: Für die Ausführung von `consume()` wechselt der Kontrollfluss auf E_1

```
char consume() {
  cli();
  ...
  char result = buf[nextout++];
  ...
  sti();
  return result;
}
```

```
void produce(char data) {
  // hier nichts zu tun
  ...
  buf[nextin++] = data;
  ...
  //hier nichts zu tun
}
```



Zustand liegt (logisch) auf E_1

Harte Synchronisation: Bewertung

- **Vorteile**
  - Konsistenz ist sicher gestellt
    - auch bei komplexen Datenstrukturen und Zugriffsmustern
    - unabhängig davon, was der Compiler macht
  - einfach anzuwenden, „funktioniert immer“
    - im Zweifelsfall legt man einfach sämtlichen Zustand auf die höchstpriore Ebene
- **Nachteile**
  - Breitbandwirkung
    - Es werden pauschal alle Unterbrechungsbehandlungen (Kontrollflüsse) auf und unterhalb der Zustandsebene verzögert
  - Prioritätsverletzung
    - Es werden Kontrollflüsse höherer Priorität verzögert
  - prophylaktisches Verfahren
    - Nachteile werden in Kauf genommen, obwohl die Wahrscheinlichkeit, dass tatsächlich eine relevante Unterbrechung eintrifft, sehr klein ist.

Harte Synchronisation: Bewertung (Forts.)

- Ob die Nachteile erheblich sind, hängt ab von
  - Häufigkeit,
  - durchschnittlicher Dauer,
  - maximaler Dauer

  der Verzögerung.
- Kritisch ist vor allem die **maximale Dauer**
  - hat direkten Einfluss auf die anzunehmende Latenz
  - Wird die Latenz zu hoch, können Daten verloren gehen
    - *edge-triggered* Unterbrechungen gehen verloren
    - Daten werden zu langsam von EA-Gerät abgeholt

Fazit

Harte Synchronisation ist eher **ungeeignet** für die Konsistenzsicherung **komplexer Datenstrukturen**

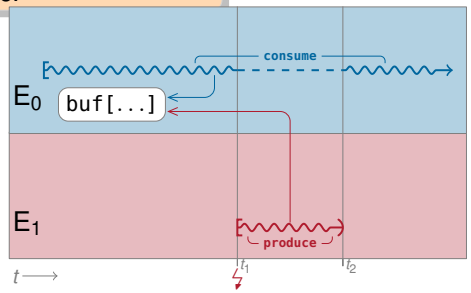
Agenda

Bounded Buffer – Ansatz mit weicher Synchronisation

Zugriff „von unten“ wird weich synchronisiert: `consume()` liefert ein korrektes Ergebnis, auch wenn während der Abarbeitung `produce()` ausgeführt wurde.

```
char consume() {
    ?
}
```

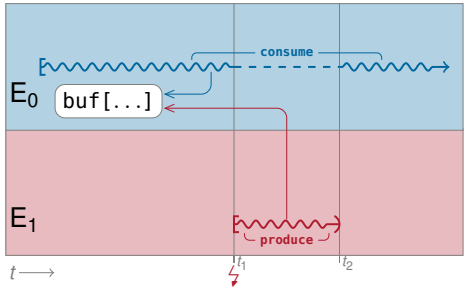
```
void produce(char data) {
    ?
}
```



Zustand liegt (logisch) auf E_0

Bounded Buffer – Konsistenzbedingungen, Annahmen

- Konsistenzbedingung
  - Ergebnis einer unterbrochenen Ausführung soll äquivalent sein zu dem einer sequentiellen Ausführung der Operation
    - *entweder* `consume()` vor `produce()` *oder* `consume()` nach `produce()`
- Annahmen
  - `produce()` unterbricht `consume()`
    - alle anderen Kombinationen kommen nicht vor
  - `produce()` läuft immer durch (*run-to-completion*)



