

## Gliederung

### 1 Teilinterpretation

- Hybride Maschine

### 2 Programmunterbrechung

- Trap
- Interrupt

### 3 Laufzeitkontext

- Ausnahmen
- Sicherung/Wiederherstellung

### 4 Nichtsequentialität

- Konkurrenz
- Koordinierung

### 5 Zusammenfassung

## Elementaroperationen der Maschinenprogrammebene

Maschinenprogramme umfassen zwei Sorten von Befehlen [4]:

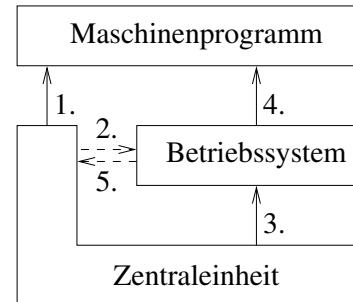
- ① Anweisungen an das Betriebssystem (Ebene 3)
  - explizit als **Systemaufruf** (engl. *system call*) kodiert
  - implizit als **Programmunterbrechung** (engl. *trap, interrupt*) ausgelöst
- ② Anweisungen an die CPU (Ebene 2)

Ausführende Instanz ist immer die CPU, die nur Ebene 2-Befehle kennt

- Ebene 3-Befehle
  - werden „wahrgenommen“, nicht ausgeführt
  - signalisieren eine **Ausnahme** (engl. *exception*)

- Betriebssysteme fangen Ebene 3-Befehle ab, behandeln Ausnahmen

## Ausführung von Maschinenprogrammen

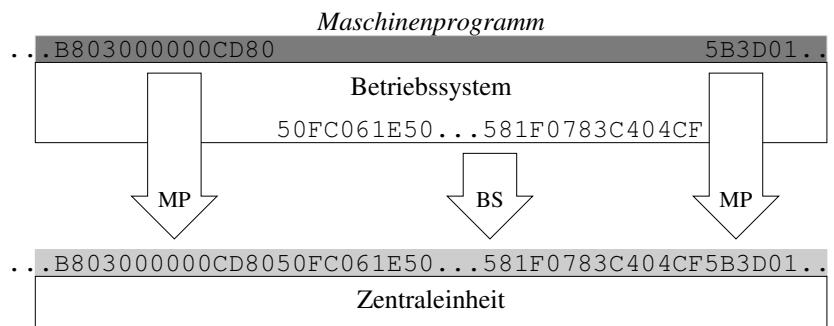


- ① Die Zentraleinheit interpretiert das Maschinenprogramm befehlsweise,
- ② setzt dessen Ausführung aus,
  - Ausnahmesituation
  - **Programmunterbrechung**
 startet das Betriebssystem und
- ③ interpretiert die Programme des Betriebssystems befehlsweise.

### Folge von 3., der Ausführung von Betriebssystemprogrammen...

- ④ Das Betriebssystem interpretiert das soeben oder zu einem früheren Zeitpunkt unterbrochene Maschinenprogramm befehlsweise und
- ⑤ instruiert die Zentraleinheit, die Ausführung des/eines zuvor unterbrochenen Maschinenprogramms wieder aufzunehmen.

## Logischer Aufbau des Befehlsstroms für die Zentraleinheit



Eine bei Ausführung eines Maschinenprogramms vom *realen Prozessor* (Ebene<sub>2</sub>, Zentraleinheit) gestartete Behandlung einer Ausnahmesituation bewirkt — bildlich gesprochen — das „Einschieben“ von Befehlsfolgen jener Programme in den Befehlsstrom, die den abstrakten Prozessor „Betriebssystem“ (Ebene<sub>3</sub>) implementieren.

## Gliederung

### 1 Teilinterpretation

- Hybride Maschine

### 2 Programmunterbrechung

- Trap
- Interrupt

### 3 Laufzeitkontext

- Ausnahmen
- Sicherung/Wiederherstellung

### 4 Nichtsequentialität

- Konkurrenz
- Koordinierung

### 5 Zusammenfassung

## Zwischenzusammenfassung

Befehle der Maschinenprogrammbene, also Ebene<sub>3</sub>-Befehle...

