

— BS // —

Nebenläufigkeit

Überblick

- kritischer Bereich „Zählen“ 2

- Uniprocessorsystem
 - * unterbrechungsbehaftetes System
 - * mehrfädiges System
- Multiprocessorsystem

- kritischer Bereich „Listenverwaltung“ 10

Kritischer Bereich „Zählen“

Gegeben sei nebenstehendes C-Programm. Ferner sei die „Benutzung“ asynchroner Programmunterbrechungen (*interrupts*) vorausgesetzt und ihr Auftreten angenommen:

- Ist der korrekte Ablauf jederzeit sichergestellt?

```
int bar;  
void foo () {  
    ++bar;  
}
```

- Ist das Programm als portabel anzusehen, insbesondere wenn die Übertragbarkeit von CISC- nach RISC-Technologie und umgekehrt zu berücksichtigen wäre? Läuft es gar unverändert auf (*shared memory*) Multiprozessorssystemen?

Das Programm wird übersetzt mit `gcc -O6 -fomit-frame-pointer -S` und der für CISC- und RISC-Prozessoren erzeugte Assemblercode wird analysiert.

„CISC-Zähler“ (1)

```
_bar:
.space 4
_foo:
    incl_bar
ret
```

- es erfolgte die 1:n, n = 1, Übersetzung der C-Elop „++“
 - lesen, manipulieren und schreiben mit einem Befehl
- die unteilbare Ebene₅-Elop bleibt unteilbar auf Ebene₄
 - vorausgesetzt die Rechnerarchitektur ist passend

- auf Speicherbussebene ($l, l \leq \text{Ebene}_1$) kann die Elop ein **kritischer Befehl** sein
 - die Manipulation des Wertes von bar kann nur in der ALU geschehen
 - dazu ist der Wert vorher zu lesen und das Ergebnis danach zu schreiben
- die CPU führt einen *read-modify-write*-Zyklus aus — der teilbar sein kann

„CISC-Zähler“ (2)

Multiprozessor/x86

Prozessor 1	Prozessor 2		bar
	Zyklus	ATU	
read	42	42	42
modify	43	42	42
write	43	43	43

- **Parallelität** kann zu überlappenden Zugriffen auf dieselbe Speicherzelle führen
- in Parallelrechnern mit gemeinsamen Speicher (*shared memory*) ist **inc1** teilbar

„RISC-Zähler“ (1)

sparc

```
bar:      .skip 4
foo:      sethi %hi(bar), %g1
          ld      [%g1+%l0(bar)], %o0
          add     %o0, 1, %o1
          retl
          st      %o1, [%g1+%l0(bar)]
```

- 1:n, n = 4, Übersetzung der Elop „++“:
 1. Lesen des Operanden
 2. Manipulation des Wertes
 3. Schreiben des Operanden
- die Ebene₅-Elop ist teilbar auf Ebene₄
 - sie ist keine atomare Operation
- der **kritische Bereich** darf nicht überlappt zur Ausführung kommen [warum?]

„RISC-Zähler“ (2)

sparc

überlapptes Programm		überlappendes Programm	
Elop		Elop	
%00	%01	%00	%01
ld [%g1+%l0(bar)],%00	ld [%g1+%l0(bar)],%00	ld [%g1+%l0(bar)],%00	ld [%g1+%l0(bar)],%00
add %00,1,%01	add %00,1,%01	add %00,1,%01	add %00,1,%01
42	42	42	42
-	43	-	43
42	42	42	42
42	42	42	42
42	43	42	43
42	43	42	43
42	43	42	43
42	43	42	43
42	43	42	43
42	43	42	43

- Überlappung bringt die mehrmalige Ausführung der Programmsequenz mit sich
- bei n -maligem Durchlauf hätte sich bar demzufolge um n erhöhen müssen

