Systemprogrammierung

Rechnerorganisation: Betriebssystemebene

Wolfgang Schröder-Preikschat

Lehrstuhl Informatik 4

17. November 2010

Gliederung

Elementaroperationen der Maschinenprogrammebene

Maschinenprogramme umfassen zwei Sorten von Befehlen [4]:

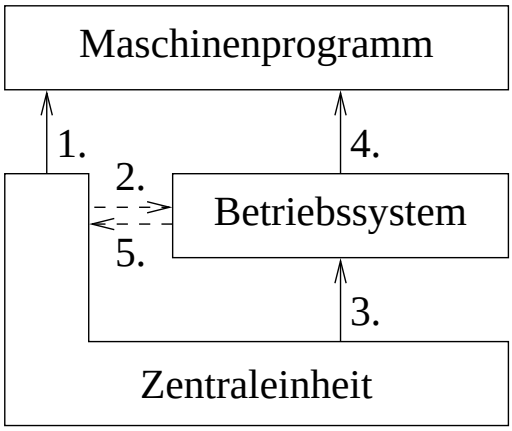
1. Anweisungen an das Betriebssystem (Ebene $_3$)
   - explizit als Systemaufruf (engl. *system call*) kodiert
   - implizit als Programmunterbrechung (engl. *trap*, *interrupt*) ausgelöst
2. Anweisungen an die CPU (Ebene $_2$)

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



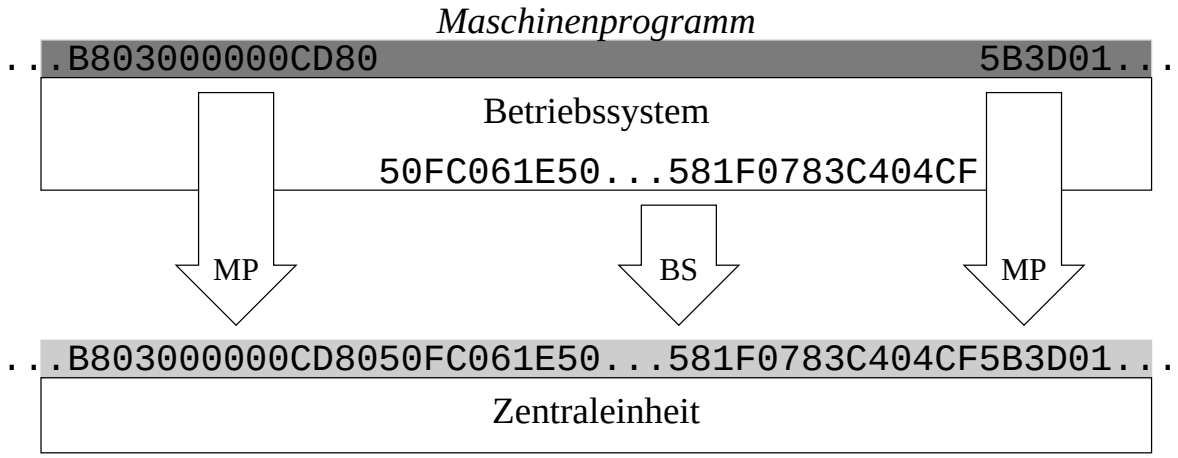
1. Die Zentraleinheit interpretiert das Maschinenprogramm befehlsweise,
2. setzt dessen Ausführung aus,
   - Ausnahmesituation
   - **Programmunterbrechung**

   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 2, Zentraleinheit) gestartete Behandlung einer Ausnahmesituation bewirkt — bildlich gesprochen — das „Einschieben“ von Befehlsfolgen jener Programme in den Befehlsstrom, die den abstrakten Prozessor „Betriebssystem“ (Ebene 3) implementieren.

Zwischenzusammenfassung

Befehle der Maschinenprogrammbene, also Ebene 3-Befehle...

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

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

- Interpret 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)

Gliederung

Unterbrechungsarten und Ausnahmesituationen

Ausnahmesituationen (der Ebene $_2$) fallen in zwei Kategorien:

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

Unterschiede ergeben sich hinsichtlich...

