© 2023 A. Lochmann (TUDO), 2022 P. Ulbrich (TUDO), 2021-16 V.r Sieh (FAU), 2016-07 D. Lohmann (LUHH), 2006 O. Spinczyk (UOS)

Betriebssystembau (BSB)

VL 4 – Unterbrechungen, Software

Alexander Krause

Informatik-Rechner-Betriebsgruppe / LS12 – Arbeitsgruppe Systemsoftware
Technische Universität Dortmund

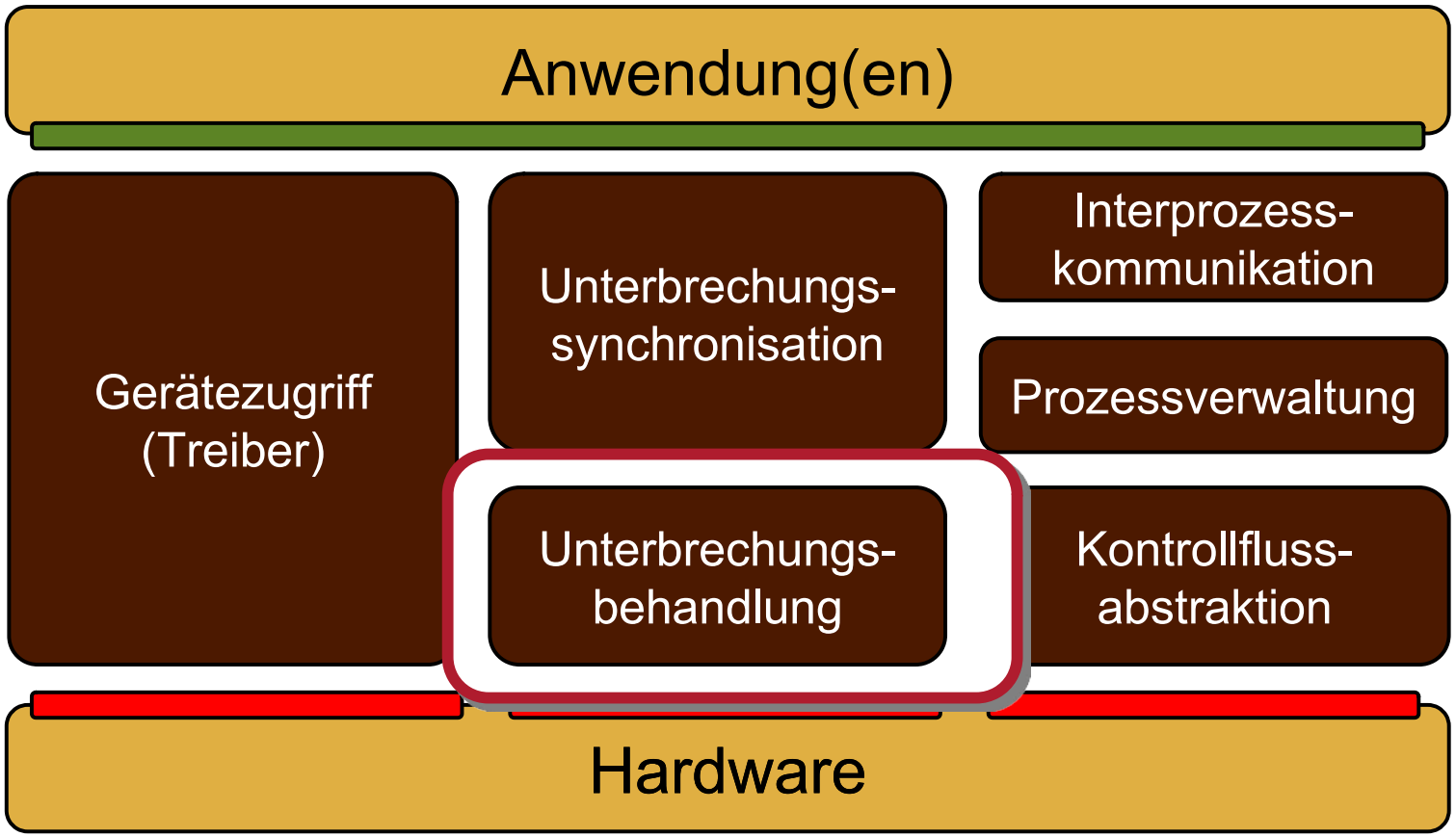
`https://irb.cs.tu-dortmund.de/lehre/ws26/bsb`

WS 26 – 03. November 2026

technische universität dortmund

arbeitsgruppe systemsoftware

Überblick: Einordnung dieser VL



Betriebssystementwicklung

Agenda

- Einordnung
- Begriffe und Grundannahmen
- Zustandssicherung
- Zustandsänderungen
- Zusammenfassung