Bounded Buffer – Implementierung aus der letzten VL

Kritisch ist der gemeinsam verwendete Zustand

```
// Pufferklasse in C++
class BoundedBuffer {
  char buf[SIZE]; int occupied; int nextin, nextout;
public:
  BoundedBuffer(): occupied(0), nextin(0), nextout(0) {}
  void produce(char data) {    // Unterbrechungsbehandlung:
    int elements = occupied;       // Elementzaehler merken
    if (elements == SIZE) return; // Element verloren
    buf[nextin] = data;            // Element schreiben
    nextin++; nextin %= SIZE;      // Zeiger weitersetzen
    occupied = elements + 1;       // Zaehler erhoehen
  }
  char consume() {             // normaler Kontrollfluss:
    int elements = occupied;       // Elementzaehler merken
    if (elements == 0) return 0;  // Puffer leer, kein Ergebnis
    char result = buf[nextout];    // Element lesen
    nextout++; nextout %= SIZE;    // Lesezeiger weitersetzen
    occupied = elements - 1;       // Zaehler erniedrigen
    return result;                 // Ergebnis zurueckliefern
} };
```


Bounded Buffer – Implementierung aus der letzten VL

Kritisch ist der gemeinsam verwendete Zustand

Insbesondere Zustand, auf den von beiden Seiten **schreibend** zugegriffen wird.

```
// Pufferklasse in C++
class BoundedBuffer {
  char buf[SIZE]; int occupied; int nexti
public:
  BoundedBuffer(): occupied(0), nextin(0), nextout(0) {}
  void produce(char data) {    // Unterbrechungsbehandlung:
    int elements = occupied;       // Elementzaehler merken
    if (elements == SIZE) return; // Element verloren
    buf[nextin] = data;            // Element schreiben
    nextin++; nextin %= SIZE;      // Zeiger weitersetzen
    occupied = elements + 1;       // Zaehler erhoehen
  }
  char consume() {             // normaler Kontrollfluss:
    int elements = occupied;       // Elementzaehler merken
    if (elements == 0) return 0;  // Puffer leer, kein Ergebnis
    char result = buf[nextout];    // Element lesen
    nextout++; nextout %= SIZE;    // Lesezeiger weitersetzen
    occupied = elements - 1;       // Zaehler erniedrigen
    return result;                 // Ergebnis zurueckliefern
} };
```


Bounded Buffer – Alternative Implementierung

Diese alternative Implementierung kommt ohne gemeinsam beschriebenen Zustand aus.

```
// Pufferklasse in C++ (alternativ)
class BoundedBuffer {
  char buf[SIZE]; int nextin, nextout;
public:
  BoundedBuffer(): nextin(0), nextout(0) {}
  void produce(char data) {

    if ((nextin + 1) % SIZE == nextout) return;
    buf[nextin] = data;
    nextin = (nextin + 1) % SIZE;

  }
  char consume() {

    if (nextout == nextin) return 0;
    char result = buf[nextout];
    nextout = (nextout + 1) % SIZE;

    return result;
} };
```

Bounded Buffer – Alternative Implementierung

```
// Pufferklasse in C++ (alternativ)
class BoundedBuffer {
  char buf[SIZE]; int nextin, nextout;
public:
  BoundedBuffer(): nextin(0), nextout(0) {}
  void produce(char data) {

    if ((nextin + 1) % SIZE == nextout) return;
    buf[nextin] = data;
    nextin = (nextin + 1) % SIZE;

  }
  char consume() {

    if (nextout == nextin) return 0;
    char result = buf[nextout];
    nextout = (nextout + 1) % SIZE;

    return result;
} };
```

Allerdings gibt es hier jetzt Zustand, der von einer Seite gelesen und von der jeweils anderen beschrieben wird.

An genau diesen Stellen müssen wir prüfen, ob die Konsistenzbedingung gilt.