- sind „normale“ Befehle der Ebene<sub>2</sub>, die die CPU ausführt
- sind „ausnahmebedingte“ Befehle, die das Betriebssystem ausführt

... implementieren z.B. Adressräume, Dateien, Prozesse

- Interpreter dieser zusätzlichen Befehle ist das Betriebssystem

## Betriebssysteme sind ausnahmsweise aktiv

Sie werden immer von außerhalb aktiviert...

- im Falle eines Systemaufrufs (**CD80**: Linux/x86), programmiert
  - im Falle von Ausnahmesituationen, nicht programmiert
- ... und deaktivieren sich immer selbst, programmiert (**CF**: x86)

## Unterbrechungsarten und Ausnahmesituationen

Ausnahmesituationen (der Ebene<sub>2</sub>) fallen in zwei Kategorien:

- ① die „Falle“ (engl. *trap*)
- ② die „Unterbrechung“ (engl. *interrupt*)

Unterschiede ergeben sich hinsichtlich...

- Quelle
- Synchronität
- Vorhersagbarkeit
- Reproduzierbarkeit

## Behandlung ist zwingend und grundsätzlich prozessorabhängig

- vom jeweiligen realen *und* abstrakten Prozessor
- genauer: von **Zentraleinheit** und **Betriebssystem**

## Synchrone Programmunterbrechung

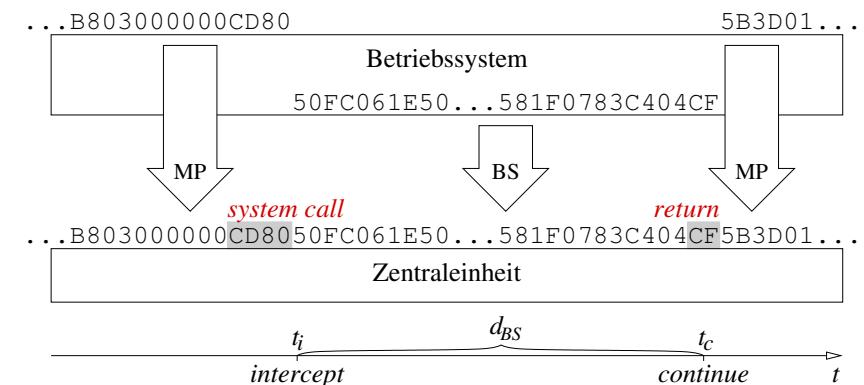
- unbekannter Befehl, falsche Adressierungsart oder Rechenoperation
- Systemaufruf, Adressraumverletzung, unbekanntes Gerät
- Seitenfehler im Falle lokaler Ersetzungsstrategien

### Trap — synchron, vorhersagbar, reproduzierbar

*Ein in die Falle gelaufenes („getrapptes“) Programm, das unverändert wiederholt und jedesmal mit den selben Eingabedaten versorgt auf ein und dem selben Prozessor zur Ausführung gebracht wird, wird auch immer wieder an der selben Stelle in die selbe Falle tappen.*

- durch das Programm in Ausführung (≡ Prozess) selbst ausgelöst
  - bei Ablauf eines Maschinenbefehls des zugrundeliegenden Prozessors
- Trapvermeidung ohne Behebung der Ausnahmebedingung unmöglich

## Synchrone Programmunterbrechung — Trap



### Unterbrechungsverzögerung (engl. trap latency)

$d_{BS}$  Ebene 3; begrenzt bei Echtzeitfähigkeit, unbegrenzt sonst

## Asynchrone Programmunterbrechung

- Signalisierung „externer“ Ereignisse
- Beendigung einer DMA- bzw. E/A-Operation
- Seitenfehler im Falle globaler Ersetzungsstrategien