Agenda

Einordnung

Begriffe und Grundannahmen

- Interrupt, Exception, Trap
- Grundannahmen

Zustandssicherung

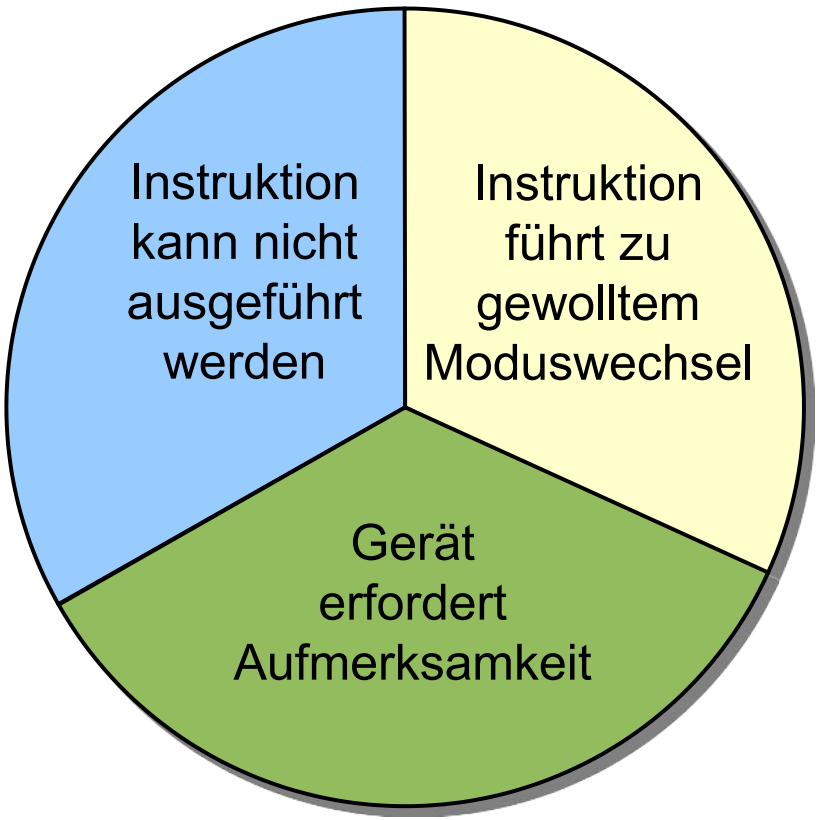
Zustandsänderungen

Zusammenfassung



Begriffe

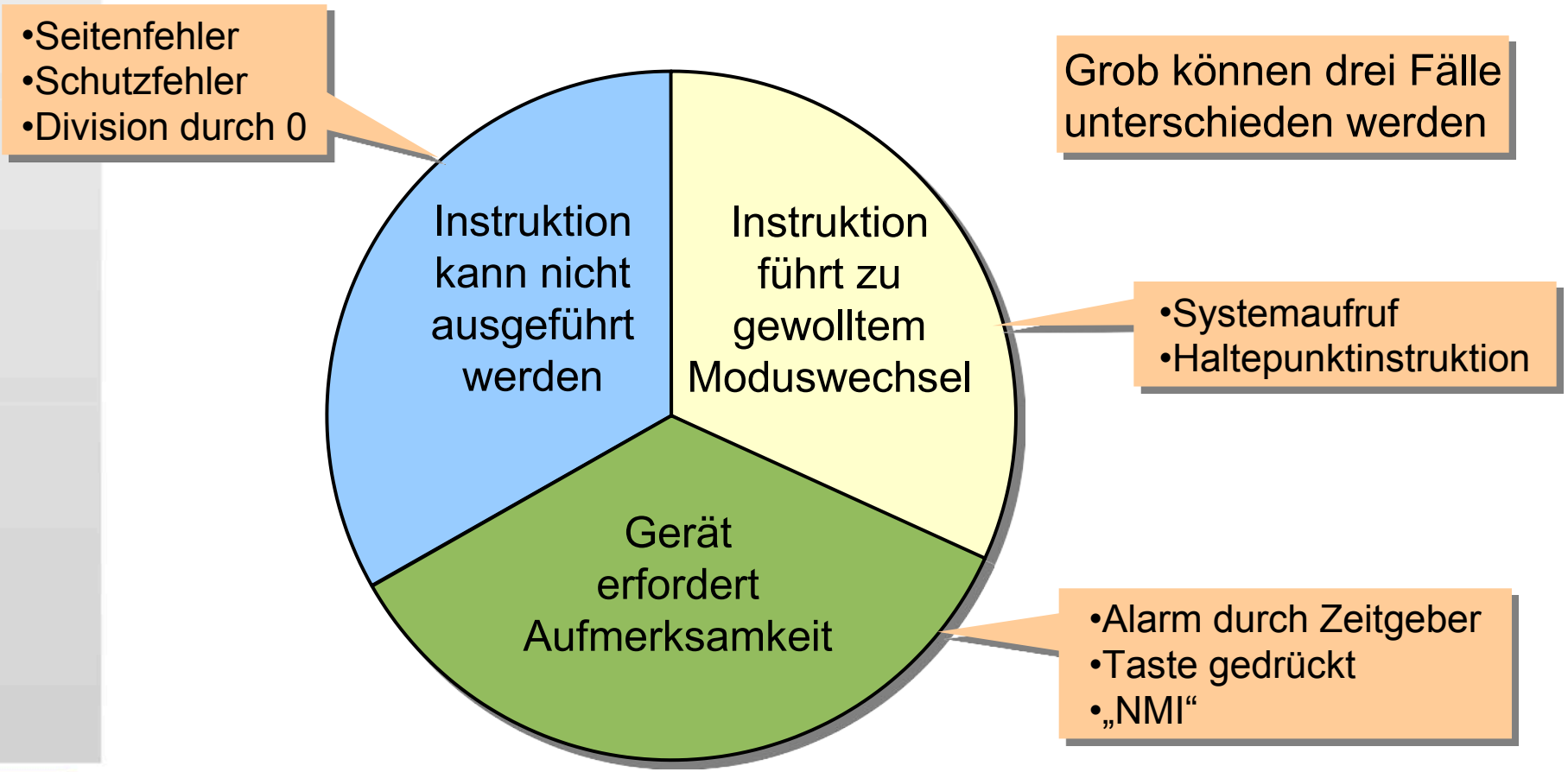
- das Verständnis der Begriffe ist unterschiedlich ...
  - zwecks Klärung begeben wir uns auf die technische Ebene



Grob können drei Fälle unterschieden werden

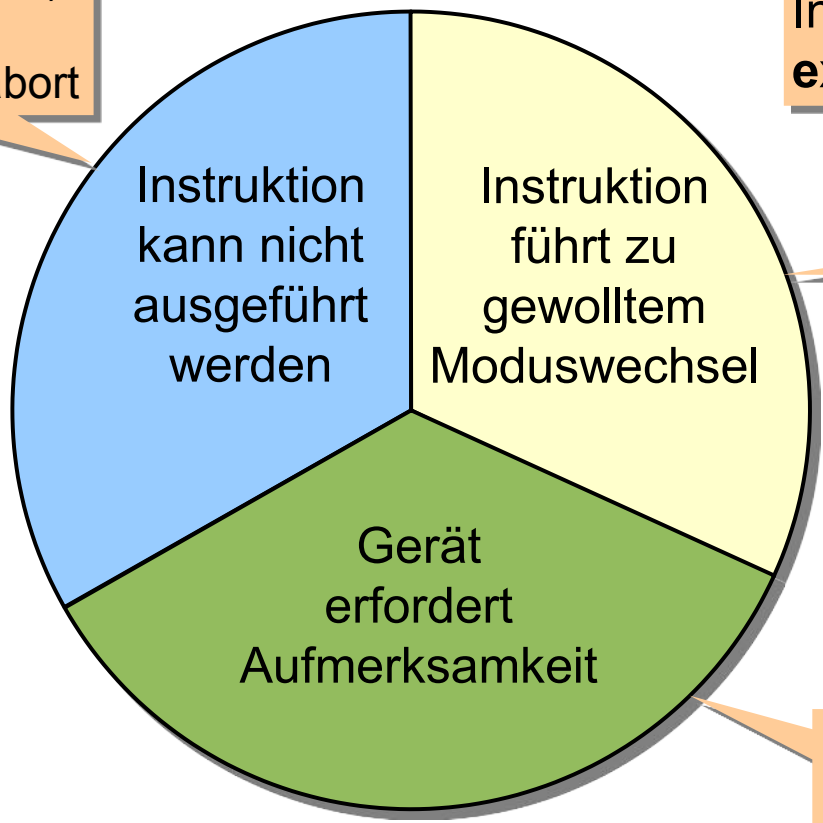
Begriffe

- 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



(software-generated) **exceptions**
• fault, trap oder abort

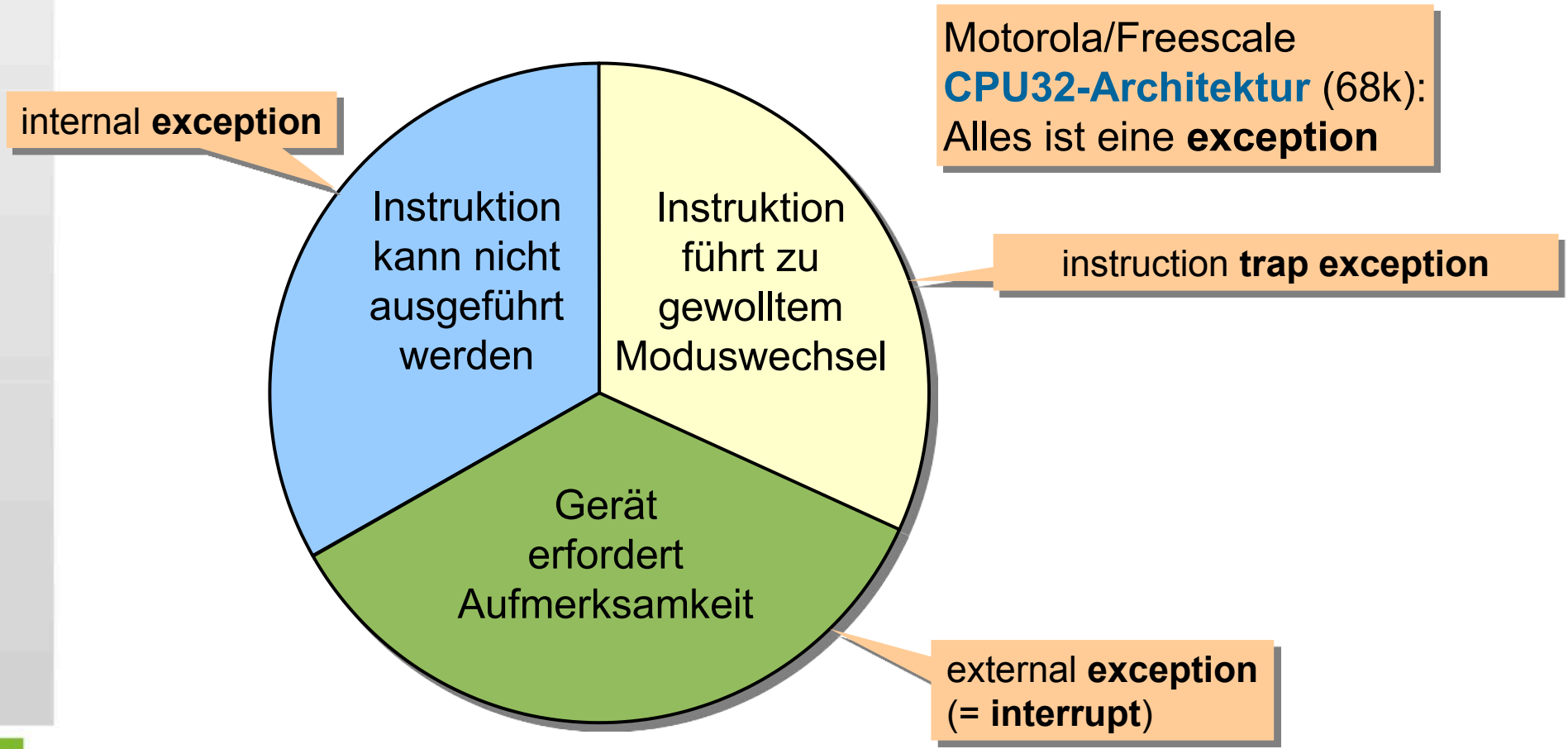
Intel **IA-32-Architektur** (x86): **exceptions** und **interrupts**