Bounded Buffer – Analyse der neuen Implementierung

- Angenommen, die Unterbrechung von `consume()` erfolgt:
  - aus der Sicht von `consume()`
    - vor dem Lesen von **`nextin`** ⇔ `consume()` nach `produce()` ✔
    - nach dem Lesen von **`nextin`** ⇔ `consume()` vor `produce()` ✔
  - aus der Sicht von `produce()`
    - vor dem Schreiben von **`nextout`** ⇔ `produce()` vor `consume()` ✔
    - nach dem Schreiben von **`nextout`** ⇔ `produce()` nach `consume()` ✔

```
char consume() {
  if (nextout == nextin) return 0;
  char result = buf[nextout];
  nextout = (nextout + 1) % SIZE;
  return result;
}
```

```
void produce(char data) {
  if ((nextin + 1) % SIZE == nextout) return;
  buf[nextin] = data;
  nextin = (nextin + 1) % SIZE;
}
```

Konsistenzbedingung ist in jedem Fall erfüllt!

Systemzeit – Implementierung aus der letzten Vorlesung

```
/* globale Zeitvariable */
extern volatile time_t global_time;
```

```
/* Systemzeit abfragen */
time_t time () {
  return global_time;
}
```

```
/* Unterbrechungs- *
 * behandlung      */
void timerHandler () {
  global_time++;
}
```

`h8300-hms-g++` (16-Bit-Architektur)

```
time:
  mov global_time, %r0; lo
  mov global_time+2, %r1; hi
  ret
```

Problem:
Daten werden nicht atomar gelesen.

Systemzeit – Konsistenzbedingungen, Annahmen, Ansatz

- Konsistenzbedingung
  - Ergebnis einer unterbrochenen Ausführung soll äquivalent sein zu dem einer sequentiellen Ausführung der Operation
    - *entweder* `time()` vor `timerHandler()` *oder umgekehrt*
- Annahmen
  - `timerHandler()` unterbricht `time()`
    - alle anderen Kombinationen kommen nicht vor
  - `timerHandler()` läuft immer durch (*run-to-completion*)
- Lösungsansatz: In `time()` optimistisch herangehen
  1. lese Daten unter der Annahme nicht unterbrochen zu werden
  2. überprüfe, ob Annahme zutraf – wurden wir unterbrochen?
  3. falls unterbrochen, setze neu auf ab Schritt 1

Systemzeit – Neue Implementierung

```
/* globale Zeitvariable */
extern volatile time_t global_time;
extern volatile bool interrupted;
```

```
/* Systemzeit abfragen */
time_t time () {
  time_t res;
  do {
    interrupted = false;
    res = global_time;
  } while (interrupted);
  return res;
}
```

```
/* Unterbrechungsbehandlung */
void timerHandler () {
  interrupted = true;
  global_time++;
}
```

Konsistenzbedingung ist nun in jedem Fall erfüllt!

Weiche Synchronisation: Bewertung

- **Vorteile**
  - Konsistenz ist sichergestellt (durch Unterbrechungstransparenz)
  - Priorität wird nie verletzt
    - Kontrollflüsse der höherprioren Ebenen kommen immer durch
  - Kosten entstehen entweder gar nicht oder nur im Konfliktfall
    - gar nicht ⇝ Beispiel Bounded Buffer
    - im Konfliktfall ⇝ optimistische Verfahren, Beispiel Systemzeit (zusätzliche Kosten durch Wiederaufsetzen)
- **Nachteile**
  - Lösungen häufig sehr komplex
    - Wenn man überhaupt eine Lösung findet, ist diese in der Regel schwer zu verstehen – und noch schwieriger zu verifizieren
  - Lösungen häufig sehr fragil (bezüglich Randbedingungen)
    - Kleinste Änderungen können die Konsistenzgarantie zerstören
    - Codegenerierung des Compilers ist zu beachten
  - Bei größeren Datenmengen steigen die Wiederaufsetzkosten

Weiche Synchronisation: Bewertung (Forts.)