„RISC-Zähler“ (3)

sparc

Prozess 1		Prozess 2		bar	
Elop		Elop			
%00	%01	%00	%01		
ld [%g1+%l0(bar)],%00	42	-	42	ld [%g1+%l0(bar)],%00	42
add %00,1,%01	42	43	42	add %00,1,%01	42
st %01, [%g1+%l0(bar)]	43	43	42	st %01, [%g1+%l0(bar)]	43
					43
					43

- als Folge einer Unterbrechung könnte Prozess 1 von Prozess 2 verdrängt werden
- die Verdrängung wird durch ein präemptives scheduling-Verfahren bewirkt
- die Unterbrechungsbehandlung selbst durchläuft den kritischen Bereich nicht

Unbeständiger „CISC-Zähler“ (1)

x86

- es gibt einige Fälle, in denen Variablenwerte nicht „gecacht“ werden dürfen
 - (memory mapped) E/A-Register, Sperren (*spin locks*), . . . , ggf. auch Zähler

- eine Typverweiterung instruiert den Übersetzer
 - `volatile int bar; /* don't cache */` – jeder bar-Zugriff geht zum Speicher
 - die 1:n, n = 3, Übersetzung der C-Elap „++“
 - eine auf Ebene₅ „unteilbare“ Elap . . .
 - . . . repräsentiert sich **teilbar auf Ebene₄**

```
bar:
.space 4
foo:
movl _bar,%eax
incl %eax
movl %eax,-bar
```

Unbeständiger „CISC-Zähler“ (2)

x86

überlappendes Programm		Elop		überlappendes Programm	
		Elop		-bar	
		%eax		%eax	
movl _bar,%eax	42	movl _bar,%eax	42	incl %eax	43
incl %eax	43	incl %eax	43	movl %eax,_bar	43
movl %eax,_bar	43	movl %eax,_bar	43		43

- das beim RISC gebräuchliche *load/store*-Modell wird dem CISC auferlegt
- mit den bekanntesten RISC-Problemen sich überlappende Programmsequenzen
- die Identifikation kritischer unbeständiger Variablen ist alles andere als trivial

Kritischer Bereich „Listenverwaltung“

Gegeben seien folgende C++-Klassen und die „Benutzung“ von Interrupts:

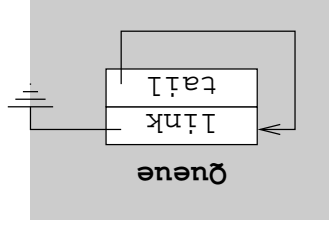
```
class Chain {  
protected:  
    Chain* link;  
public:  
    Chain () { link = 0; }  
    Chain* operator = (Chain* item) {  
        return link = item;  
    }  
    operator Chain* () const {  
        return link;  
    }  
};
```

```
class Queue {  
    Chain* tail;  
public:  
    Queue () { tail = this; }  
    Chain* operator = (Chain& item) {  
        item = 0;  
        return tail = *tail = &item;  
    }  
    operator Chain* () {  
        Chain* item;  
        if ((item = link) && !(link = *item))  
            tail = this;  
        return item;  
    }  
};
```

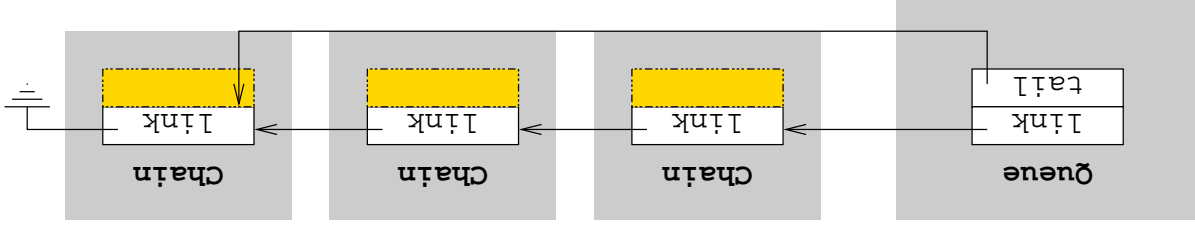
„Schlangenkette“

Idee Das Einfügen der Kettenlieder (Chain) soll in deterministischer Zeit erfolgen. Dazu zeigt tail immer auf den Verkettungszeiger eines Schlangenelements. Bei leerer Schlange zeigt tail auf den Kopfzeiger link (= NIL).