software(-generated) **interrupts**

external (hardware-generated) **interrupts**

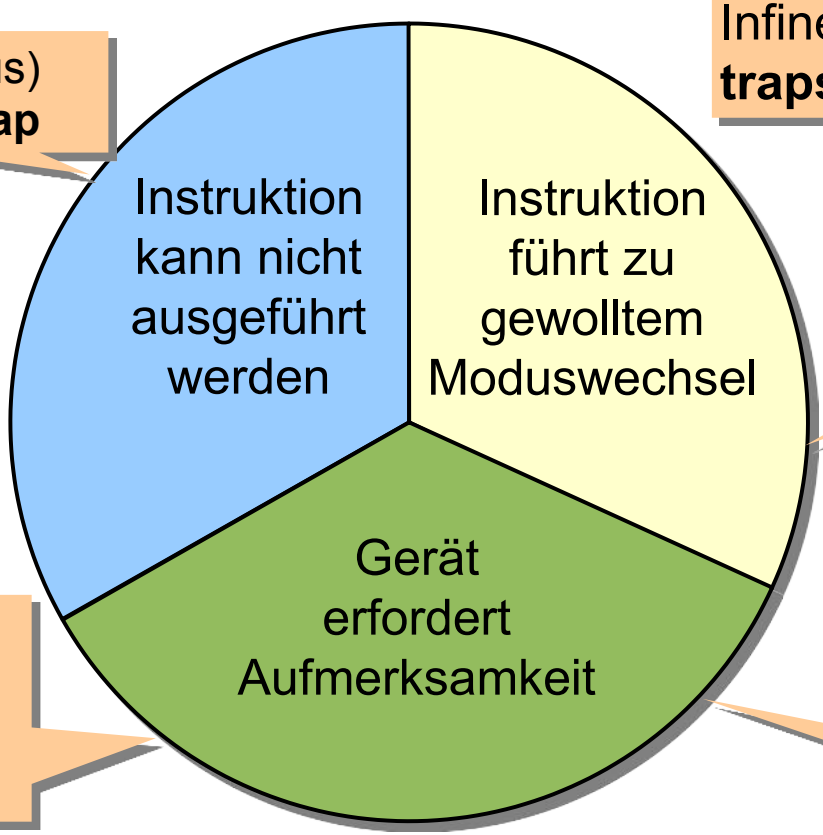
Begriffe: Motorola/Freescale CPU32

- 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



Infineon **TC1-Architektur** (TriCore): **traps** und **interrupts**

(synchronous) hardware **trap**

(synchronous) software **trap**

(asynchronous) hardware **trap**
- „NMI“
- DMA-Fehler

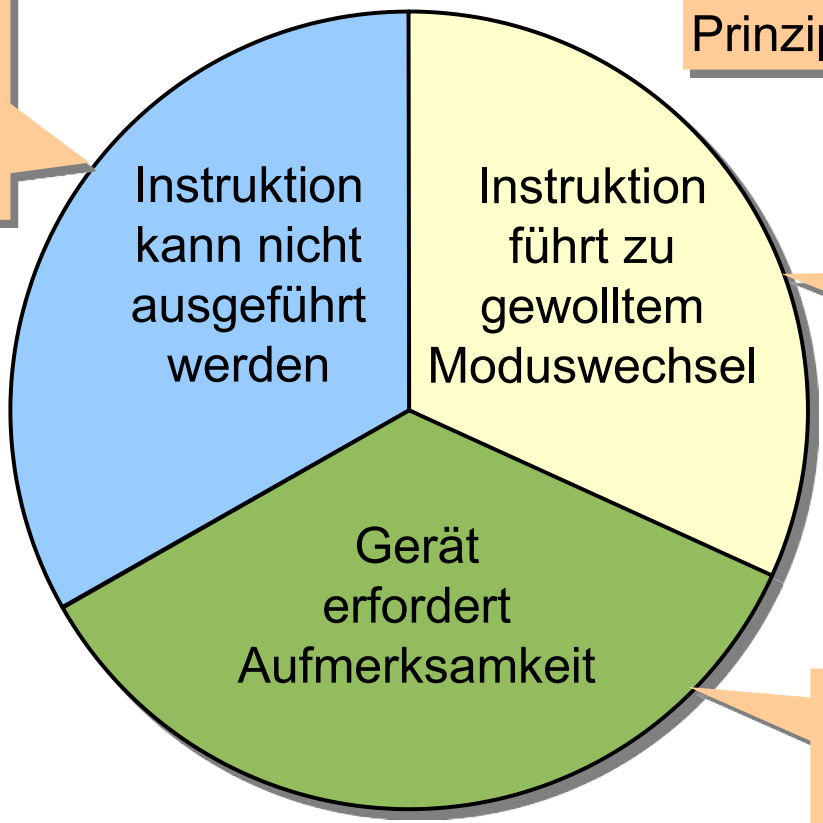
interrupt

Begriffe: Literatur (Stallings)