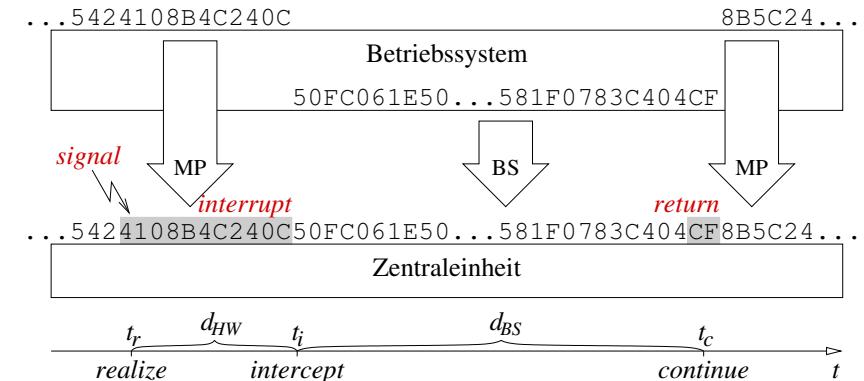
### Interrupt — asynchron, unvorhersagbar, nicht reproduzierbar

*Ein „externer Prozess“ (z.B. ein Gerät) signalisiert einen Interrupt unabhängig vom Arbeitszustand des gegenwärtig sich in Ausführung befindlichen Programms.*

*Ob und ggf. an welcher Stelle das betreffende Programm unterbrochen wird, ist nicht vorhersehbar.*

- durch einen anderen, externen (Soft-/Hardware-) Prozess ausgelöst
- Ausnahmesituationsbehandlung muss **nebeneffektfrei** verlaufen

## Asynchrone Programmunterbrechung — Interrupt



### Unterbrechungsverzögerung (engl. interrupt latency)

$d_{HW}$  Ebene 2; konstant bei RISC, variabel (begrenzt) bei CISC

$d_{BS}$  Ebene 3 ~ Trap (S. 10) ..... ↳ BS kann  $d_{HW}$  beeinträchtigen

## Trap oder Interrupt?

```
#include <stdlib.h>

float frandom () {
    return random()/random();
}
```

### Division (durch Null)

- Programmunterbrechung (je nach Prozessor/CPU)
- wenn, dann wird diese zufällig geschehen

**Programmierfehler**, der sich jedoch nicht zwingend auswirken muss:

- die Unterbrechung verläuft **synchron** zum Programmablauf..... Trap?
- die Unterbrechungsstelle im Programm ist **vorhersagbar** ..... Trap?
- der Zufall macht die Unterbrechung  $\neg$ reproduzierbar ..... Interrupt?
- je nach Anfangswert (*random seed*) und Güte des Zufallsgenerators
- die mögliche Unterbrechung ist „programmiert“, sie wird vom Prozess selbst ausgelöst und bedeutet damit einen **Trap!**

## Gliederung

### 1 Teilinterpretation

- Hybride Maschine

### 2 Programmunterbrechung

- Trap
- Interrupt

### 3 Laufzeitkontext

- Ausnahmen
- Sicherung/Wiederherstellung

### 4 Nichtsequentialität

- Konkurrenz
- Koordinierung

### 5 Zusammenfassung

## Trap oder Interrupt? (Forts.)

```
extern edata, end;
int main () {
    char* p = (char*)&edata;
    do *p++ = 0;
    while (p != (char*)&end);
}
```

### Indirekte Adressierung (\*p++)

- Programmunterbrechung (je nach Systemauslastung)
- trotz korrektem Text

**Seitenfehler** (engl. *page fault*), vorbehaltlich eines virtuellen Speichers

- die Unterbrechung verläuft **synchron** zum Programmablauf..... Trap?

### Diskussionsstoff: Ersetzungsstrategie (engl. replacement policy)

- optionales Merkmal der **Speicherverwaltung** eines Betriebssystems:
  - **lokal**  $\leadsto$  Stelle **vorhersagbar**, Unterbrechung **reproduzierbar** ... Trap?
  - **global**  $\leadsto$  Stelle  $\neg$ **vorhersagbar**, Unterbrechung  $\neg$ **reproduzierbar**. Int.?
- die Antwort bleibt ohne Wissen über Betriebssystemabläufe offen

## Ausnahmen von der normalen Programmausführung

Programmunterbrechungen werden durch **Ereignisse** hervorgerufen, die — aus Anwendungssicht — normalerweise unerwünscht sind:

