

Systemprogrammierung

Grundlagen von Betriebssystemen

Teil B – V.3 Rechnerorganisation: Betriebssystemmaschine

4. Juli 2024

Rüdiger Kapitza

(© Wolfgang Schröder-Preikschat, Rüdiger Kapitza)



Lehrstuhl für Informatik 4
Systemsoftware



Friedrich-Alexander-Universität
Technische Fakultät

Gliederung

Einführung

Hybride Maschine

Teilinterpretation

Ausnahmen

Trap

Interrupt

Programmunterbrechung

Ausnahmen

Sicherung/Wiederherstellung

Nichtsequentialität

Wettlaufsituation

Kritischer Abschnitt

Zusammenfassung

SP

Einführung

B – V3 / 3

Agenda

Einführung

Hybride Maschine

Teilinterpretation

Ausnahmen

Trap

Interrupt

Programmunterbrechung

Ausnahmen

Sicherung/Wiederherstellung

Nichtsequentialität

Wettlaufsituation

Kritischer Abschnitt

Zusammenfassung

SP

Einführung

B – V3 / 2

Lehrstoff

- den Prozessor der Maschinenprogrammebene als **hybride Maschine** kennenlernen und sein **Operationsprinzip** begreifen
 - Ablauf der Teilinterpretation von Maschinenprogrammen verinnerlichen
 - den Aspekt der Ablaufinvarianz eines Betriebssystems nachvollziehen
- Gemeinsamkeiten und Unterschiede von synchronen und asynchronen **Unterbrechungen** verstehen
 - Konzepte „Trap“ und „Interrupt“ differenzieren, voneinander abgrenzen
 - beide als Ausnahme von der normalen Programmausführung sehen
 - den Prozessorstatus bzw. -zustand eines Programmablaufs identifizieren
 - daraus Implikationen für die Unterbrechungsbehandlung ableiten
- ein Betriebssystem als **nichtsequentielles Programm** erkennen und in die „Untiefe“ solcher Programme einführen
 - durch Wettlaufsituationen verursachte Laufgefahren erläutern
 - eine erste Einführung zum zentralen Begriff „kritischer Abschnitt“ geben

virtuelle Maschine ↔ **Betriebssystem** ↔ hybride Maschine

SP

Einführung

B – V3 / 4

Einführung

Hybride Maschine

Elementaroperationen der Maschinenprogrammebene

- Maschinenprogramme umfassen zwei Sorten von Befehlen [2, S. 5]:
 - i Anweisungen an das Betriebssystem, das Ebene₃ implementiert
 - explizit als **Systemaufruf** (*system call*) kodiert
 - implizit als **Unterbrechung** (*trap, interrupt*) ausgelöst
 - ii Anweisungen an die CPU, die Ebene_[2,3] implementiert
 - Ebene₂ direkt, nur dort ist die Ausführung aller Befehle der CPU gültig
 - Ebene₃ indirekt, in enger Kooperation mit dem Betriebssystem
- wirklich ausführende Instanz im Rechensystem ist immer die CPU
 - reine Ebene₃-Befehle $\left\{ \begin{array}{l} \text{werden „wahrgenommen“, nicht ausgeführt,} \\ \text{signalisieren jew. eine Ausnahme (exception),} \\ \text{die ans Betriebssystem „hochgereicht“ wird} \\ \text{um dort behandelt zur werden.} \end{array} \right.$
- Betriebssysteme fangen Ebene₃-Befehle ab, behandeln Ausnahmen
 - sie bilden jeweils eine (logisch) eigenständige **Maschine**
 - die die von ihr ausführenden Befehle von der CPU zugestellt bekommt

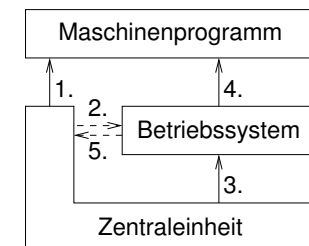
SP

Einführung

B-V3 / 5

Partielle Interpretation

- die CPU (Zentraleinheit):
 1. interpretiert das Maschinenprogramm eingeschränkt befehlsweise,
 2. setzt dessen Ausführung aus,
 - Ausnahmesituation
 - **Unterbrechung**startet das Betriebssystem und
 3. interpretiert die Programme des Betriebssystems befehlsweise.
- Folge von 3., der Ausführung von Betriebssystemprogrammen:
 4. das Betriebssystem interpretiert das in seiner Ausführung unterbrochene Maschinenprogramm befehlsweise und
 5. instruiert die CPU (Zentraleinheit), die Ausführung des/eines zuvor unterbrochenen Maschinenprogramms wieder aufzunehmen.