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

Trap
Behandlung eines Fehlers oder eines Ausnahme-zustandes

William Stallings: „Betriebssysteme: Prinzipien und Umsetzung“



Supervisor-Aufruf
Aufruf einer Betriebssystemfunktion

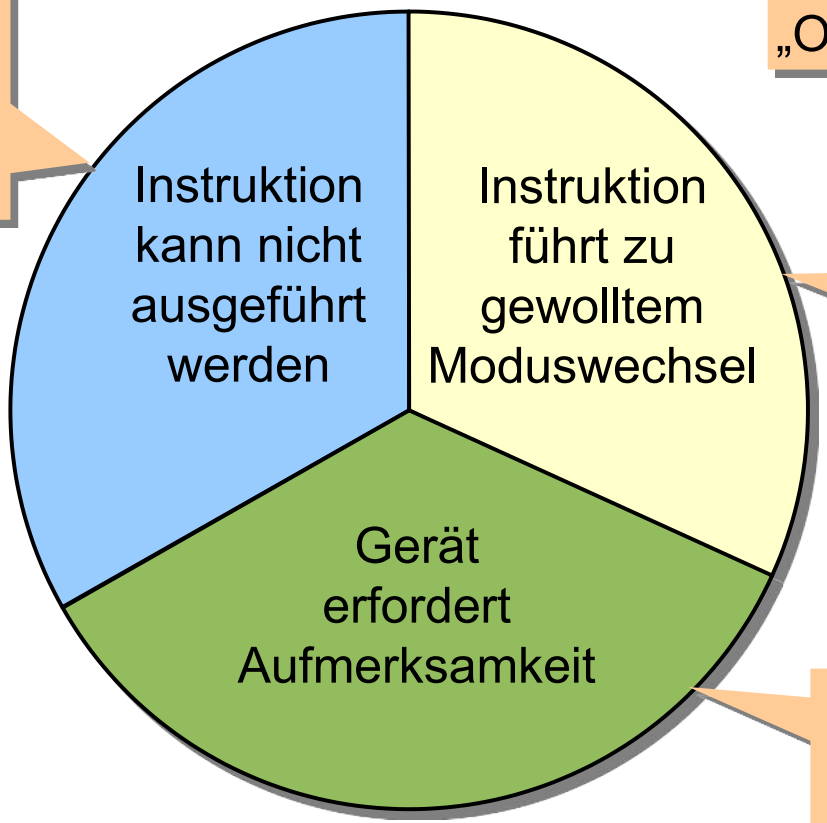
Interrupt
Reaktion auf ein externes asynchrones Ereignis

Begriffe: Literatur (Silberschatz)

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

exception
Behandlung eines Fehlers oder eines Ausnahme-zustandes

Silberschatz *et al* :
„Operating System Concepts“



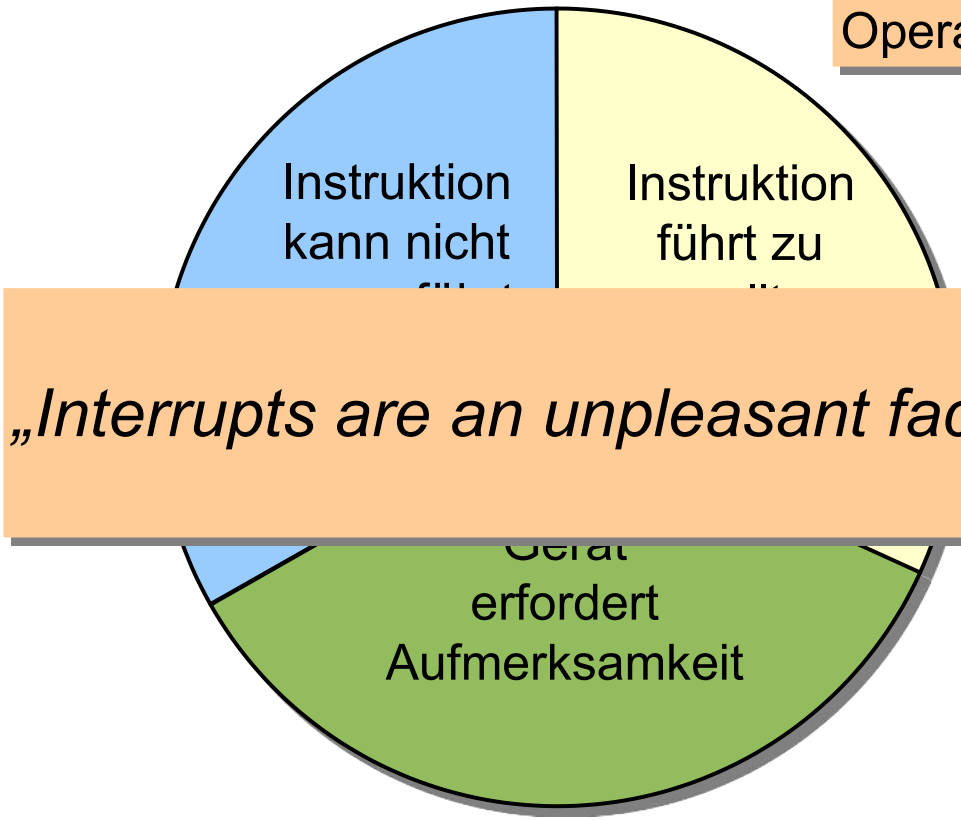
software interrupt / trap
Aufruf einer Betriebssystemfunktion

interrupt
Reaktion auf ein externes asynchrones Ereignis

Begriffe: Literatur (Tanenbaum)

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

Andrew S. Tanenbaum: „Modern Operating Systems“ (ältere Ausgabe)

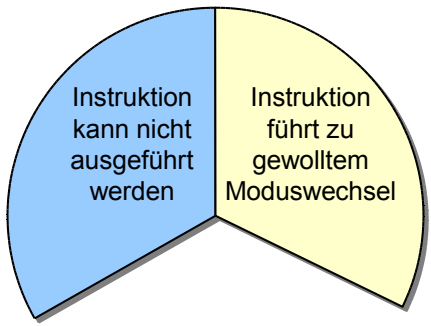


„Interrupts are an unpleasant fact of life“

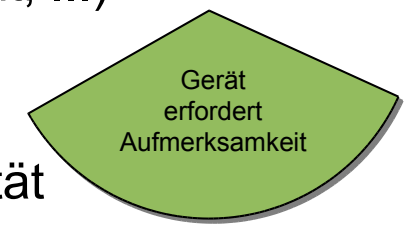


Begriffe: Unser Verständnis in BS

- **„*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

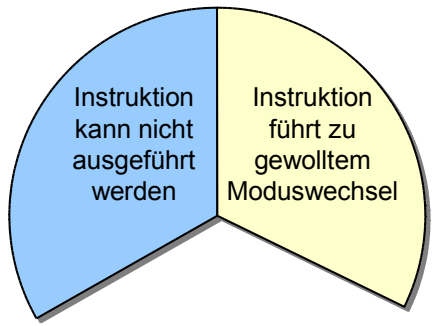


- **„Unterbrechung“** (engl. *Interrupt*):
  - durch **Hardware** ausgelöst
    - Hardware verlangt die Aufmerksamkeit der Software (Zeitgeber, Tastatursteuereinheit, Festplattensteuereinheit, ...)
  - Eigenschaften:
    - nicht vorhersagbar, nicht reproduzierbar
    - in der Regel Wiederaufnahme der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

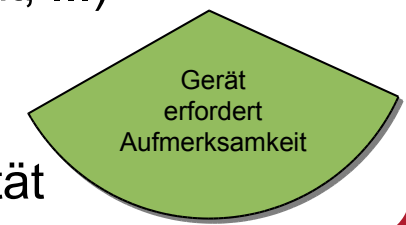


Begriffe: Unser Verständnis in BS

- „*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

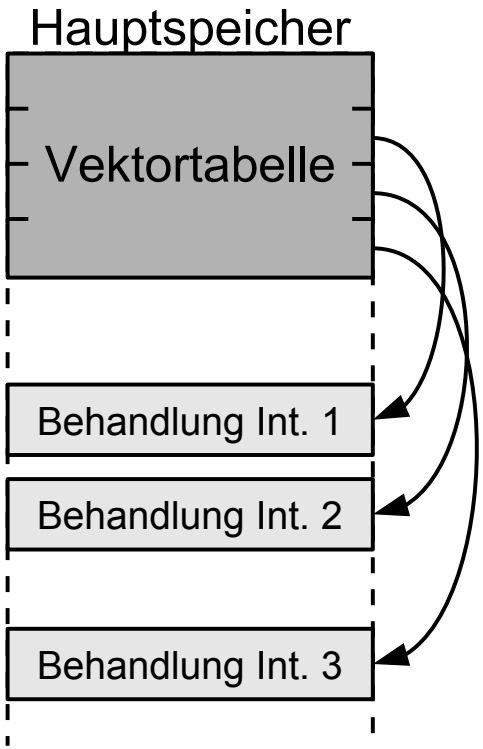
1. Die CPU startet die Behandlungsroutine automatisch.
2. Die Unterbrechnungsbehandlung erfolgt im Systemmodus.
3. Das unterbrochene Programm kann fortgesetzt werden.
4. Die Maschineninstruktionen verhalten sich atomar.
5. Die Unterbrechungsbehandlung kann unterdrückt werden.