- Signale von der Peripherie (z.B. E/A, Zeitgeber oder „Wachhund“)
- Wechsel der Schutzdomäne (z.B. Systemaufruf)
- Programmierfehler (z.B. ungültige Adresse)
- unerfüllbare Speicheranforderung (z.B. bei Rekursion)
- Einlagerung auf Anforderung (z.B. beim Seitenfehler)
- Warnsignale von der Hardware (z.B. Energiemangel)

Vorkehrungen zur **Ereignisbehandlung** sind im Betriebssystem unabdingbar, im Anwendungsprogramm dagegen nicht zwingend erforderlich:

- sie sind in beiden Fällen problemspezifisch auszulegen

## Ausnahmebehandlung (engl. exception handling, [2])

Bezug zur Softwaretechnik und Übertragung auf Betriebssystemtechnik

### Wiederaufnahmemodell (engl. resumption model)

- die Behandlung der Ausnahmesituation führt zur **Fortsetzung** der Ausführung des unterbrochenen Programms
- ein Trap kann, ein Interrupt muss so behandelt werden

### Beendigungsmodell (engl. termination model)

- die Behandlung der Ausnahmesituation führt zum **Abbruch** der Ausführung des unterbrochenen Programms
  - ggf. als Folge eines schwerwiegenden Laufzeitfehlers
- ein Trap kann, ein Interrupt darf **niemals** so behandelt werden

- Auslösung (engl. *raising*) einer Ausnahme impliziert **Kontextwechsel**

## Prozessorstatus invariant halten

Ebene 2 (CPU) sichert bei Ausnahmen einen Zustand minimaler Größe

- Statusregister (SR) und Befehlszeiger (engl. *program counter*, PC)
- möglicherweise aber auch den kompletten Registersatz
- je nach CPU werden dabei wenige bis sehr viele Daten(bytes) bewegt

Ebene 3/5 (Betriebssystem/Kompilierer) sichert den restlichen Zustand

- d.h., alle  $\left\{ \begin{array}{l} \text{dann noch ungesicherten} \\ \text{flüchtigen}^1 \\ \text{im weiteren Verlauf verwendeten} \end{array} \right\}$  CPU-Register

- die zu ergreifenden Maßnahmen sind höchst **prozessorabhängig**

<sup>1</sup> Register, deren Inhalte nach Rückkehr von einem Prozeduraufruf verändert worden sein dürfen: festgelegt in den **Prozedurkonventionen** des Kompilierers.

## Abrupter Wechsel des Laufzeitkontextes

Programmunterbrechungen implizieren **nicht-lokale Sprünge**:

vom  $\left\{ \begin{array}{l} \text{unterbrochenen} \\ \text{behandelnden} \end{array} \right\}$  Programm zum  $\left\{ \begin{array}{l} \text{behandelnden} \\ \text{unterbrochenen} \end{array} \right\}$  Programm

Aufrufe von und Rückkehr aus **Unterbrechungsbehandlungs Routinen** geschehen sporadisch und wechseln den Laufzeitkontext:

- erfordert Maßnahmen zur Zustandssicherung/-wiederherstellung
- Mechanismen liefert das behandelnde Programm/die tiefere Ebene

- der **Prozessorstatus** unterbrochener Programme muss invariant sein

## Prozessorstatus sichern und wiederherstellen

... alle dann noch ungesicherten CPU-Register

### Zeile

```
1: train:  
2: pushal  
3: call handler  
4: popal  
5: iret
```

### x86

### m68k

```
train:  
moveml d0-d7/a0-a6,a7@-  
jsr handler  
moveml a7@+,d0-d7/a0-a6  
 rte
```

### train (trap/interrupt):

- Arbeitsregisterinhalte im RAM sichern (2) und wiederherstellen (4)
- Unterbrechungsbehandlung durchführen (3)
- Ausführung des unterbrochenen Programms wieder aufnehmen (5)

## Beteiligte Prozessoren

- CPU (Ebene 2), Betriebssystem (Ebene 3)