Einführung

Teilinterpretation

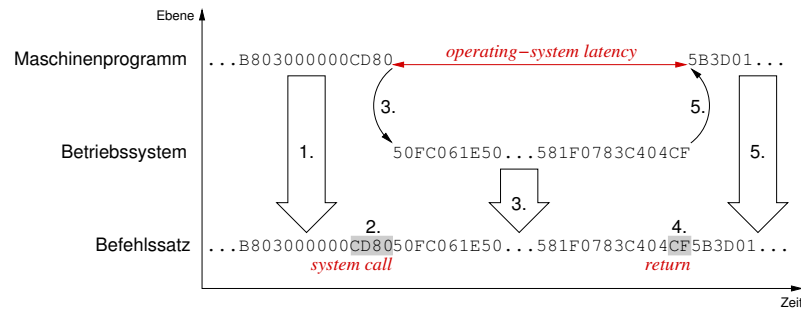
In Phase 1. ist nur eine Teilmenge von Ebene₂-Befehlen direkt von der CPU ausführbar, in Phase 3. dagegen alle.

SP

Einführung

B-V3 / 6

- logischer Aufbau des Befehlsstroms für die Zentraleinheit, in Analogie zu den umseitig (S. 8) genannten fünf Phasen:



- Ausführung eines Maschinenprogramms durch die Zentraleinheit (CPU)
- Wahrnehmung einer synchronen/asynchronen Ausnahme (hier: synchron)
- Teilinterpretation durch das Betriebssystem, Ausnahmebehandlung
- Beendigung der Ausnahmebehandlung/Teilinterpretation
- Wiederaufnahme der Ausführung des Maschinenprogramms

Die in Phase 3. (S. 8) erfolgende Ausführung von Programmen des Betriebssystems kann ebenfalls ausgesetzt werden. Jede Programmausführung kann eine Unterbrechung erfahren, wenn der ausführende Prozessor dazu befähigt ist.

- ablaufinvariant** (*re-entrant*) ausgelegte Betriebssysteme ermöglichen den **Wiedereintritt** während der eigenen Ausführung
- auch wenn Betriebssysteme normalerweise keine Systemaufrufe absetzen, können sie sehr wohl von Unterbrechungen betroffen sein:
 - im Kontext der Ausführung eines Systemaufrufs ☹
 - hervorgerufen durch Peripheriegeräte (Ein-/Ausgabe, Zeitgeber) ☹
 - bedingt durch einen Programm(ier)fehler ~ Panik ☹
- die in der Folge notwendige Unterbrechungsbehandlung gestaltet sich wie eine **indirekte Rekursion**
 - das Betriebssystem wird in seiner Definition selbst nochmals aufgerufen
 - nämlich indirekt durch die CPU im Rahmen der partiellen Interpretation
- der Wiedereintritt kann **asynchron** erfolgen, was das Betriebssystem als **nichtsequentielles Programm** darstellt

Zwischenzusammenfassung

- Befehle der Maschinenprogrammebene, also Ebene₃-Befehle sind...
 - „normale“ Befehle der Ebene₂, die die CPU direkt ausführen kann
 - unprivilegierte Befehle**, die in jedem Arbeitsmodus ausführbar sind
 - „unnormale“ Befehle der Ebene₂, die das Betriebssystem ausführt
 - privilegierte Befehle**, die nur im privilegierten Arbeitsmodus ausführbar sind
- die „aus der Reihe fallenden“ Befehle stellen Adressräume, Prozesse, Speicher, Dateien und Wege zur Ein-/Ausgabe bereit
 - Interpreter dieser Befehle ist das Betriebssystem
 - der dadurch definierte Prozessor ist die **Betriebssystemmaschine**
- demzufolge ist ein Betriebssystem immer nur **ausnahmsweise** aktiv
 - es muss von außerhalb aktiviert werden
 - programmiert im Falle eines Systemaufrufs (CD80: Linux/x86) oder einer sonstigen synchronen Programmunterbrechung (*trap*)
 - nicht programmiert, also nicht vorhergesehen, im Falle einer asynchronen Programmunterbrechung (*interrupt*)
 - es deaktiviert sich immer selbst, in beiden Fällen programmiert (CF: x86)