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

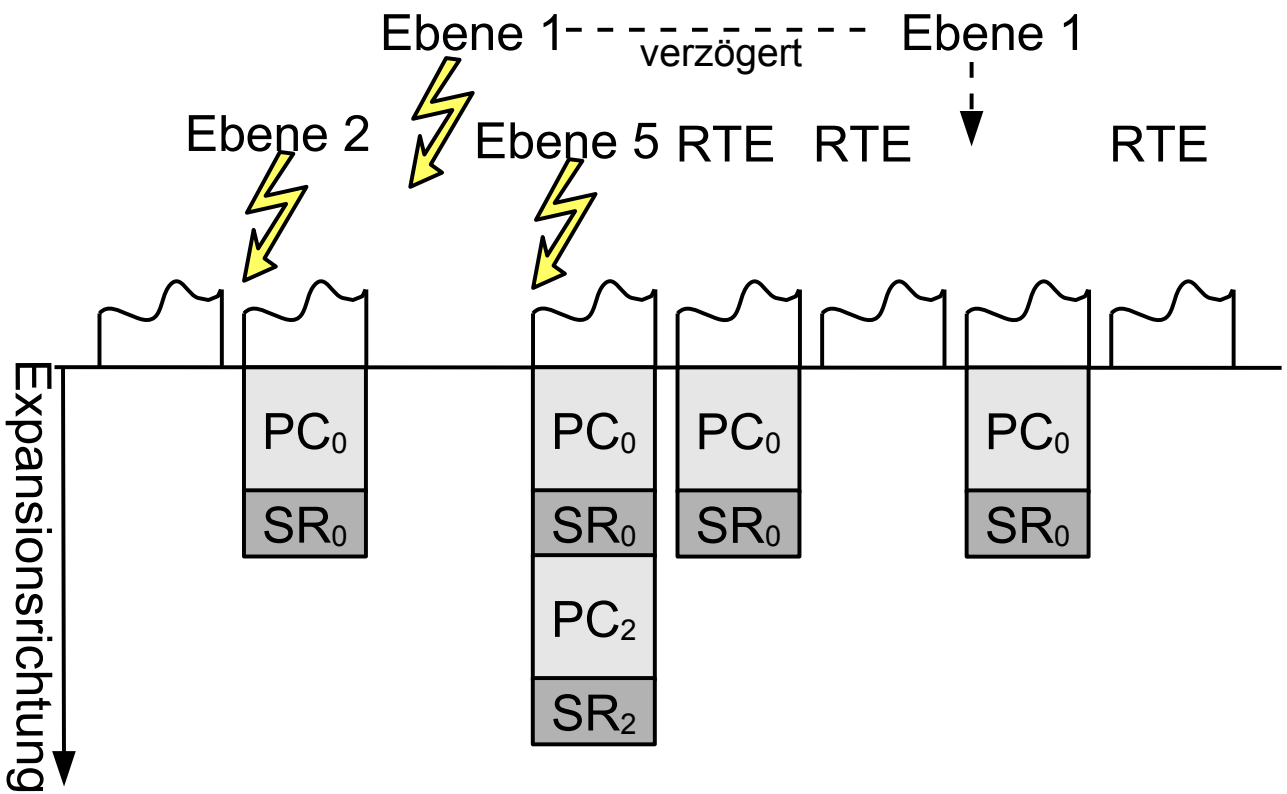
Varianten:
- bei **16-Bit-CPUs** ist eine Aufteilung in Benutzer-/Systemmodus eher die Ausnahme
- bei **8-Bit-CPUs** (oder kleiner) gibt es diese Aufteilung nicht



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

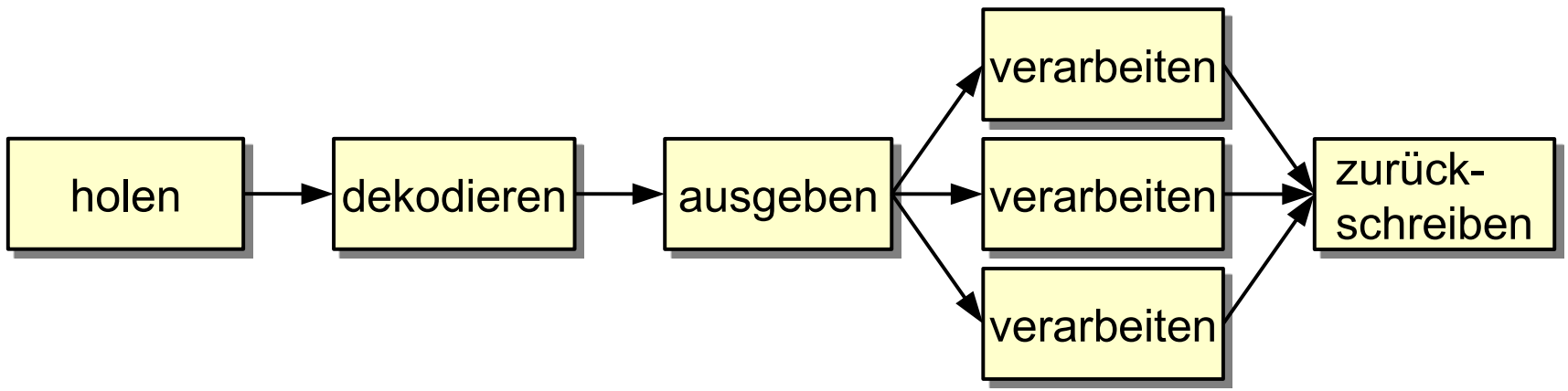
- trivial bei CPUs mit klassischem von-Neumann-Zyklus
- nicht-trivial bei modernen CPUs:
  - Fließbandverarbeitung: Befehle müssen annuliert werden
  - Superskalare CPUs: zusätzlich Befehlsreihenfolge merken