- Quelle
- Synchronität
- Vorhersagbarkeit
- Reproduzierbarbeit

Behandlung ist zwingend und grundsätzlich prozessorabhängig

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

Synchrone Programmunterbrechung

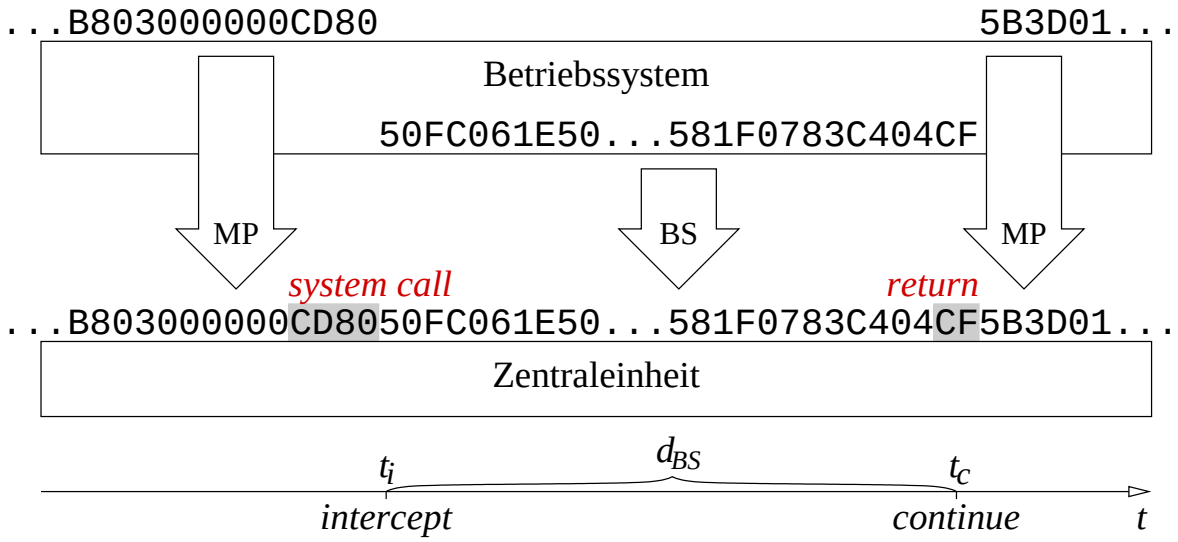
- unbekannter Befehl, falsche Adressierungsart oder Rechenoperation
- Systemaufruf, Adressraumverletzung, unbekanntes Gerät
- Seitenfehler im Falle lokaler Ersetzungsstrategien

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.

- 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

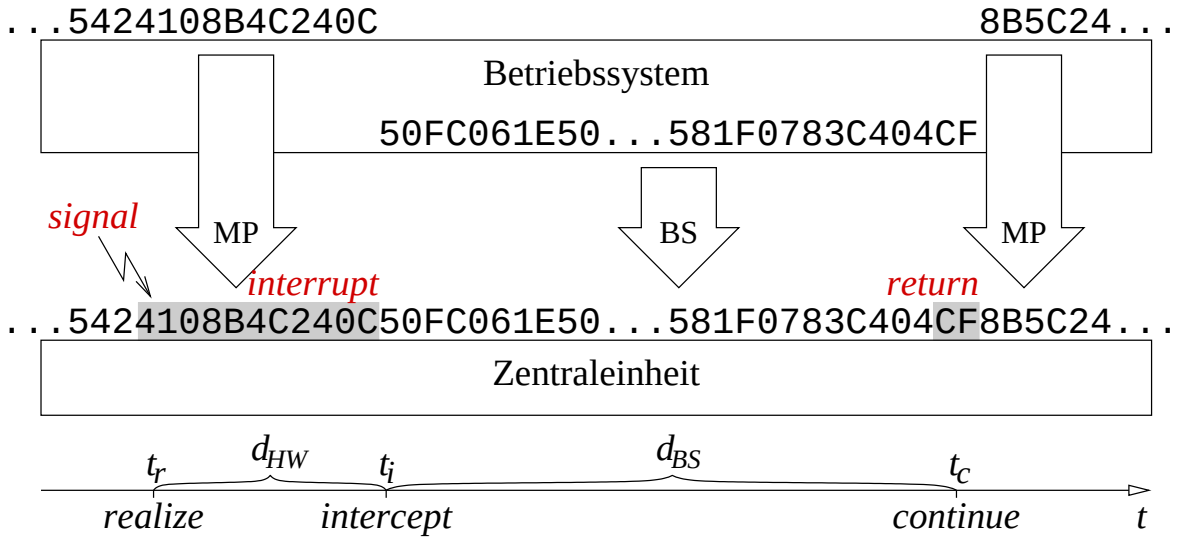
Interrupt — asynchron, unvorhersagbar, nicht reproduzierbar