## Prozessorstatus sichern und wiederherstellen (Forts.)

... alle flüchtigen CPU-Register

Zeile

x86

```
1: train:
2:     pushl %edx; pushl %ecx; pushl %eax
3:     call handler
4:     popl %eax; popl %ecx; popl %edx
5:     iret
```

m68k

```
train:
    moveml d0-d1/a0-a1,a7@-
    jsr handler
    moveml a7@+,d0-d1/a0-a1
    rte
```

**train** (trap/interrupt):

- Inhalte flüchtiger Arbeitsregister sichern (2) und wiederherstellen (4)
- Unterbrechungsbehandlung durchführen (3)
- Ausführung des unterbrochenen Programms wieder aufnehmen (5)

### Beteiligte Prozessoren

- CPU (Ebene 2), Betriebssystem (Ebene 3), Kompilierer (Ebene 5)

## Unvorhersagbare Laufzeitvarianzen

Unterbrechungen verzögern Programmabläufe

Problem für **determinierte Programme**...

- lassen bei ein und derselben Eingabe verschiedene Abläufe zu
  - alle Abläufe liefern jedoch stets das gleiche Resultat
- ... da asynchrone Unterbrechungen sie **nicht-deterministisch** machen
- nicht zu jedem Zeitpunkt ist bestimmt, wie weitergefahrene wird

### Kritisch für **echtzeitabhängige Programme**

- die Laufzeitumgebung dieser Programme muss echtzeitfähig sein
  - ermöglicht einem sich in Ausführung befindlichem Programm alle Zeit- und Terminvorgaben seiner Umgebung einzuhalten
  - die Echtzeitbedingungen (des Programms) sind weich, fest oder hart
- nicht vergessen: Berücksichtigung aller Last- und Fehlerbedingungen

## Prozessorstatus sichern und wiederherstellen (Forts.)

... alle im weiteren Verlauf verwendeten CPU-Register

gcc

```
void __attribute__ ((interrupt)) train () {
    handler();
}
```

**\_\_attribute\_\_ ((interrupt))**

- Generierung der speziellen Maschinenbefehle durch den **Kompilierer**
  - zur Sicherung/Wiederherstellung der Arbeitsregisterinhalte
  - zur Wiederaufnahme der Programmausführung
- nicht jeder „Prozessor“ (für C/C++) implementiert dieses Attribut

### Beteiligte Prozessoren

- CPU (Ebene 2), Kompilierer (Ebene 5)

## Gliederung

### 1 Teilinterpretation

- Hybride Maschine

### 2 Programmunterbrechung

- Trap
- Interrupt

### 3 Laufzeitkontext

- Ausnahmen
- Sicherung/Wiederherstellung

### 4 Nichtsequentialität

- Konkurrenz
- Koordinierung

### 5 Zusammenfassung

## Parallele und Funktionale Programmierung, 2. Semester

Ergänzung, Verfeinerung bzw. Vertiefung von PFP

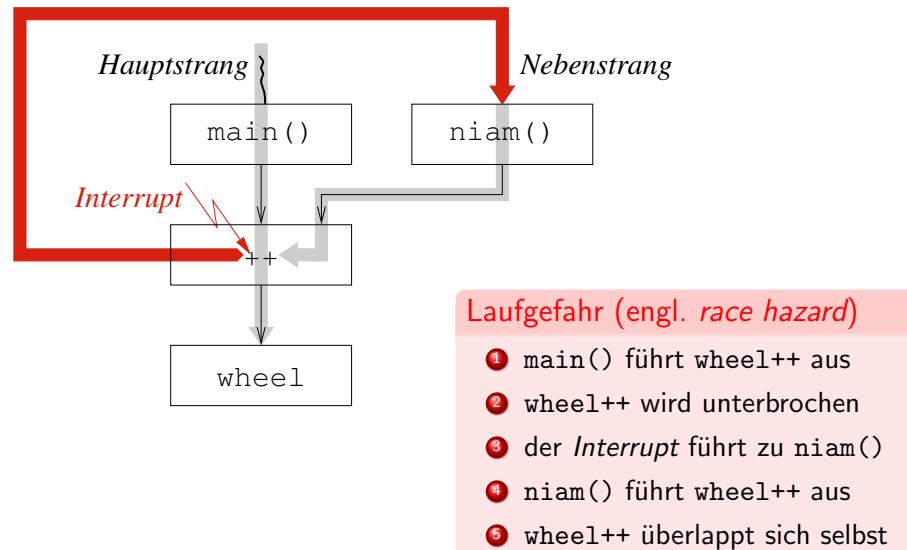


- Teil I, 3. **Datensynchronisation** [3], speziell...
  - ✓ Wettlaufsituationen
  - ✓ gemeinsamer Zustand
  - ✓ kritische Abschnitte