Gliederung

- Einführung
- Hybride Maschine
- Teilinterpretation
- Ausnahmen
- Trap
- Interrupt
- Programmunterbrechung
- Ausnahmen
- Sicherung/Wiederherstellung
- Nichtsequentialität
- Wettlaufsituation
- Kritischer Abschnitt
- Zusammenfassung

Unterbrechungsarten und Ausnahmesituationen

- die Ausnahmesituationen der Ebene₂ fallen in zwei Kategorien:
 - trap** ■ **Abfangung** für Ausnahmen von interner Ursache
 - interrupt** ■ **Unterbrechung** durch Ausnahmen von externer Ursache
- **Unterschiede** ergeben sich hinsichtlich...
 - Quelle, Synchronität, Vorhersagbarkeit und Reproduzierbarkeit
- ihre **Behandlung ist zwingend** und grundsätzlich prozessorabhängig
 - aufwerfen (*raising*) einer Ausnahme kommt entweder einem realen (CPU) oder einem abstrakten (Betriebssystem) Prozessor zu
 - die CPU wirft eine Ausnahme der Hardware (IRQ, NMI, Fehler)
 - das Betriebssystem wirft eine Ausnahme der Software (POSIX: SIG*)
 - wogegen die Behandlung (*handling*) einem abstrakten Prozessor obliegt
 - Hardwareausnahmen behandelt das Betriebssystem (auf Ebene₂)
 - Betriebssystemausnahmen behandelt das Maschinenprogramm (auf Ebene₃)

SP

Ausnahmen

B - V.3 / 11

Synchrone Ausnahme

- unbekannter Befehl, falsche Adressierungsart oder Rechenoperation
- Systemaufruf, Adressraumverletzung, unbekanntes Gerät
- Seitenfehler im Falle lokaler Ersetzungsstrategien (vgl. [3, S. 16])

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 tapen.

- durch das Programm in Ausführung (\equiv Prozess) selbst ausgelöst
- als Folge der Interpretation eines Befehls des ausführenden Prozessors
- im **Fehlerfall** ist die Behebung der Ausnahmebedingung zwingend

SP

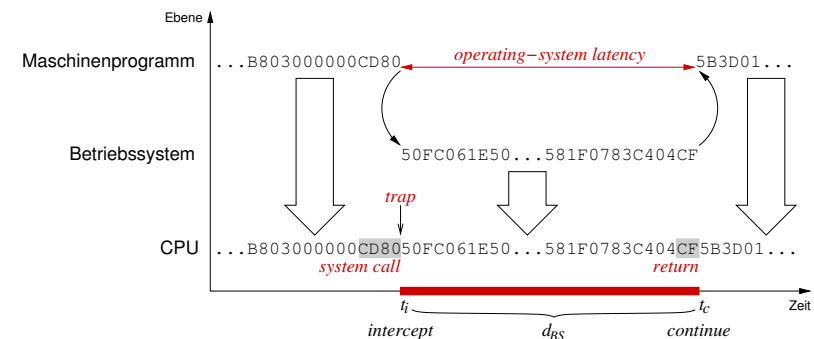
Ausnahmen

B - V.3 / 12

Ausnahmen

Trap

Synchrone Ausnahme – Trap



Betriebssystemlatenz

Verzögerung zwischen dem Abfang-/Unterbrechungszeitpunkt und dem Zeitpunkt der Wiederaufnahme der Programmausführung.

- d_{BS} ■ muss begrenzt sein für ein echtzeitfähiges Betriebssystem
- **maximale Ausführungszeit** (*worst-case execution time, WCET*)

SP

Ausnahmen

B - V.3 / 13

Ausnahmen

Interrupt

Asynchrone Ausnahme

- Signalisierung „externer“ Ereignisse
- Beendigung einer DMA- bzw. E/A-Operation
- Seitenfehler im Falle globaler Ersetzungsstrategien (vgl. [3, S. 16])

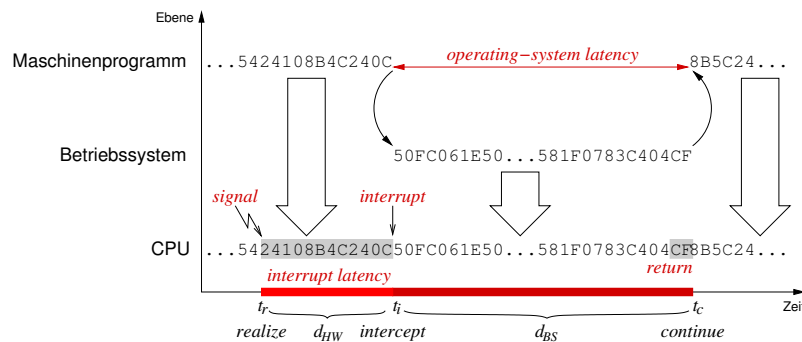
Interrupt — asynchron, unvorhersagbar, nicht reproduzierbar

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 die Ausführung des betreffenden Programms unterbrochen wird, ist nicht vorhersehbar.