Der Kopfzeiger ist selbst ein Kettenglied (Chain). Damit könnte Entnahme des letzten Elements aus der Schlange, wird link NIL und tail wird auf link gesetzt. **Trick ist, dass die Schlange**



praktisch, d.h. aus Sicht von tail, **niemals leer ist**. Das erste Element bedarf keiner besonderen Behandlung.



“CISC-Schlange“ (1)

```
}  
    q = e;  
void put (Queue& q, Chain& e) {
```

.....

```
-put-FR5QueueR5Chain:  
    movl 4(%esp), %eax  
    movl 8(%esp), %edx  
    movl $0, (%edx)  
    movl 4(%eax), %ecx  
    movl %edx, (%ecx)  
    movl %edx, 4(%eax)  
    ret
```

x86

“CISC-Schlange“ (2)

```
Chain* get (Queue& q) {  
    return q;  
}
```

```
get_FR5Queue:  
    movl 4(%esp), %ecx  
    movl (%ecx), %edx  
    testl %edx, %edx  
    je L21  
    movl (%edx), %eax  
    movl %eax, (%ecx)  
    testl %eax, %eax  
    jne L21  
    movl %ecx, 4(%ecx)  
L21:  
    movl %edx, %eax  
    ret
```

x86

Nebenläufigkeitsproblem (1)

x86 vs. C/C++

x86 Ebene₄ (Assembler) bzw. Ebene₂ (konventionelle Maschine)

- wie `put()`/`get()` zeigt, ist der generierte Code kritisch → pp. 12/13
- der Versuch einer Problemlösung auf dieser Ebene ist jedoch unzuverlässig

C/C++ Ebene₅ (Hochsprache)

- die Queue-Operatoren = und **Chain*** sind bereits kritisch → p. 10
- die Problemanalyse muss sich daher zunächst der Ebene₅ zuwenden
- ebenso sollte die Problemlösung für Ebene₅ angestrebt werden¹

¹Erst wenn für diese Ebene keine (zufriedenstellende) Lösung zu finden ist, ist nach Lösungsansätzen unter Zuhilfenahme tieferer Ebenen zu suchen.

Nebenläufigkeitsproblem (2)

C/C++

- die **Problemanalyse** muss vier verschiedene Szenarien betrachten:

1. put() bzw. = unterbricht/überlappt . . .