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



Agenda

Einordnung

Begriffe und Grundannahmen

Zustandssicherung

Zustandsänderungen

Zusammenfassung



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



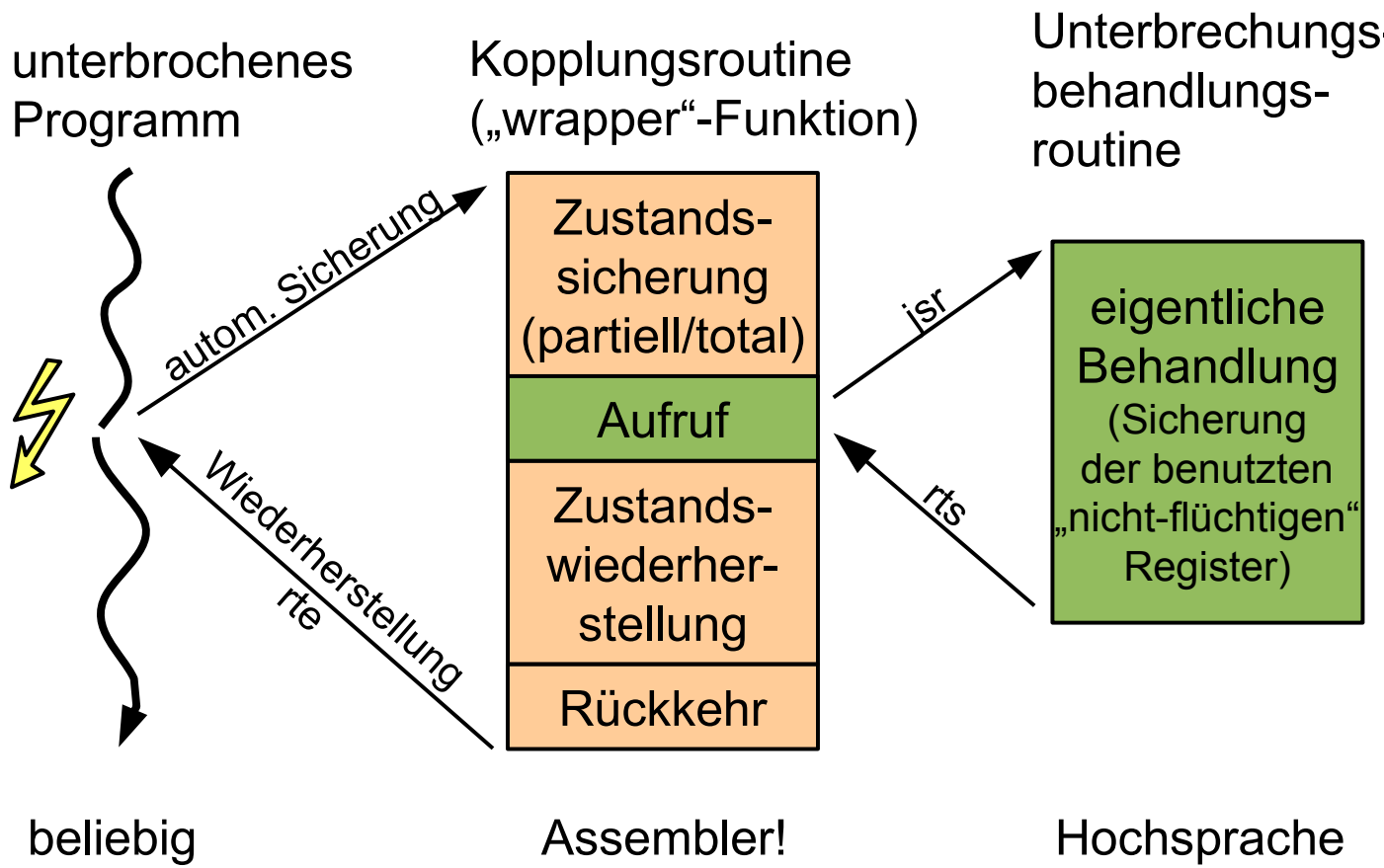
Zustandssicherungskonzepte

- **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“



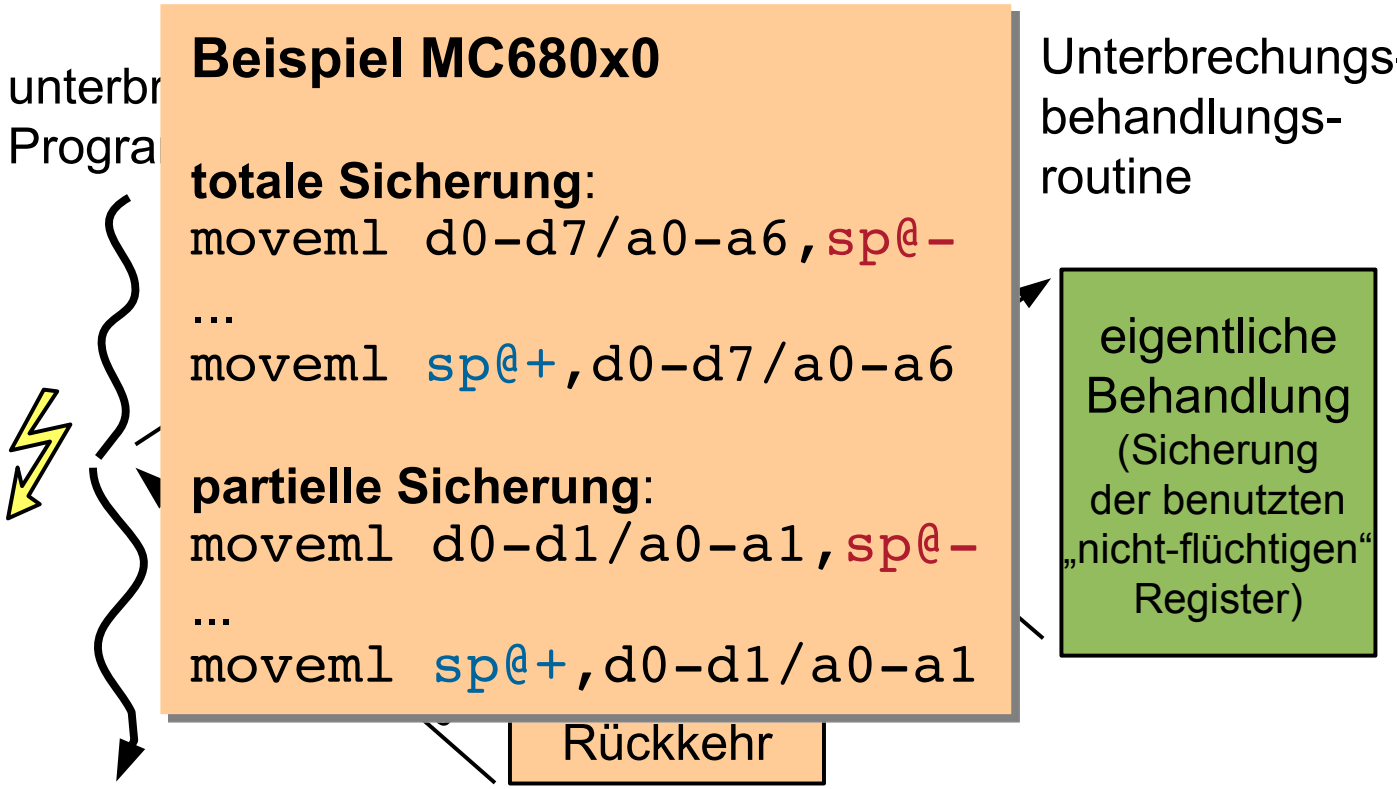
Übergang auf die Hochsprachenebene

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



Übergang auf die Hochsprachenebene

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



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.



Agenda

- Einordnung
- Begriffe und Grundannahmen
- Zustandssicherung
- Zustandsänderungen
  - Beispiele
  - Problemanalyse
  - Lösungsansätze
- Zusammenfassung