... bei asynchronen Programmunterbrechungen, genauer:

- durch Abstraktion verborgene „Untiefen“ in Maschinenprogrammen
- überlappungsfreie Ausführung kritischer Abschnitte auf Ebene 2
  - verdeutlicht durch Programme der Ebene 4 (Assemblersprache)

## Asynchronität von Programmunterbrechungen



## Nichtdeterministisches Programm

Welche wheel-Werte gibt main() aus?

```
int wheel = 0;

main () {
    for (;;)
        printf("%u\n", wheel++);
}

void __attribute__ ((interrupt))
niam () {
    wheel++;
}
```

normalerweise fortlaufende Werte im Bereich<sup>2</sup>  $[0, 2^{32} - 1]$ , Schrittweite 1

ausnahmsweise dito, allerdings mit Schrittweite  $n$ ,  $0 \leq n \leq 2^{32} - 1$

- je nachdem, wie oft die Ausführung von niam() die von main() im **kritischen Abschnitt** wheel++ überlappt
- $n = 1$  impliziert nicht, dass keine Überlappung stattgefunden hat

<sup>2</sup>Annahme: `sizeof(unsigned int) = 4 Bytes je acht Bits, d.h. 32 Bits.`

## Teilbarkeit von Operationen

wheel++

- eine **Elementaroperation** (kurz: Elop) der Ebene 5
  - der abstrakte C-Prozessor führt wheel++ **atomar**, also **unteilbar** aus
  - nicht zwingend auch eine Elop der Ebene 4 (und tiefer)
    - der reale x86-Prozessor führt wheel++ **bedingt teilbar** aus

	main()	niam()
Ebene 5		wheel++
Ebene 4	movl _wheel,%edx leal 1(%edx),%eax movl %eax,_wheel	incl _wheel
# Elop	3	1

Auszug nach gcc -O6 -S:

in `main()` **teilbar**

in `niam()` **teilbar/unteilbar**

- unteilbar nur bei Uni(kern)prozessoren

## Unterbrechungsfall

- zeitliche Überlappung einer Veränderung des wheel-Wertes
  - niam()-Ausführung überlappt sich mit main()-Ausführung

## Unterbrechungsbedingte Überlappungseffekte

`niam()`-Ausführung überlappt `main()`-Ausführung

<code>_wheel</code>	Befehlsfolge	<code>main()</code>			<code>niam()</code>
		<code>x86-Befehl</code>	<code>%edx</code>	<code>%eax</code>	<code>x86-Befehl</code>
42	1	<code>movl _wheel,%edx</code>	42	?	
43	2				<code>incl _wheel</code>
43	3	<code>leal 1(%edx),%eax</code>	42	43	
43	4	<code>movl %eax,_wheel</code>	42	43	

Kritischer Abschnitt (engl. *critical section*) „`++`“

- zweimal `wheel++` durchlaufen (`main()` und `niam()`)
- zweimal gezählt, den Wert von `wheel` aber nur um eins erhöht

## Schutz vor unterbrechungsbedingten Überlappungen

Lösungsansatz (für den gegebenen Fall, Uni(kern)prozessoren):

- sequentielle Ausführung erzwingen, Überlappungen vorbeugen
  - temporäres Abschalten asynchroner Programmunterbrechungen
  - „Synchronisationsklammern“ um den kritischen Abschnitt setzen
- auf Elop eines tieferen Prozessors (genauer: der CPU) abbilden
  - die „bessere Lösung“, sofern die CPU eine passende Elop dafür anbietet
  - ggf. praktikabel bei CISC, jedoch nicht zwingend dann auch bei RISC