– get() bzw. **Chain***

– sich selbst

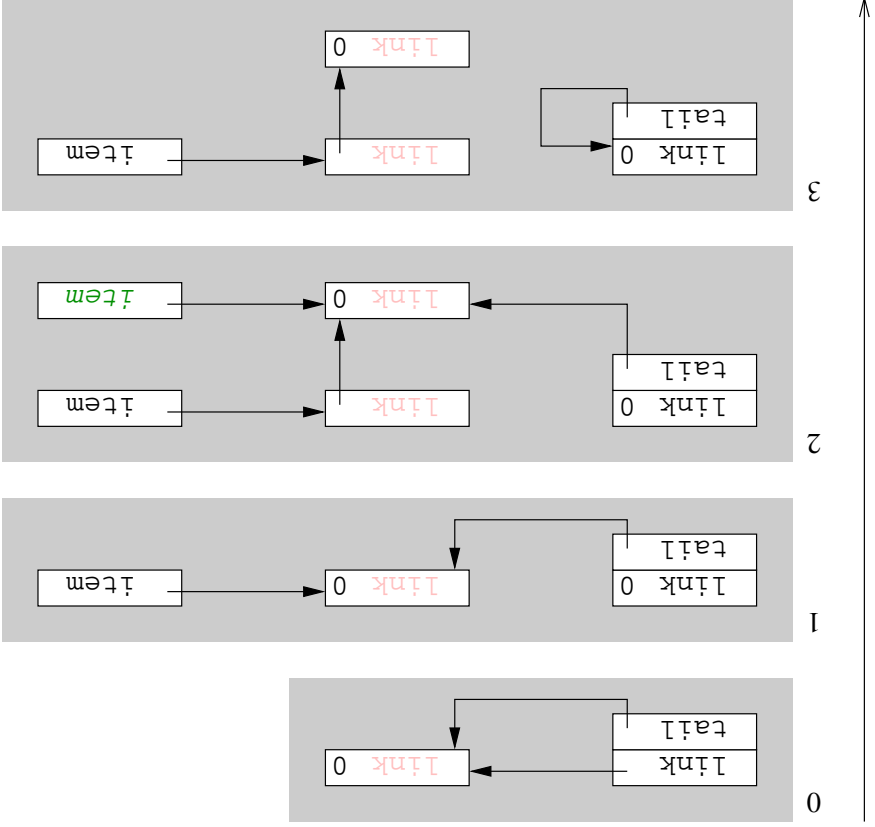
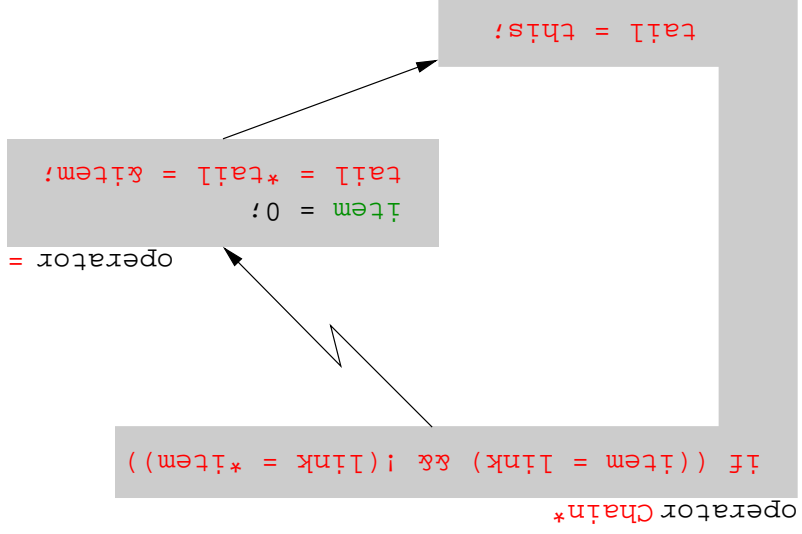
2. get() bzw. **Chain*** unterbricht/überlappt . . .