- durch einen anderen, externen (Soft-/Hardware-) Prozess ausgelöst
 - unabhängig von der Befehlsinterpretation des ausführenden Prozessors
- **Nebeneffektfreiheit** der Unterbrechungsbehandlung ist zwingend

Asynchrone Ausnahme — Interrupt



Unterbrechungslatenz

Verzögerung zwischen Wahrnehmung (durch die CPU) und Annahme (im Betriebssystem) der Unterbrechung.

- d_{HW} Restausführungszeit des laufenden Befehls der CPU plus
- restliche Dauer einer **Unterbrechungssperre** im Betriebssystem

Gliederung

- Einführung
 - Hybride Maschine
 - Teilinterpretation
- Ausnahmen
 - Trap
 - Interrupt
- Programmunterbrechung
 - Ausnahmen
 - Sicherung/Wiederherstellung
- Nichtsequentialität
 - Wettlaufsituation
 - Kritischer Abschnitt
- Zusammenfassung

Programmunterbrechung

Ausnahmen

Abrupter Wechsel des Laufzeitkontextes

vgl. [2, S. 30]

- Unterbrechungen implizieren **nicht-lokale Sprünge**:

vom { unterbrochenen
behandelnden } Prozess zum { behandelnden
unterbrochenen } Prozess

- der **Unterbrechungshandhaber**¹ wird plötzlich aktiviert, sein exakter Aktivierungszeitpunkt ist nicht vorhersehbar
- der **Prozessorstatus** des unterbrochenen Programms ist daher während der Unterbrechungsbehandlung **invariant** zu halten
 - Sicherung vor Ansprung bzw. Start der Behandlungsroutine
 - Wiederherstellung vor Rücksprung zum unterbrochenen Programm
- Mechanismen dazu liefert die Befehlssatzebene (CPU) bzw. das jeweils behandelnde Programm (Betriebssystem) selbst
- führt zu **Betriebslast**, deren Höhe die Programmierenebene des Betriebssystems und die Befehlssatzebene bestimmt

¹Dem deutschen Patentwesen entnommen, das die englische Bezeichnung „handler“ fachbegrifflich als „Handhaber“ übersetzt. Dort wird „trap“ sachlich und fachlich korrekt auch als „Falle“ verstanden (vgl. auch S. 16).

Ausnahmen von der normalen Programmausführung

Ausführungsunterbrechungen sind **Ereignisse**, die den unterbrochenen Programmablauf unerwünscht verzögern und nicht immer durch ihn selbst auch verursacht 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)

Im Betriebssystem sind Maßnahmen zur **Ereignisbehandlung** unabdingbar, im Maschinenprogramm dagegen nicht.

- sie sind in beiden Fällen jedoch immer problemspezifisch auszulegen

Programmunterbrechung

Sicherung/Wiederherstellung

Prozessorstatus invariant halten

- die CPU führt eine **totale oder partielle Zustandssicherung** durch
 - minimal**
 - Statusregister (SR) und Befehlszähler (*program counter*, PC)
 - maximal**
 - den kompletten Registersatz
- Maßnahme, die im **Unterbrechungszyklus** der CPU stattfindet
 - je nach CPU werden dabei wenige bis sehr viele Daten(bytes) bewegt
- das Betriebssystem führt eine **partielle Zustandssicherung** durch

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

^aRegister, deren Inhalte nach Rückkehr von einem Prozeduraufruf verändert worden sein dürfen: festgelegt in der **Aufrufkonvention** des Compilers.

- die erste Option betrifft ein Betriebssystem, das mit Sprachkonzepten der Ebene₄ (d.h., in Assemblersprache) programmiert wurde
- demgegenüber betreffen die letzten beiden Optionen ein in Hochsprache (Ebene₅) programmiertes Betriebssystem

SP

Programmunterbrechung

B - V.3 / 19

Option 1: Alle dann noch ungesicherten...