## Abstraktion als grundsätzliche Vorgehensweise

Elop mit den funktionalen Eigenschaften des kritischen Abschnitts bilden

- einen kritischen Abschnitt als **Modul** abkapseln
- den modularisierten Programmtext passend synchronisieren

## Wettlaufsituation (engl. *race condition*)

- zwei (oder mehr) Aktionen wetteifern um die Absicht, als erste ein Berechnungsergebnis herbeizuführen
  - Signal auslösen, Datum verändern, ..., Betriebsmittel beanspruchen
- eine Berechnung zeigt eine unerwartet kritische Abhängigkeit vom relativen Zeitverlauf von Ereignissen
  - fehlerhafte Stelle in einem (Hardware/Software) System
- ein potentielles Problem, sobald die Ausführung von Programmen nebenläufig (d.h. überlappend oder parallel) möglich ist

Programmabschnitte, die Wettlaufsituationen enthalten, bilden sogenannte **kritische Abschnitte** und sind speziell zu behandeln:

- (a) den Programmabschnitt **physisch schützen**, dass er nur sequentiell von gleichzeitigen Ausführungssträngen durchlaufen werden kann *oder*
- (b) den Programmabschnitt als **logische Einheit** auslegen, dass er trotz nichtsequentieller Ausführung stets ein konsistentes Ergebnis liefert

## Funktionale Abstraktion vom kritischen Abschnitt

Randbedingung: Anweisung `wheel++` erhöht den Wert von `wheel` um 1 *nachdem* dieser als **Ausdruckswert** bestimmt wurde (engl. *post-increment*)

- den Wert erst holen, dann inkrementieren (engl. *fetch and increment*)
- entsprechend funktioniert die benötigte Elop: `int fai (int *)`

```
int wheel = 0;
```

```
main () {
    for (;;) {
        printf("%u\n", fai(&wheel));
    }
}
```

```
void __attribute__ ((interrupt))
niam () {
    fai(&wheel);
}
```

## Optionale Merkmale der (System-) Software

- niam()**
  - muss ggf. Überlappung durch sich selbst ins Kalkül ziehen
    - bei Verschachtelungen von Programmunterbrechungen
    - bei `niam()`-Ausführung in der Unterbrechungsbehandlung
- main()**
  - muss ggf. Überlappung durch `niam()` ins Kalkül ziehen

## Auslegungsvarianten der kritischen Funktion

### Abilden auf Komplexbefehl inc

```
int fai (int *ref) {
    int aux = *ref;

    asm volatile ("incl %0"
        : "=m" (*ref)
        : "m" (*ref));

    return aux;
}
```

### Abilden auf Komplexbefehl add

```
int fai (int *ref) {
    int aux = *ref;

    asm volatile ("addl $1,%0"
        : "=m" (*ref)
        : "m" (*ref));

    return aux;
}
```

**Semantik:** `return aux` liefert nicht zwingend `(*ref) - 1`

- Bestimmung des Ausdruckswerts und Veränderung der Variablen erfolgt durch Ausführung von (wenigstens) zwei Maschinenbefehlen
- dazwischen ist eine Unterbrechung möglich, die eine Überlappung der Ausführung der Anweisungsfolgen nach sich ziehen kann

## Zweckmäßigkeit der vorgestellten Implementierungen

- |  |   |
|--|---|
| <code>inc</code> bzw. <code>add</code> | <ul style="list-style-type: none"> <li>brauchbar, wenn die abweichende Semantik tolerierbar ist</li> <li>eine Entscheidung allein der Anwendungsebene</li> </ul>  |
| <code>cli/sti</code>                   | <ul style="list-style-type: none"> <li>nur gültig für Programme der Befehlssatzebene, beispielsweise Betriebssysteme</li> <li>brauchbar, falls die <b>Unterbrechungsverzögerung</b> (S. 12) dadurch nicht zu stark beeinträchtigt wird</li> <li>eine Entscheidung maßgeblich der Anwendungsebene</li> </ul> |
| <code>xadd</code>                      | <ul style="list-style-type: none"> <li>uneingeschränkt brauchbar</li> </ul>   |

