

Nebenläufigkeit

Betriebssysteme, ©Wolfgang Schröder-Preikschat

Überblick

- kritischer Bereich „Zählen“ 2
 - Uniprozessorsystem
 - * unterbrechungsbehaftetes System
 - * mehrfädiges System
 - Multiprozessorsystem

- kritischer Bereich „Listenverwaltung“ 10

Kritischer Bereich „Zählen“

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

```
int bar;

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

- Ist der korrekte Ablauf jederzeit sichergestellt?
- 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*) Multiprozessorsystemen?

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

„CISC-Zähler“ (1)

x86

```
_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 Speicherbusebene (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	ALU	Zyklus	ALU	
<i>read</i>	42			42
<i>modify</i>	43	<i>read</i>	42	42
<i>write</i>	43	<i>modify</i>	43	43
		<i>write</i>	43	43

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

„RISC-Zähler“ (1)

sparc

```
bar:
    .skip 4
foo:
    sethi %hi(bar),%g1
    ld    [%g1+%lo(bar)],%o0
    add  %o0,1,%o1
    retl
    st   %o1, [%g1+%lo(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			bar
<i>Elop</i>	%o0	%o1	<i>Elop</i>	%o0	%o1	
ld [%g1+%lo(bar)],%o0	42	–				42
add %o0,1,%o1	42	43				42
			ld [%g1+%lo(bar)],%o0	42	–	42
			add %o0,1,%o1	42	43	42
			st %o1,[%g1+%lo(bar)]	42	43	43
st %o1,[%g1+%lo(bar)]	42	43				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
<i>Elop</i>	%o0	%o1	<i>Elop</i>	%o0	%o1	
ld [%g1+%lo(bar)],%o0	42	–				42
			ld [%g1+%lo(bar)],%o0	42	–	42
			add %o0,1,%o1	42	43	42
			st %o1,[%g1+%lo(bar)]	42	43	43
add %o0,1,%o1	42	43				43
st %o1,[%g1+%lo(bar)]	42	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 „gecached“ werden dürfen
 - (*memory mapped*) E/A-Register, Sperren (*spin locks*), . . . , ggf. auch Zähler

```

_bar:
    .space 4

_foo:
    movl _bar,%eax
    incl %eax
    movl %eax,_bar
    ret
    
```

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

Unbeständiger „CISC-Zähler“ (2)

x86

überlapptes Programm		überlappendes Programm		_bar
<i>Elop</i>	%eax	<i>Elop</i>	%eax	
movl _bar,%eax	42	movl _bar,%eax	42	42
		incl %eax	43	42
		movl %eax,_bar	43	43
incl %eax	43			43
movl %eax,_bar	43			43

- das beim RISC gebräuchliche *load/store*-Modell wird dem CISC auferlegt
 - mit den bekannten RISC-Problemen sich überlappender 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;
    }
};
```

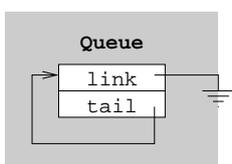
```
class Queue : public Chain {
    Chain* tail;
public:
    Queue () { tail = this; }

    Chain* operator = (Chain& item) {
        item = 0;
        return tail = *tail = &item;
    }

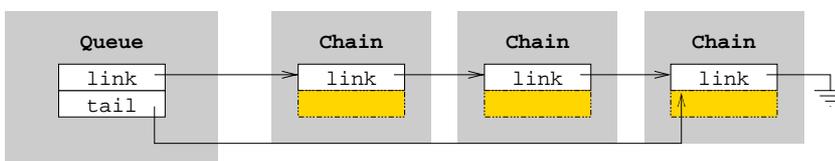
    operator Chain* () {
        Chain* item;
        if ((item = link) && !(link = *item))
            tail = this;
        return item;
    }
};
```