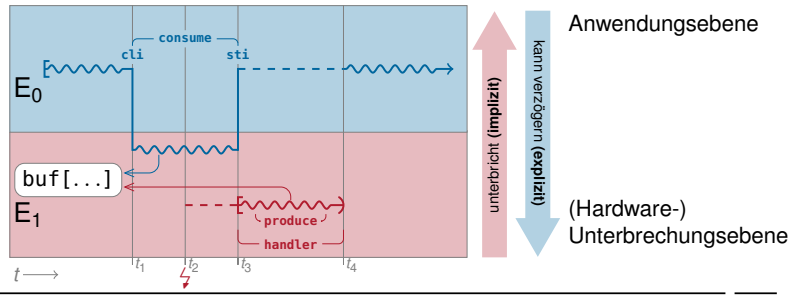
Fazit

- Weiche Synchronisation durch Unterbrechungstransparenz ist **grundsätzlich erstrebenswert**!
- Es handelt sich bei den Algorithmen jedoch immer um **Speziallösungen** für **Spezialfälle**.
- Als allgemein verwendbares Mittel für die Sicherung **beliebiger Datenstrukturen** ist sie **nicht geeignet**.

Agenda

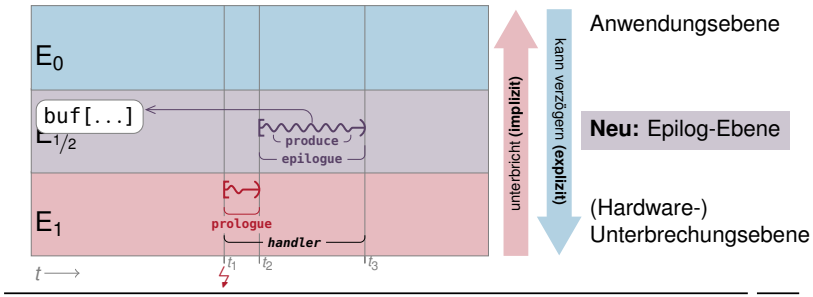
Prolog/Epilog-Modell – Motivation

- **Reprise:** Harte Synchronisation
  - einfach, korrekt, „funktioniert immer“ ✔
  - Hauptproblem ist die hohe Latenz ✘
    - Verzögerung bei Zugriff auf den Zustand aus höheren Ebenen
    - Verzögerung bei Bearbeitung des Zustands in der UB selbst
  - letztlich dadurch verursacht, dass der Zustand (logisch) auf der/einer Hardwareunterbrechungsebene $E_{1\ldots n}$ liegt.



Prolog/Epilog-Modell – Ansatz

- **Ansatz:** Latenzverbergung durch zusätzliche Ebene
  - Wir fügen eine weitere *logische* Ebene ein: $E_{1/2}$
    - $E_{1/2}$ liegt zwischen der Anwendungsebene E_0 und den UB-Ebenen $E_{1\ldots n}$
  - Unterbrechungsbehandlung wird zweigeteilt in Prolog und Epilog
    - Prolog arbeitet auf Unterbrechungsebene $E_{1\ldots n}$
    - Epilog arbeitet auf der neuen (Software-)Ebene $E_{1/2}$ (Epilogebene)
  - Zustand liegt (so weit wie möglich) auf der Epilogebene
    - eigentliche Unterbrechungsbehandlung wird nur noch kurz gesperrt



Prolog/Epilog-Modell – Ansatz (Forts.)

- Unterbrechungsbehandlungsroutinen werden zweigeteilt
  - beginnen im **Prolog** (immer)
  - werden fortgesetzt im **Epilog** (bei Bedarf)
- **Prolog** (⇝ Hardwareunterbrechung)
  - läuft auf Hardwareunterbrechungsebene
    - hat damit Priorität über Anwendungsebene und Epilogebene
  - ist **kurz**, fasst wenig oder gar keinen Zustand an
    - Üblicherweise wird nur der Hardware-Zustand gesichert und bestätigt
    - Unterbrechungen bleiben nur kurz gesperrt (⇝ Latenzminimierung)
    - kann bei Bedarf einen Epilog für die weitere Verarbeitung anfordern
- **Epilog** (⇝ Softwareunterbrechung)
  - läuft auf Epilogebene $E_{1/2}$ (zusätzliche Kontrollflussebene)
    - Ausführung erfolgt verzögert zum Prolog
    - erledigt die eigentliche Arbeit (⇝ Latenzverbergung)
  - hat Zugriff auf größten Teil des Zustands
    - Zustand wird auf Epilogebene synchronisiert