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.

- 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 $\sim$ *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 CPU)
- wird 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 ¬ reproduzierbar . . . . . Interrupt
  - je nach Anfangswert (*random seed*) und Güte des Zufallsgenerators
- die Unterbrechung ist „programmiert“, sie bedeutet einen **Trap**

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 ⇝ Stelle vorhersagbar, Unterbrechung reproduzierbar . . . . Trap
  global ⇝ Stelle ¬ vorhersagbar, Unterbrechung ¬ reproduzierbar . . Int.
- die Antwort bleibt ohne Wissen über Betriebssystemabläufe offen

Gliederung

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 unabdingbar im Betriebssystem aber abdingbar im Anwendungsprogramm:

- sie sind in beiden Fällen problemspezifisch auszulegen

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

Bezug zur Softwaretechnik und Übertragung auf Betriebssytemtechnik

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**

Abrupter Wechsel des Laufzeitkontextes

Programmunterbrechungen implizieren **nicht-lokale Sprünge**:

vom { unterbrochenen / behandelnden } Programm zum { behandelnden / unterbrochenen } Programm

Aufrufe von und Rückkehr aus **Unterbrechungsbehandlungsroutinen** 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 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 $\left\{\begin{array}{c}\text{dann noch ungesicherten}\\ \text{flüchtigen}[1]\\ \text{im weiteren Verlauf verwendeten}\end{array}\right\}$ CPU-Register

- die zu ergreifenden Maßnahmen sind höchst prozessorabhängig

[1] Register, deren Inhalte nach Rückkehr von einem Prozeduraufruf verändert worden sein dürfen: festgelegt in den **Prozedurkonventionen** des Kompilierers.

Prozessorstatus sichern und wiederherstellen

. . . alle dann noch ungesicherten CPU-Register

Zeile	x86	m68k
1:	`train:`	`train:`
2:	`  pushal`	`  moveml d0-d7/a0-a6,a7@-`
3:	`  call handler`	`  jsr handler`
4:	`  popal`	`  moveml a7@+,d0-d7/a0-a6`
5:	`  iret`	`  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	m68k
1:	`train:`	`train:`
2:	`  pushl %edx; pushl %ecx; pushl %eax`	`  moveml d0-d1/a0-a1,a7@-`
3:	`  call  handler`	`  jsr handler`
4:	`  popl  %eax; popl  %ecx; popl  %edx`	`  moveml a7@+,d0-d1/a0-a1`
5:	`  iret`	`  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)

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)

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 weitergefahren 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

Gliederung

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 (Assembliersprache)