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



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++;
}
```

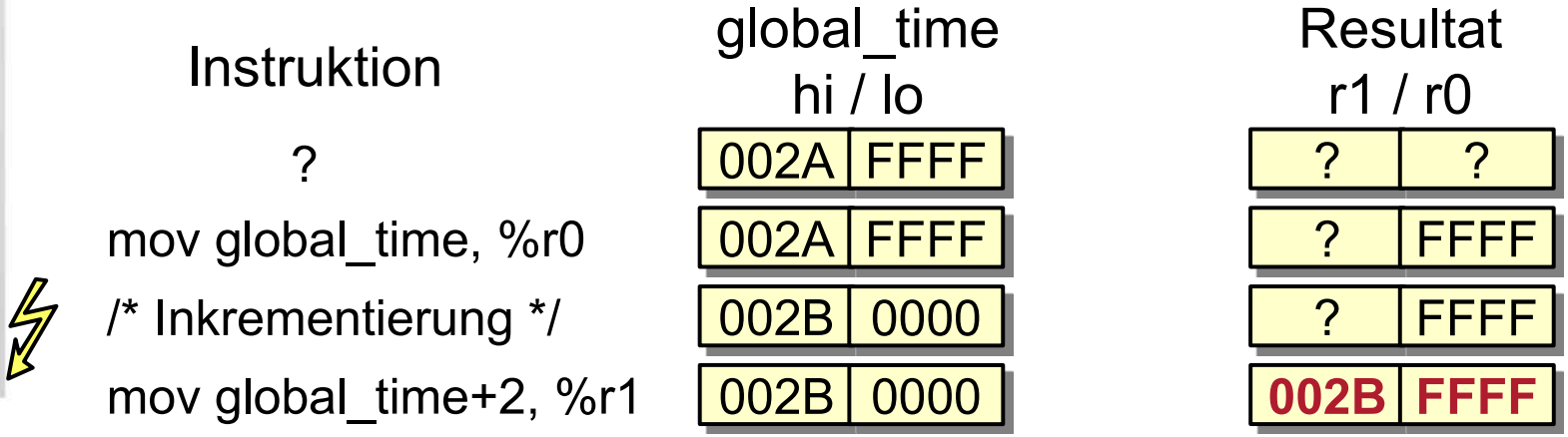
Beispiel 1: Systemzeit

- 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



Beispiel 1: Systemzeit

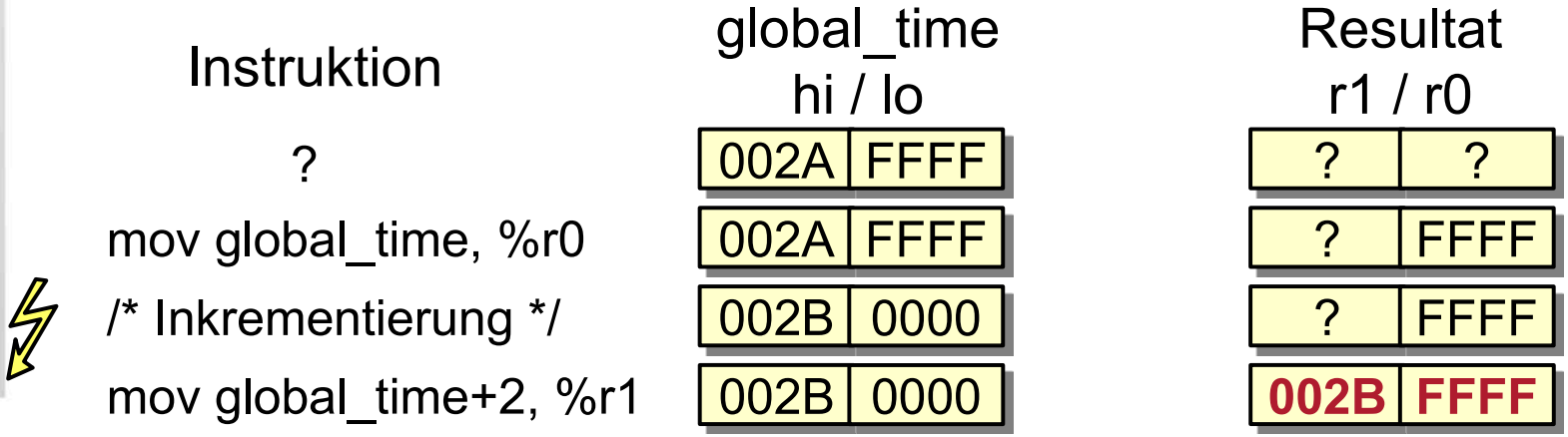
- hier schlummert möglicherweise ein Fehler ...
  - das Lesen von global_time erfolgt nicht notwendigerweise atomar!

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

Problem:

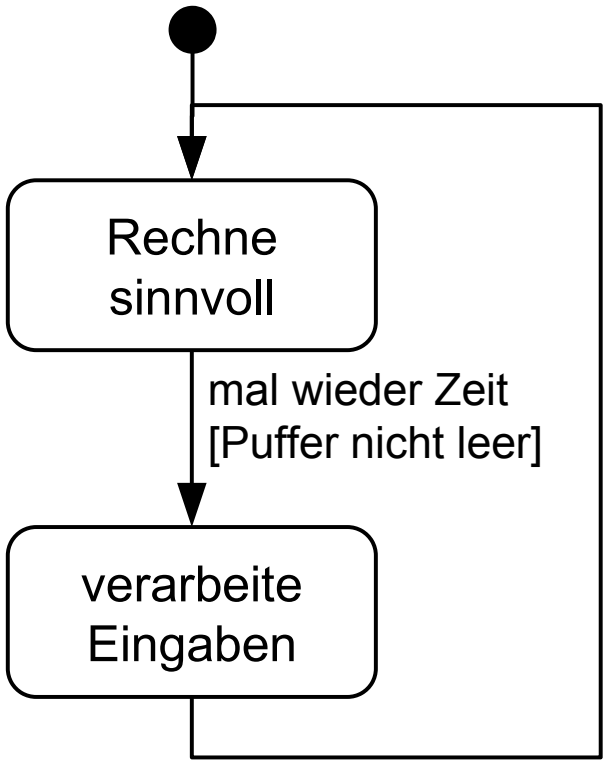
Alle 18,2 Stunden kann die Systemzeit (kurz) um etwa die gleiche Zeit vorgehen. Leider ist das Problem nicht verlässlich reproduzierbar.

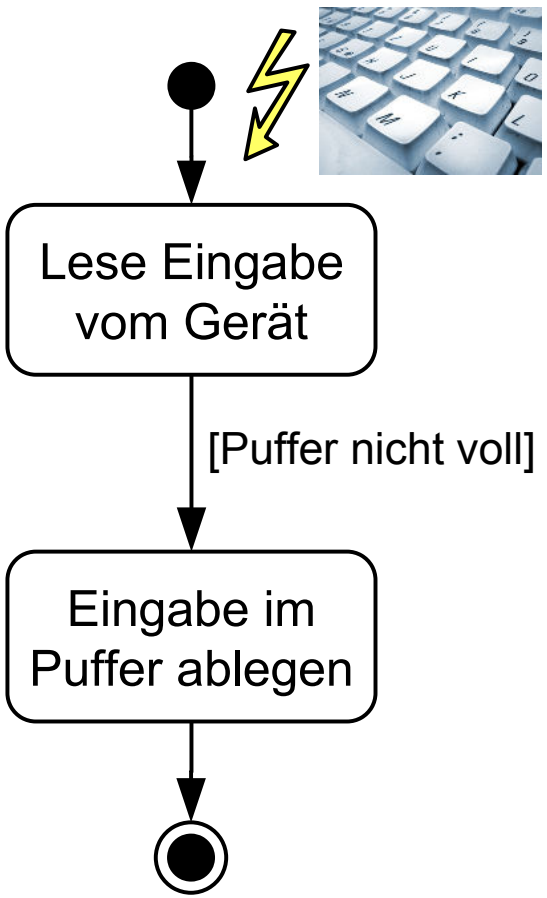
- kritisch ist eine Unterbrechung zwischen den beiden Leseinstruktionen bei der 16 Bit CPU



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



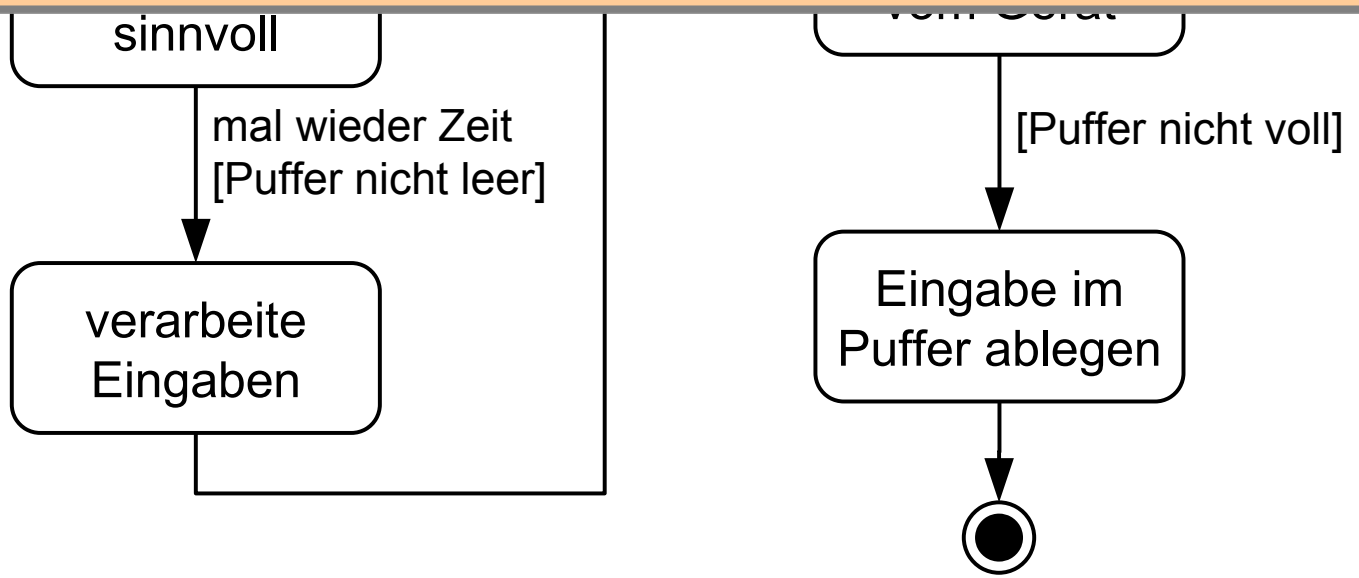


Beispiel 2: Ringpuffer

- 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 ...

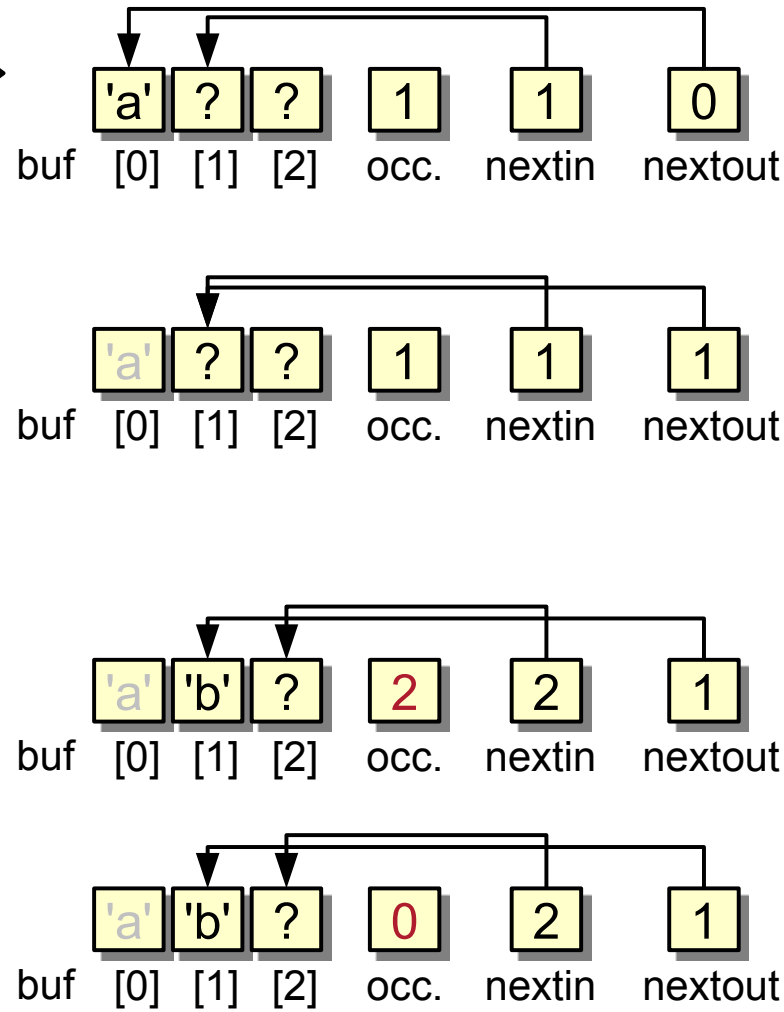
Ausführung

Zustand