- Mantelprozedur** zur Statussicherung auf Ebene₂:
 - Behandlungsroutine (*handler*) evtl. in Assemblersprache programmiert

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

```
m68k
1 train:
2   moveml d0-d7/a0-a6,a7@-
3   jsr handler
4   moveml a7@+,d0-d7/a0-a6
5   rte
```

- train (trap/interrupt):**
 - alle Arbeitsregisterinhalte im RAM (Stapelspeicher) sichern
 - Unterbrechungsbehandlung durchführen
 - im RAM gesicherten Arbeitsregisterinhalte wiederherstellen
 - unterbrochene Programmausführung wieder aufnehmen
- beteiligte Prozessoren:
 - CPU (Ebene₂), Betriebssystem (Ebene₃)

SP

Programmunterbrechung

B - V.3 / 20

Option 2: Alle flüchtigen...

vgl. [2, S.30]

- Mantelprozedur** zur Statussicherung in Bezug auf Ebene₅:
 - Behandlungsroutine (*handler*) in Hochsprache programmiert

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

```
m68k
1 train:
2   moveml d0-d1/a0-a1,a7@-
3   jsr handler
4   moveml a7@+,d0-d1/a0-a1
5   rte
```

- train (trap/interrupt):**
 - Inhalte flüchtiger Arbeitsregister im RAM (Stapelspeicher) sichern
 - Unterbrechungsbehandlung durchführen
 - im RAM gesicherten Arbeitsregisterinhalte wiederherstellen
 - unterbrochene Programmausführung wieder aufnehmen
- beteiligte Prozessoren:
 - CPU (Ebene₂), Betriebssystem (Ebene₃), Compiler (Ebene₅)

SP

Programmunterbrechung

B - V.3 / 21

Option 3: Alle im weiteren Verlauf verwendeten...

- Mantelprozedur** zur Statussicherung auf Ebene₅:
 - Behandlungsroutine (*handler*) in Hochsprache programmiert

```
1 inline void __attribute__((interrupt)) train () {
2   handler();
3 }
```

- __attribute__((interrupt))**:
 - Generierung der speziellen Maschinenbefehle durch den **Compiler**
 - zur Sicherung/Wiederherstellung der Arbeitsregisterinhalte
 - zur Wiederaufnahme der Programmausführung
 - nicht jeder „Prozessor“ (für C/C++) implementiert dieses Attribut
- beteiligte Prozessoren:
 - CPU (Ebene₂), Compiler (Ebene₅)

SP

Programmunterbrechung

B - V.3 / 22

Gliederung

Einführung

Hybride Maschine

Teilinterpretation

Ausnahmen

Trap

Interrupt

Programmunterbrechung

Ausnahmen

Sicherung/Wiederherstellung

Nichtsequentialität

Wettlaufsituation

Kritischer Abschnitt

Zusammenfassung

SP

Nichtsequentialität

B - V.3 / 23

Nichtsequentialität

Wettlaufsituation

Nichtsequentielles Programm

vgl. [8, S. 9]

```
1 int wheel = 0;
```

- welche wheel-Werte gibt main() aus?

```
2 main () {  
3     for (;;)   
4         printf("%u\n", wheel++);  
5 }
```

- normalerweise fortlaufende Werte im Bereich² $[0, 2^{32} - 1]$, Schrittweite 1

- falls niam() main() unterbricht: welche Ausgabewerte nun?

```
6 void __attribute__((interrupt)) niam () {  
7     wheel++;  
8 }
```

- mit Schrittweite n , $0 \leq n \leq 2^{32} - 1$, jenachdem...
 - wie wheel++ vom Compiler für die zugrunde liegende CPU übersetzt und
 - wie oft und wo main() dann von niam() unterbrochen wurde

- $n = 1$ impliziert nicht, dass keine Unterbrechung stattgefunden hat

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

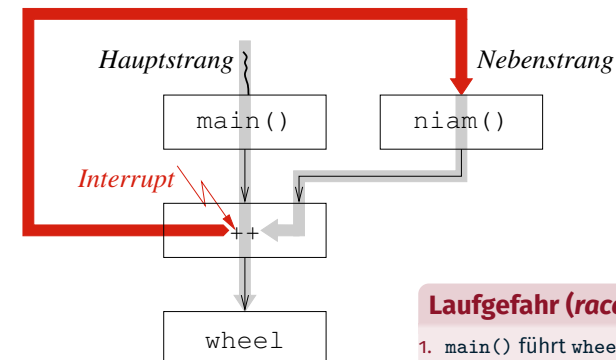
Unterbrechungen bewirken, dass nicht zu jedem Zeitpunkt bestimmt ist, wie es weiter geht. (vgl. auch S. 45)