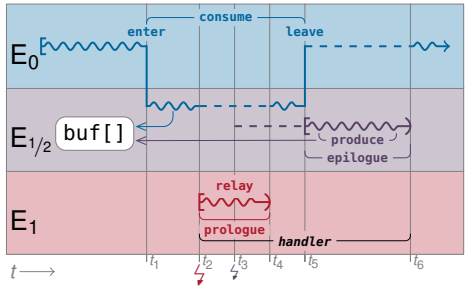
Prolog/Epilog-Modell – Epilogebene

- Die Epilogebene wird (ganz oder teilweise) in Software implementiert
  - trotzdem handelt es sich um eine ganz normale Prioritätsebene des Ebenenmodells
  - es müssen daher auch dieselben Gesetzmäßigkeiten gelten
- Es gilt: Kontrollflüsse auf der Epilogebene $E_{1/2}$ werden
  1. jederzeit unterbrochen durch Kontrollflüsse der Ebenen $E_{1\ldots n}$
     ⇝ Prologe (Unterbrechungen) haben Priorität über Epiloge
  2. nie unterbrochen durch Kontrollflüsse der Ebene E_0
     ⇝ Epiloge haben Priorität über Anwendungskontrollflüsse
  3. sequentialisiert mit anderen Kontrollflüssen von $E_{1/2}$
     ⇝ Anhängige Epiloge werden nacheinander abgearbeitet.
     ⇝ Bei Rückkehr zur Anwendungsebene sind alle Epiloge abgearbeitet.

Prolog/Epilog-Modell – Implementierung

- Benötigt werden Operationen, um
  1. explizit die Epilogebene zu betreten: **`enter()`**
     - entspricht dem `cli` bei der harten Synchronisation
  2. explizit die Epilogebene zu verlassen: **`leave()`**
     - entspricht dem `sti` bei der harten Synchronisation
  3. einen Epilog anzufordern: **`relay()`**
     - entspricht dem Hochziehen der IRQ-Leitung beim PIC

Prolog/Epilog-Modell – Ablaufbeispiel



E_1-Unterbrechungen werden nie gesperrt.

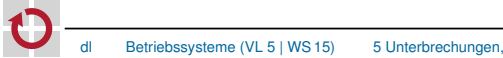
Aktivierungslatenz der Unterbrechungsbehandlung ist minimal.

- t_1 Anwendungskontrollfluss betritt Epilogebene $E_{1/2}$ (`enter()`).
- t_2 Unterbrechung ↯ auf Ebene E_1 wird signalisiert ⇝ Prolog wird ausgeführt.
- t_3 Prolog fordert Epilog für die nachgeordnete Bearbeitung an (`relay()` ↯).
- t_4 Prolog terminiert, unterbrochener $E_{1/2}$-Kontrollfluss läuft weiter.
- t_5 Anwendungskontrollfluss verlässt die Epilogebene $E_{1/2}$ (`leave()`) ⇝ zwischenzeitlich aufgelaufene Epiloge werden nun abgearbeitet.
- t_6 Epilog terminiert, Anwendungskontrollfluss fährt auf E_0 fort.

Prolog/Epilog-Modell – Implementierung

- Benötigt werden Operationen, um
  1. explizit die Epilogebene zu betreten: **`enter()`**
     - entspricht dem `cli` bei der harten Synchronisation
  2. explizit die Epilogebene zu verlassen: **`leave()`**
     - entspricht dem `sti` bei der harten Synchronisation
  3. einen Epilog anzufordern: **`relay()`**
     - entspricht dem Hochziehen der IRQ-Leitung beim PIC
- Außerdem Mechanismen, um
  4. anhängige Epiloge zu „merken“: **`queue`** (z. B.)
     - entspricht dem IRR (Interrupt-Request-Register) beim PIC
  5. sicherzustellen, dass anhängige Epiloge abgearbeitet werden
     - entspricht bei der harten Synchronisation dem Protokoll zwischen CPU und PIC

Dieser Punkt muss etwas genauer betrachtet werden!