**Beachte:** Elop `fai()` allein garantiert noch keine korrekten Ausgaben

- die Operation garantiert nur konsistentes Zählen im Überlappungsfall
- Schrittweiten ungleich 1 bei der Ausgabe sind weiterhin möglich
  - bedingt durch Verdrängungslatenz bzw. Unterbrechungsfrequenz
- eine Lösung im Anwendungsprogramm selbst ist ggf. noch erforderlich
- Koordinierung paralleler Abläufe ist ein **querschneidender Belang...**

## Auslegungsvarianten der kritischen Funktion (Forts.)

Stärkere Semantik: `fai()` liefert den zum „`++`-Zeitpunkt gültigen Ausgangswert

### Unterbrechungen sperren

```
int fai (int *ref) {
    int aux;

    asm volatile ("cli");
    aux = (*ref)++;
    asm volatile ("sti");

    return aux;
}
```

### Spezialbefehl `xadd` nutzen

```
int fai (int *ref) {
    int aux = 1;

    asm volatile ("xaddl %0,%1"
        : "=g" (aux), "=m" (*ref)
        : "0" (aux), "m" (*ref));

    return aux;
}
```

**Beachte:** Ausführungsrechte und Arbeitsmodus einer CPU<sup>a</sup>

<sup>a</sup>Privilegierte Befehle sind nur im privilegierten Arbeitsmodus ausführbar.

- |                   |  |   |
|-------------------|--|---|
| <code>cli</code>  | <ul style="list-style-type: none"> <li>privilegierter Maschinenbefehl, bedingt ausführbar</li> </ul>         | ☺ |
| <code>sti</code>  | <ul style="list-style-type: none"> <li>privilegierter Maschinenbefehl, bedingt ausführbar</li> </ul>         | ☺ |
| <code>xadd</code> | <ul style="list-style-type: none"> <li>nicht-privilegierter Maschinenbefehl, unbedingt ausführbar</li> </ul> | ☺ |

## Gliederung

- 1 Teilinterpretation
  - Hybride Maschine
- 2 Programmunterbrechung
  - Trap
  - Interrupt
- 3 Laufzeitkontext
  - Ausnahmen
  - Sicherung/Wiederherstellung
- 4 Nichtsequentialität
  - Konkurrenz
  - Koordinierung
- 5 Zusammenfassung

## Resümee

- Maschinenprogramme werden durch **Teilinterpretation** ausgeführt
  - normalerweise laufen sie auf der Befehlssatzebene (CPU) ab
  - ausnahmeweise greift die Betriebssystemebene in die Ausführung ein
- **Programmunterbrechungen** leiten die Teilinterpretation ein
  - synchron* zur Programmausführung durch einen *Trap*
  - asynchron* zur Programmausführung durch einen *Interrupt*
- die Behandlung von Unterbrechungen ist eine **Ausnahmesituation**
  - bedingt Sicherung und Wiederherstellung des Laufzeitkontextes
  - hat allgemein unvorhersagbare Laufzeitvarianzen zur Folge
- asynchrone Programmunterbrechungen bringen **Nichtsequentialität**
  - ggf. vorhandene kritische Abschnitte sollten modularisiert werden
  - das entstandene Modul der Wettlaufsituation entsprechend auslegen

## Literaturverzeichnis

- [1] FREE SOFTWARE FOUNDATION, INC. (Hrsg.):  
*Using Inline Assembly with gcc.*  
Boston, MA, USA: Free Software Foundation, Inc., Jan. 2000
- [2] GOODENOUGH, J. B.:  
Exception Handling: Issues and a Proposed Notation.  
In: *Communications of the ACM* 18 (1975), Nr. 12, S. 683–696
- [3] PHILIPPSEN, M. :  
*Parallele und Funktionale Programmierung.*  
<http://www2.cs.fau.de/Lehre/SS20??/PFP/material/>, jährlich. –  
Vorlesungsfolien
- [4] TANENBAUM, A. S.:  
*Structured Computer Organization.*  
Prentice-Hall, Inc., 1979. –  
443 S. –  
ISBN 0-130-95990-1
- [5] Intel Pentium Instruction Set Reference.  
<http://faydoc.tripod.com/cpu>, 2010