– put() bzw. =

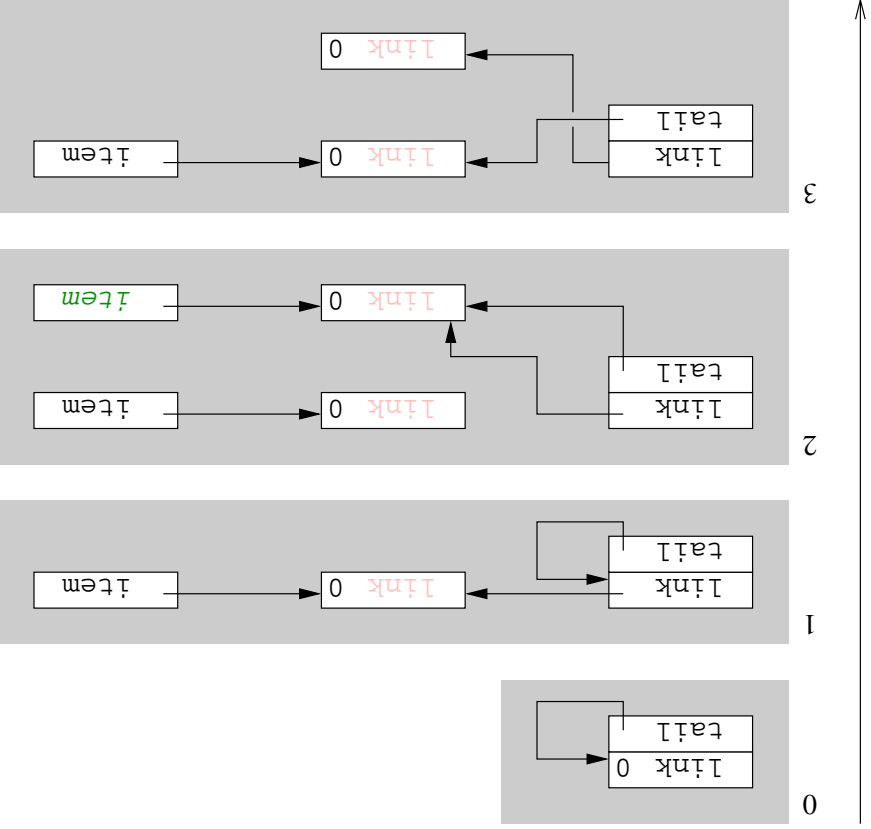
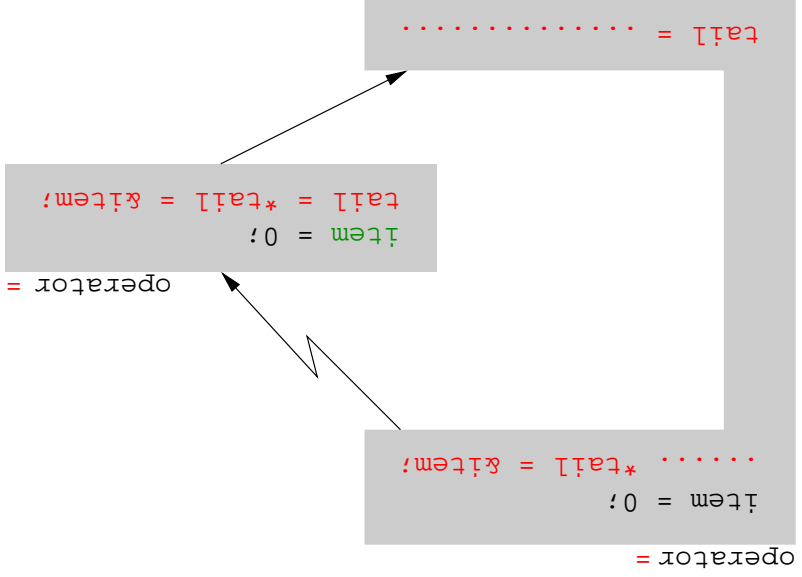
– sich selbst

- die Lösung der Probleme in C/C++ greift bis nach unten (Ebene₂ und tiefer)