Prolog/Epilog-Modell – Implementierung

5. sicherzustellen, dass anhängige Epiloge abgearbeitet werden
   - entspricht bei der harten Synchronisation dem Protokoll zwischen CPU und PIC

Wann müssen anhängige Epiloge abgearbeitet werden?

Immer unmittelbar, bevor die CPU auf E_0 zurückkehrt!

1. bei explizitem Verlassen der Epilogebene mit `leave()`
   - während der Anwendungskrontrollfluss auf $E_{1/2}$ gearbeitet hat könnten Epiloge aufgelaufen sein (↩ Sequentialisierung).
2. nach Abarbeitung des letzten Epilogs
   - während der Epilogabarbeitung könnten weitere Epiloge aufgelaufen sein (↩ Sequentialisierung).
3. wenn der **letzte** Unterbrechungsbehandler terminiert
   - während der Abarbeitung von $E_{1\ldots n}$-Kontrollflüssen könnten Epiloge aufgelaufen sein (↩ Priorisierung).

Prolog/Epilog-Modell – Implementierung

- Implementierungsvarianten
  - rein softwarebasiert (⇝ **Übung**)
  - mit Hardwareunterstützung durch einen AST (⇝ [2, 3])
- Ein AST (*asynchronous system trap*) ist eine Unterbrechung, die (nur) durch Software angefordert werden kann.
  - z. B. durch Setzen eines Bits in einem bestimmten Register
  - ansonsten technisch vergleichbar mit einer Hardware-Unterbrechung
    - AST wird (im Gegensatz zu Traps/Exceptions) asynchron abgearbeitet
    - AST läuft auf eigener Unterbrechungsebene zwischen der Anwendungsebene und den Hardware-UBs (↪ unsere $E_{1/2}$)
    - Gesetzmäßigkeiten des Ebenenmodels gelten (AST-Ausführung ist verzögerbar, wird automatisch aktiviert, . . .)
- Sicherstellung der Epilogabarbeitung wird damit sehr einfach!
  - Abarbeitung der Epiloge erfolgt im AST
    ⇝ und damit automatisch, bevor die CPU auf E_0 zurückkehrt
  - bleibt nur noch die Verwaltung der anhängigen Epiloge

Prolog/Epilog-Modell – Implementierung

- Beispiel TriCore: Implementierung mit AST
  - AST hier als Unterbrechung der E_1 konfiguriert (⇔ unsere $E_{1/2}$)
  - Geräteunterbrechungen laufen auf $E_{2...n}$

```
void enter() {
  CPU::setIRQL(1);          // betrete E1, verzoegere AST
}
void leave() {
  CPU::setIRQL(0);          // erlaube AST (anhaengiger
}                           // AST wuerde jetzt abgearbeitet)
void relay(<Epilog>) {
  <haenge Epilog an queue an>
  CPU_SRC1::trigger();      // aktiviere Level-1 IRQ (AST)
}
void __attribute__((interrupt_handler)) irq1Handler() {
  while(<Epilog in queue>) {
    <entferne Epilog aus queue>
    <arbeite Epilog ab>
} }
```

Prolog/Epilog-Modell – Implementierung

- Beispiel TriCore: Implementierung mit AST
  - AST hier als Unterbrechung der E_1 konfiguriert (⇔ unsere $E_{1/2}$)
  - Geräteunterbrechungen laufen auf $E_{2...n}$

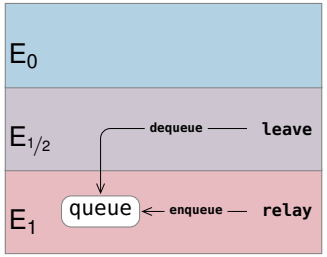
```
void enter() {
  CPU::setIRQL(1);          // b
}
void leave() {
  CPU::setIRQL(0);          // e
}                           // A
void relay(<Epilog>) {
  <haenge Epilog an queue an>
  CPU_SRC1::trigger();      // aktiviere Level-1 IRQ (AST)
}
void __attribute__((interrupt_handler)) irq1Handler() {
  while(<Epilog in queue>) {
    <entferne Epilog aus queue>
    <arbeite Epilog ab>
} }
```

Bietet die Hardware (wie z. B. IA-32) kein AST-Konzept, so kann man dieses in Software nachbilden.