```
char consume() {
  int elements = occupied; // 1
  if (elements == 0) return 0;
  char result = buf[nextout]; //'a'
  nextout++; nextout %= SIZE;

void produce(char data) { // 'b'
  int elements = occupied; // 1!
  if (elements == SIZE) return;
  buf[nextin] = data;
  nextin++; nextin %= SIZE;
  occupied = elements + 1; // 2
}

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



Beispiel 2: Ringpuffer

auch die Pufferimplementierung ist kritisch ...

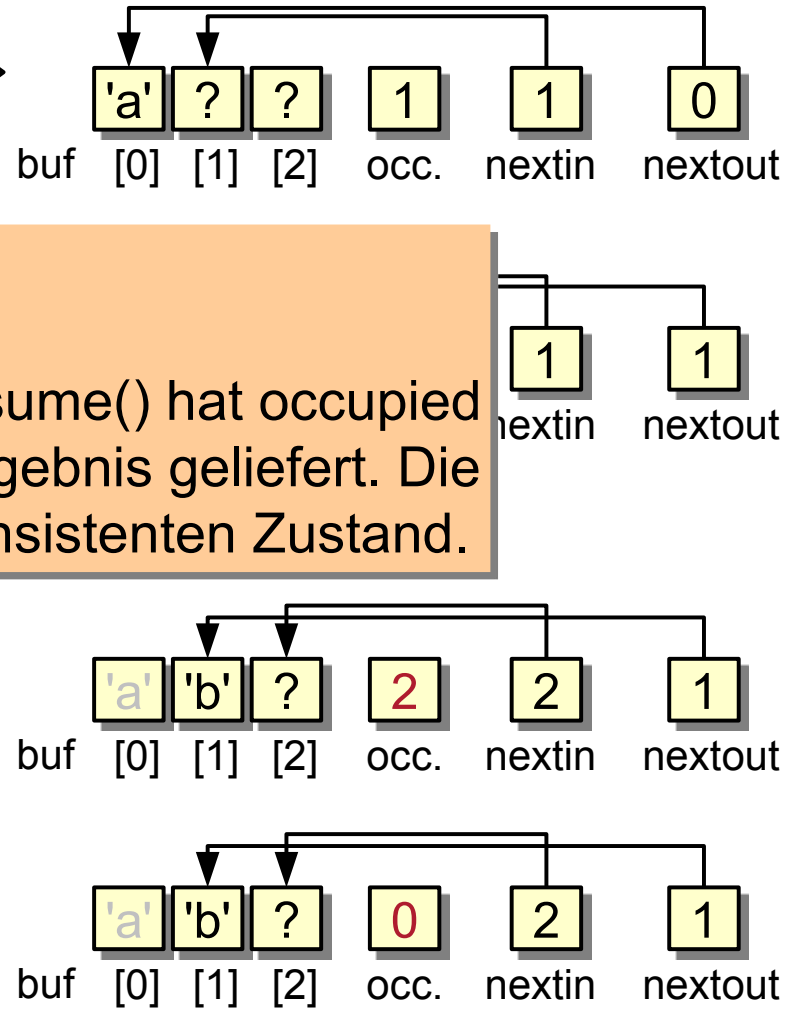
Ausführung

Zustand

```
char consume() {
  int elements = occupied; // 1
  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.

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

- Einordnung
- Begriffe und Grundannahmen
- Zustandssicherung
- Zustandsänderungen
- Zusammenfassung



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!)