SP

Nichtsequentialität

B - V.3 / 24

Asynchronität von Unterbrechungen



Laufgefahr (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

SP

Nichtsequentialität

B - V.3 / 25

- `wheel++` ist eine **Elementaroperation** (kurz: Elop) der Ebene₅
 - in Hochsprache formuliert ist diese Aktion scheinbar **atomar, unteilbar**
- nicht zwingend ist `wheel++` auch eine Elop der Ebene₄ (und tiefer)
 - in Assembler-/Maschinensprache formuliert ist diese Aktion **teilbar**

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

- dies trifft insbesondere auch zu auf die dreiphasige Aktion `incl wheel`:
 - den Wert (1) von `wheel` laden, (2) verändern und (3) an `wheel` speichern
- ein **read-modify-write-Zyklus**, teilbar bei Mehr-/Viel(kern)prozessoren
- im **Überlappungsfall** ist die gleichzeitige Ausführung von `wheel++` möglich, was falsche Berechnungsergebnisse liefern kann

³Aktion ist die Ausführung einer Anweisung einer (virtuellen/realen) Maschine.

Unterbrechungsbedingte Überlappungseffekte

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

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

- zweimal `wheel++` durchlaufen (nämlich je einmal in `main()` und `niam()`)
- zweimal gezählt, den Wert von `wheel` aber nur um eins erhöht
- in nichtsequentiellen Programmen ist die Implementierung des Inkrementoperators⁴ (`++`) als **kritischer Abschnitt** aufzufassen

*critical in the sense, that the processes have to be constructed in such a way, that at any moment at most one of the two is engaged in its **critical section**. [1, S. 11]*

⁴Gleiches gilt für den Dekrementoperator, egal ob Prä- oder Postfix.

Nichtsequentialität

Kritischer Abschnitt

Semantikkonforme Elementaroperation

- der **Postfix-Inkrementoperator** (`wheel++`) hat folgende Semantik:
 - **fetch** den Operandenwert (`wheel`) als Ausdruckswert bereitstellen
 - **add** dann dem Operanden den um eins erhöhten (`++`) Wert zuweisen

- ein **„fetch and add“** (FAA), geschieht logisch in einem Schritt
 - um der Laufgefahr vorzubeugen, muss diese Aktion physisch unteilbar sein

- dies leistet `xadd` (x86):


```

1 inline int FAA(int *ref, int val) {
2     int aux = val;
3
4     asm volatile ("xaddl %0, %1"
5                 : "=g" (aux), "=m" (*ref)
6                 : "0" (aux), "m" (*ref)
7                 : "memory", "cc");
8
9     return aux;
10 }
            
```
- Befehl durchsetzen: 4-7
 - *inline assembler* (gcc)
- die nunmehr unteilbare Aktion für `printf()` sicherstellen:


```

12 ...
13 movl $1, %eax
14 xaddl %eax, wheel
15 ...
11 printf("%u\n", FAA(&wheel, 1));
            
```

Definition (Grundblock (*basic block*))

Ein aus einer Anweisungsfolge bestehender Programmabschnitt mit genau einem Eintrittspunkt und einem Austrittspunkt.

Gelingt die Abbildung einer in Hochsprache ausformulierten kritischen Operation auf einen elementaren Maschinenbefehl nicht — weil sie zu komplex ist oder ein äquivalenter Maschinenbefehl nicht existiert, gefunden werden kann oder gesucht werden will —, muss die Operation als kritischer Abschnitt ausformuliert werden.

■ Unterbrechungssperre

- 4 ■ IRQ abwehren
- 5–6 ■ unteilbar, atomar
- 7 ■ IRQ zulassen
- IRQ ■ *interrupt request*

■ Holzhammermethode

- Kollateraleffekte
- Alternative [7, S. 5–15]

```

1 inline int FAA(int *ref, int val) {
2     int aux;
3
4     enter(INTERRUPT_LOCK);
5     aux = *ref;
6     *ref += val;
7     leave(INTERRUPT_LOCK);
8
9     return aux;
10 }
```

vgl. auch S. 50ff

- Einführung
 - Hybride Maschine
 - Teilinterpretation
- Ausnahmen
 - Trap
 - Interrupt
- Programmunterbrechung
 - Ausnahmen
 - Sicherung/Wiederherstellung
- Nichtsequentialität
 - Wettlaufsituation
 - Kritischer Abschnitt
- Zusammenfassung