Näheres dazu in der Übung.

Prolog/Epilog-Modell – Ziel erreicht?

- Kernzustand kann jetzt auf Epilogebene verwaltet und synchronisiert werden.
  - Hardware-UBs müssen nicht (mehr) gesperrt werden!
- Ein Problem bleibt noch: Die Epilog-Warteschlange
  - Zugriff erfolgt aus Prologen und der Epilogebene
    - muss also entweder hart synchronisiert werden (im Bild)
    - oder man sucht eine Speziallösung mit weicher Synchronisation



Harte Synchronisation erscheint hier akzeptabel, da die Sperrzeit (⇔ Ausführungszeit von `dequeue()`) kurz und deterministisch ist.

Eine Lösung mit weicher Synchronisation (z. B. [7]) wäre natürlich schöner!

Prolog/Epilog-Modell: Bewertung

- **Vorteile**
  - Konsistenz ist sichergestellt (durch Synchronisation auf Epilogebene)
  - Programmiermodell entspricht dem (einfach verständlichen) Modell der harten Synchronisation
  - Auch komplexer Zustand kann synchronisiert werden
    - ohne das dabei Unterbrechungsanforderungen verloren gehen
    - ermöglicht es, den gesamte BS-Kern auf Epilogebene zu schützen
- **Nachteile**
  - Zusätzliche Ebene führt zu zusätzlichem Overhead
    - Epilogaktivierung könnte länger dauern als direkte Behandlung
    - Komplexität für den BS-Entwickler wird erhöht
  - Unterbrechungsperren lassen sich nicht vollständig vermeiden
    - Gemeinsamer Zustand von Pro- und Epilog muss weiter hart oder weich synchronisiert werden

Prolog/Epilog-Modell: Bewertung (Forts.)

Fazit

- Das Prolog/Epilog-Modell ist ein **guter Kompromiss** für die Synchronisation des Kernzustands.
- Es ist auch für die Konsistenzsicherung **komplexer Datenstrukturen geeignet**

Prolog/Epilog-Modell – Verwandte Konzepte

- UNIX: top/bottom half [4]
  - Aktivitäten der bottom half ($\mapsto E_1$) sind asynchron zu den Aktivitäten der top half ($\mapsto E_{1/2}$) und dürfen keine Systemfunktionen aufrufen
- Windows: ISRs / deferred procedure calls (DPCs) [8]
  - Unterbrechungsbehandler ($\mapsto$ Prologe) können DPCs ($\mapsto$ Epiloge) in eine Warteschlange einhängen. Diese wird verzögert abgearbeitet, bevor die CPU auf Faden-Ebene zurückkehrt
- Linux: top halves / bottom halves, tasklets, irq threads [1, 5, 6]
  - Klassisch: Unterbrechungsbehandler (ISR) setzt Bit, durch das eine verzögerte bottom half (BH $\mapsto$ Epilog) angefordert werden kann.
  - Aktuell: BH $\mapsto$ Softirqs, dazu kommen tasklets (vgl. mit Windows DPCs) und interrupt threads.
- eCos: ISRs / deferred service routines (DSRs)
- ...

Nahezu alle Betriebssysteme, die Unterbrechungsbehandlung verwenden, bieten auch eine „Epilogebene“.