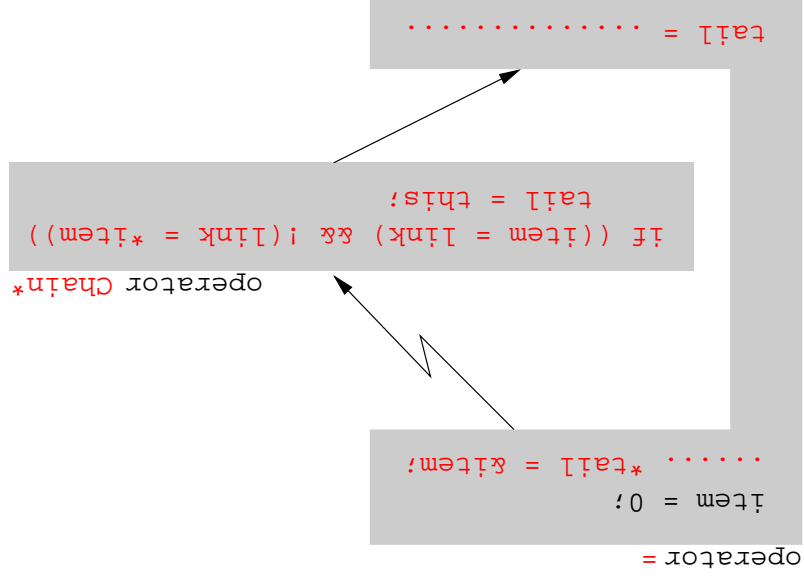
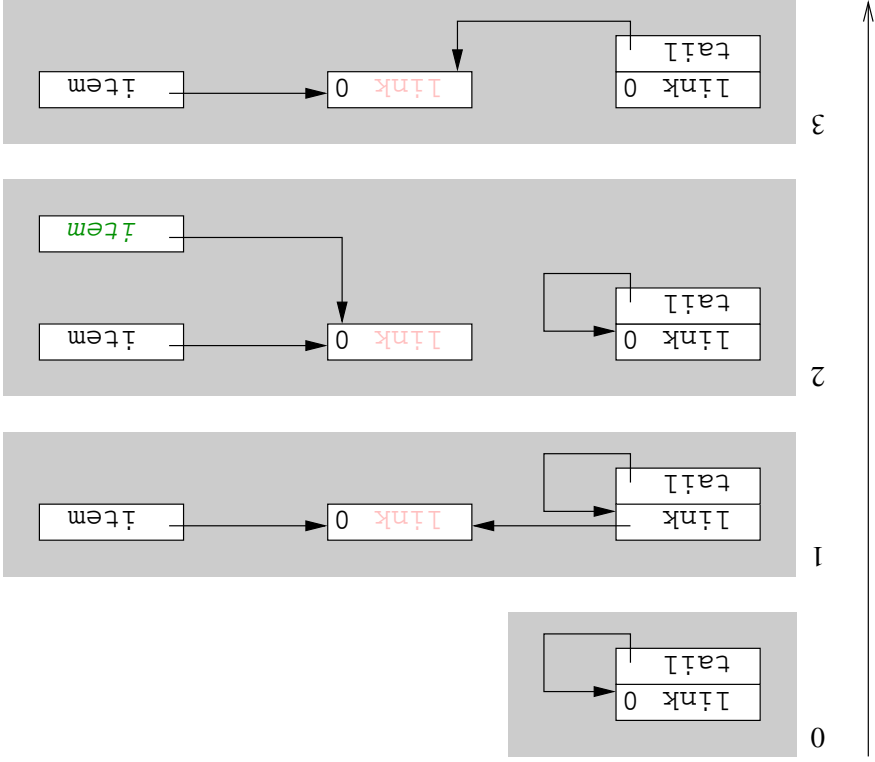
= unterbricht/überlappt Chain*