- zur Einführung wurden **virtuelle Maschinen** erneut aufgegriffen
 - um zu verdeutlichen, dass ein Betriebssystem eine **hybride Maschine** ist
 - die die **Teilinterpretation** von Maschinenprogrammen bewerkstelligt
 - was den Wiedereintritt ins Betriebssystem einschließt: **Ablaufinvarianz**
- das Mittel zur Teilinterpretation ist die **Ausnahme**
 - trap** ■ die synchron, vorhersagbar und reproduzierbar ist
 - interrupt** ■ sich asynchron, unvorhersagbar und nicht reproduzierbar zeigt
- wodurch in ihrer Ausführung unterbrochene Programme verzögert werden
- dabei ist der aktuelle **Laufzeitkontext** invariant zu halten
 - was **Sicherung/Wiederherstellung** des Prozessorstatus zur Folge hat
 - geleistet durch die Befehlssatzebene (CPU) und dem Betriebssystem
- überlappende Programmausführung bringt **Nichtsequentialität**
 - die eine **Wettlaufsituation** bei der Programmausführung bewirken kann
 - der diesbezügliche Programmbereich ist ein **kritischer Abschnitt**
 - in dem sich zu jedem Zeitpunkt nur ein einziger Prozess befinden darf

Zusammenfassung

Bibliographie

Literaturverzeichnis (1)

- [1] DIJKSTRA, E. W.:
Cooperating Sequential Processes / Technische Universiteit Eindhoven.
Eindhoven, The Netherlands, 1965 (EWD-123). –
Forschungsbericht. –
(Reprinted in *Great Papers in Computer Science*, P. Laplante, ed., IEEE Press, New York, NY, 1996)
- [2] KLEINÖDER, J. ; SCHRÖDER-PREIKSCHAT, W. :
Maschinenprogramme.
In: [4], Kapitel 5.2
- [3] KLEINÖDER, J. ; SCHRÖDER-PREIKSCHAT, W. :
Speichervirtualisierung.
In: [4], Kapitel 12.3

Literaturverzeichnis (2)

- [4] KLEINÖDER, J. ; SCHRÖDER-PREIKSCHAT, W. ; LEHRSTUHL INFORMATIK 4 (Hrsg.):
Systemprogrammierung.
FAU Erlangen-Nürnberg, 2015 (Vorlesungsfolien)
- [5] KOPETZ, H. :
Real-Time Systems: Design Principles for Distributed Embedded Applications.
Kluwer Academic Publishers, 1997. –
ISBN 0-7923-9894-7
- [6] SCHRÖDER-PREIKSCHAT, W. ; LEHRSTUHL INFORMATIK 4 (Hrsg.):
Concurrent Systems – Nebenläufige Systeme.
FAU Erlangen-Nürnberg, 2014 (Vorlesungsfolien)

Literaturverzeichnis (3)

- [7] SCHRÖDER-PREIKSCHAT, W. :
Guarded Sections.
In: [6], Kapitel 10
- [8] SCHRÖDER-PREIKSCHAT, W. :
Processes.
In: [6], Kapitel 3

Anhang

Interrupt

- Unterbrechungen verursachen **Zittern** (*jitter*) im Ablaufverhalten, machen Programme **nicht-deterministisch**
 - nicht zu jedem Zeitpunkt ist bestimmt, wie weitergefahren wird
- je nach „Räumlichkeit“ des unterbrochenen (P_i) und behandelnden (P_h) Programms ergeben sich verschiedene Ausprägungen
 - getrennt**
 - P_i auf Ebene₃, P_h auf Ebene₂
 - in räumlicher Hinsicht hat P_h keinen Einfluss auf P_i
 - gemeinsam**
 - P_i und P_h zusammen auf Ebene₃ oder Ebene₂
 - in räumlicher Hinsicht kann P_h einen Einfluss auf P_i haben
 - P_i und P_h bilden ein **nichtsequentielles Programm**
- aber in beiden Fällen wird P_i um die jeweilige Dauer von P_h verzögert
- in zeitlicher Hinsicht beeinflusst die Unterbrechungsart „*interrupt*“ jedes Programm, dessen Ausführung dadurch ausgesetzt wird
- Dies ist kritisch für **echtzeitabhängige Programme**
 - deren korrektes Verhalten hängt nicht nur von logischen Ergebnissen von Berechnungen ab, sondern auch von dem **physikalischen Zeitpunkt** der Erzeugung und Verwendung der Berechnungsergebnisse. [5]

Unteilbarkeit

Definition (in Anlehnung an den Duden)

Das Unteilbarsein, um etwas als Einheit oder Ganzheit in Erscheinung treten zu lassen.