Agenda

Zusammenfassung: Unterbrechungssynchronisation

- Konsistenzsicherung im BS-Kern
  - muss anders erfolgen als zwischen Prozessen – einseitig
  - Kontrollflüsse arbeiten auf verschiedenen Prioritätsebenen
- Maßnahmen zur Konsistenzsicherung
  - harte Synchronisation (durch Unterbrechungssperren)
    - einfach, jedoch negative Auswirkungen auf Latenz
    - Unterbrechungsanforderungen können verloren gehen
  - weiche Synchronisation (durch Unterbrechungstransparenz)
    - gut und effizient, jedoch nur in Spezialfällen möglich
    - Implementierung kann sehr komplex werden
  - Prolog/Epilog-basierte Synchronisation (Zweiteilung der Unterbrechungsbehandlung)
    - guter Kompromiss
    - Stand der Technik in heutigen Betriebssystemen

Nachtrag: Mehrkernsysteme

Beachte: Unterbrechungsbehandlung $\neq$ Parallelität

- Techniken funktionieren (so) nur bei echter Unterbrechungssemantik: A und UB werden auf **demselben** Prozessor ausgeführt
- Wird die UB „echt parallel" (auf einem weiteren Prozessor) ausgeführt, kommt es zu Problemen
  - Annahmen des Prioritätsebenenmodells gelten nicht mehr! (Sequentialisierung, Priorisierung, *run-to-completion*)
  - Asymmetrie (UB unterbricht A) ist nicht länger gegeben (weiche Synchronisation wird dadurch viel schwieriger)
- Zusätzlich erforderlich: **Interprozessor-Synchronisation**
  - „hart" $\mapsto$ zweiseitig blockierend, z. B. mit *Spin-Locks* $\rightsquigarrow$ Übung
  - „weich" $\mapsto$ algorithmisch nichtblockierend (**schwer!**) $\rightsquigarrow$ [CS]

Referenzen

[1] Daniel P. Bovet und Marco Cesati. *Understanding the Linux Kernel*. O'Reilly, 2001. ISBN: 0-596-00002-2.

[2] Digital Equipment Corporation. *VAX-11 Architecture Reference Manual*. Document Number EK-VAXAR-RM-001. Digital Equipment Corporation. Maynard, MA, USA: Digital Press, Mai 1982.

[3] Samuel J. Leffler, Marshall Kirk McKusick, Michael J. Karels u. a. *The Design and Implementation of the 4.3 BSD UNIX Operating System*. Addison-Wesley, Mai 1989. ISBN: 0-201-06196-1.

[4] John Lions. *Lions' Commentary on UNIX (6th Edition)*. Peer-to-Peer Communications Inc., 1977. ISBN: 978-1573980135.

[5] Robert Love. *Linux Kernel Development (2nd Edition)*. Novell Press, 2005. ISBN: 978-0672327209.

[6] Valentin Rothberg. *Interrupt Handling in Linux*. Technical Report. Friedrich-Alexander-Universität Erlangen-Nürnberg, Lehrstuhl für Informatik 4, 2015. URL: `https://www4.cs.fau.de/~vrothberg/Interrupt_Handling_in_Linux.pdf`.

Referenzen (Forts.)

[7] Friedrich Schön, Wolfgang Schröder-Preikschat, Olaf Spinczyk u. a. „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)*. (Newport Beach, CA, USA). IEEE Computer Society Press, März 2000, S. 270–277. DOI: `10.1109/ISORC.2000.839540`.

[CS] Wolfgang Schröder-Preikschat. *Concurrent Systems*. Vorlesung mit Übung. Friedrich-Alexander-Universität Erlangen-Nürnberg, Lehrstuhl für Informatik 4, 2015 (jährlich). URL: `https://www4.cs.fau.de/Lehre/WS15/V_CS`.

[SP] Wolfgang Schröder-Preikschat. *Systemprogrammierung*. Vorlesung mit Übung. Friedrich-Alexander-Universität Erlangen-Nürnberg, Lehrstuhl für Informatik 4, 2015 (jährlich). URL: `https://www4.cs.fau.de/Lehre/WS15/V_SP`.

[8] David A. Solomon und Mark Russinovich. *Inside Microsoft Windows 2000 (3rd Edition)*. Microsoft Press, 2000. ISBN: 3-86063-630-8.