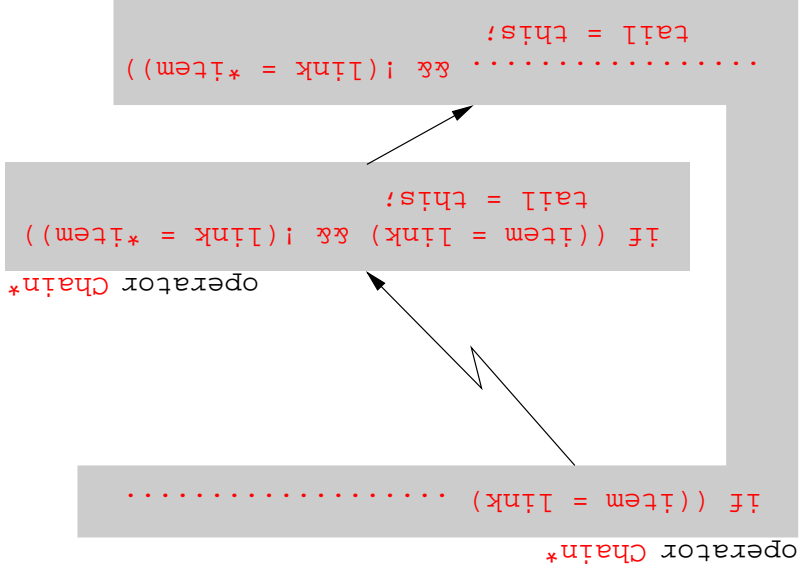
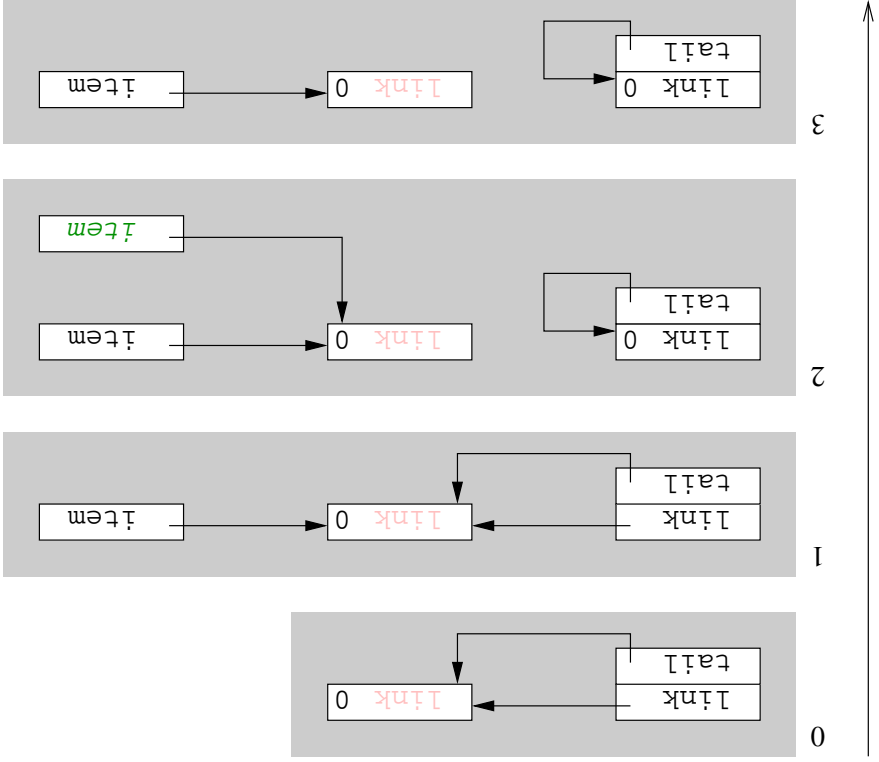
= unterbricht/überlappt sich selbst



Chain* unterbricht/überlappt =



Chain* unterbricht/überlappt sich selbst



Zusammenfassung

- die Syntax einer **Elementaroperation** (Elop) sagt nichts aus zur Unteilbarkeit
 - die Semantik könnte Aussagen zum Verhalten bei Nebenläufigkeit treffen
 - Realität ist, dass die wenigsten (CISC-) Befehle wirklich atomar sind
- Nebenläufigkeit ist eine **nicht-funktionale Eigenschaft** des Systems
 - nicht in allen Fällen treten Nebenläufigkeitsprobleme auch wirklich auf:
 - * wenn Nebenläufigkeit in der gegebenen Konfiguration unmöglich ist
 - * wenn ein Zugriff auf gemeinsame Datenbestände überhaupt nicht erfolgt
 - * bzw. wenn sich das Zugriffsmuster als grundsätzlich konfliktfrei erweist
 - in dem Sinne ist es zweckmäßig, Unteilbarkeit nur bedingt zu implementieren
- **Mechanismen** zur „unteilbaren Programmausführung“ sind bereitzustellen

Ausblick

- die **Problemlösung** geht auf verschiedene Verfahren und Techniken zurück:
 - blockierende Synchronisation
 - nicht-blockierende, wartebehaftete Synchronisation
 - nicht-blockierende, wartefreie Synchronisation
 - Synchronisation mit und ohne speziellen Ebene²-Operationen
 - Lösungen für Uni- und/oder (*shared memory*) Multiprozessorssysteme
- die „Qual der Wahl“ ist bestimmt durch den jeweiligen Anwendungsfall