- eine Frage der „Distanz“ des Betrachters (Subjekts) auf ein Objekt
 - **Aktion** auf höherer, **Aktionsfolge** auf tieferer Abstraktionsebene
- | Ebene | Aktion | Aktionsfolge |
|-------|--------------------------|--|
| 5 | <code>i++</code> | |
| 4-3 | <code>incl i*</code> | <code>movl i,%r</code>
<code>addl \$1,%r*</code> |
| | <code>addl \$1,i*</code> | <code>movl %r,i</code> |
| 2-1 | | * read from memory into accumulator
* modify contents of accumulator
* write from accumulator into memory |
- typisch für den Komplexbefehl eines „abstrakten Prozessors“ (C, CISC)

Anhang

Wettlaufsituation

Unteilbarkeit komplexer Operationen

Ganzheit oder Einheit einer Aktionsfolge, deren Einzelaktionen alle scheinbar gleichzeitig stattfinden (d.h., synchronisiert sind)

- wesentliche Eigenschaft für eine **atomare Operation**⁶
 - die logische Zusammengehörigkeit von Aktionen in zeitlicher Hinsicht
 - wodurch die Aktionsfolge als **Elementaroperation** (ELOP) erscheint

Beispiele von Aktionen zum Inkrementieren eines Zählers:

■ Ebene $5 \mapsto 3$	■ Ebene $3 \mapsto 2$		
C/C++	ASM	ASM	ISA
1 <code>i++;</code>	1 <code>movl i, %eax</code>	1 <code>incl i</code>	1 <code>read A from <i></code>
	2 <code>addl \$1, %eax</code>		2 <code>modify A by 1</code>
	3 <code>movl %eax, i</code>		3 <code>write A to <i></code>

- die Inkrementierungsaaktionen (`i++`, `incl`) sind nur **bedingt unteilbar**
 - unterbrechungsfreier Betrieb (Ebene $5 \mapsto 3$), Uniprozessor (Ebene $3 \mapsto 2$)
 - **Problem: zeitliche Überlappung** von Aktionsfolgen hier gezeigter Art von (eig.) **atomar** „unteilbar“!

Anhang

Kritischer Abschnitt

Unteilbarer Grundblock Ebene₄

ASM/x86

- Übersetzung von `faa.c` (vgl. S. 37) mit `-S`

```
1 FAA:
2   movl 4(%esp), %edx
3   cli
4   movl (%edx), %eax
5   movl 8(%esp), %ecx
6   addl %eax, %ecx
7   movl %ecx, (%edx)
8   sti
9   ret
```

oben `-D"int_t=long int"`

3-8 unteilbare Sequenz

rechts `-D"int_t=long long int"`

8-15 unteilbare Sequenz

`cli` clear interrupt flag, abschalten

`sti` set interrupt flag, einschalten

```
1 FAA:
2   subl $8, %esp
3   movl %ebx, (%esp)
4   movl 16(%esp), %ecx
5   movl %esi, 4(%esp)
6   movl 20(%esp), %ebx
7   movl 12(%esp), %esi
8   cli
9   movl (%esi), %eax
10  movl 4(%esi), %edx
11  addl %eax, %ecx
12  adcl %edx, %ebx
13  movl %ecx, (%esi)
14  movl %ebx, 4(%esi)
15  sti
16  movl (%esp), %ebx
17  movl 4(%esp), %esi
18  addl $8, %esp
19  ret
```

Ein-/Austrittsprotokolle

Synchronisation von Anweisungsfolgen

```
1 typedef enum safeguard {INTERRUPT_LOCK} safeguard_t;
2
3 extern void panic(char *);
4
5 inline void enter(safeguard_t type, ...) {
6     if (type == INTERRUPT_LOCK)
7         asm volatile ("cli" : : : "cc");
8     /* more safeguard variants... */
9     else
10        panic("bad safeguard");
11 }
12
13 inline void leave(safeguard_t type, ...) {
14     if (type == INTERRUPT_LOCK)
15         asm volatile ("sti" : : : "cc");
16     /* more safeguard variants... */
17     else
18        panic("bad safeguard");
19 }
```

SP

Anhang

B - V.3 / 38