Nichtdeterministisches Programm

Welche `wheel`-Werte gibt `main()` aus?

```
int wheel = 0;
```

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

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

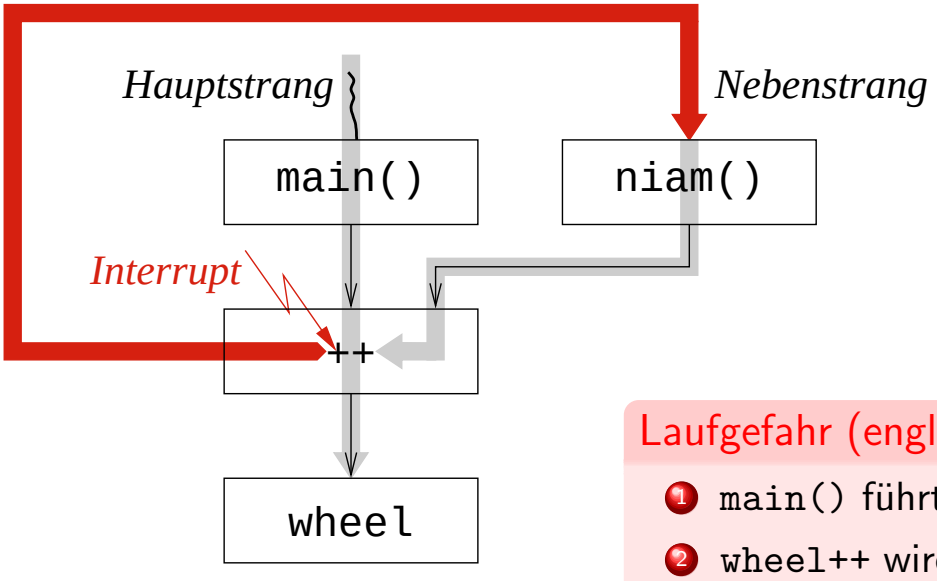
normalerweise fortlaufende Werte im Bereich[2] $[0, 2^{32} - 1]$, Schrittweite 1

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

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

[2] Annahme: `sizeof(unsigned int)` = 4 Bytes je acht Bits, d.h. 32 Bits.

Asynchronität von Programmunterbrechungen



Laufgefahr (engl. *race hazard*)

1. `main()` führt `wheel++` aus
2. `wheel++` wird unterbrochen
3. der *Interrupt* führt zu `niam()`
4. `niam()` führt `wheel++` aus
5. `wheel++` überlappt sich selbst

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

`_wheel`	Befehls-folge	`main()`			`niam()`
		x86-Befehl	%edx	%eax	*x86-Befehl*
42	1	`movl _wheel,%edx`	42	?	
43	2				`incl _wheel`
43	3	`leal 1(%edx),%eax`	42	43	
43	4	`movl %eax,_wheel`	42	43	

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

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

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

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

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 inkremtieren (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

Abbilden auf Komplexbefehl `inc`

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

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

    return aux;
}
```

Abbilden auf Komplexbefehl `add`

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

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

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), "=g" (*ref)
        :  "0" (aux),  "1" (*ref));

    return aux;
}
```

Beachte: Ausführungsrechte und Arbeitsmodus einer CPU[a]

[a] Privilegierte Befehle erfordern die Ausführung im privilegierten Arbeitsmodus.

`cli`	• privilegierter Maschinenbefehl, bedingt ausführbar	☹
`sti`	• privilegierter Maschinenbefehl, bedingt ausführbar	☹
`xadd`	• nicht-privilegierter Maschinenbefehl, unbedingt ausführbar	☺

Zweckmäßigkeit der vorgestellten Implementierungen

`inc` bzw. `add`	• brauchbar, wenn die abweichende Semantik tolierbar ist • eine Entscheidung allein der Anwendungsebene
`cli`/`sti`	• nur gültig für Programme der Befehlssatzebene, beispielsweise Betriebssysteme • brauchbar, falls die Unterbrechungsverzögerung (S. 12) dadurch nicht zu stark beeinträchtig wird • eine Entscheidung maßgeblich der Anwendungsebene
`xadd`	• uneingeschränkt brauchbar

Beache: 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**. . .

Gliederung

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/SS2008/PFP/material/, 2008. –
Vorlesungsfolien

[4] Tanenbaum, A. S.:
Structured Computer Organization.
Prentice-Hall, Inc., 1999. –
ISBN 0–130–95990–1

[5] *Intel Pentium Instruction Set Reference.*
http://faydoc.tripod.com/cpu, 2010