„Schlangenkette“

Idee Das Einfügen der Kettenglieder (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 der Schlangenkopf ebenfalls Element einer Schlange sein. Bei der 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)

x86

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

„CISC-Schlange“ (2)

x86

```
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
```

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 unzweckmäßig

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

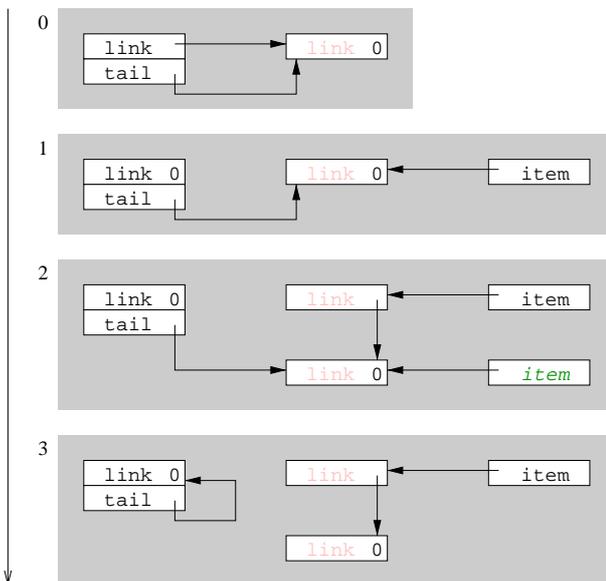
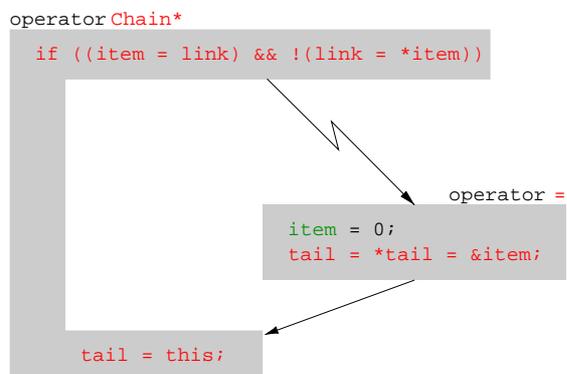
¹Erst wenn für diese Ebene keine (zufriedenstellende) Lösung zu finden ist, ist nach Lösungsansätzen unter Zuhilfenahme tieferer Ebenen zu suchen.

Nebenläufigkeitsproblem (2)

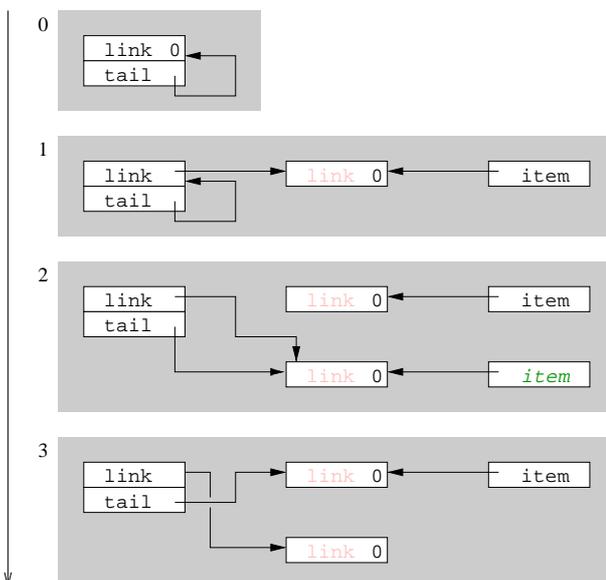
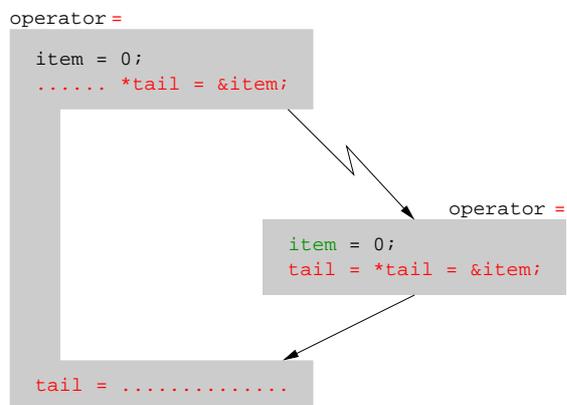
C/C++

- 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)

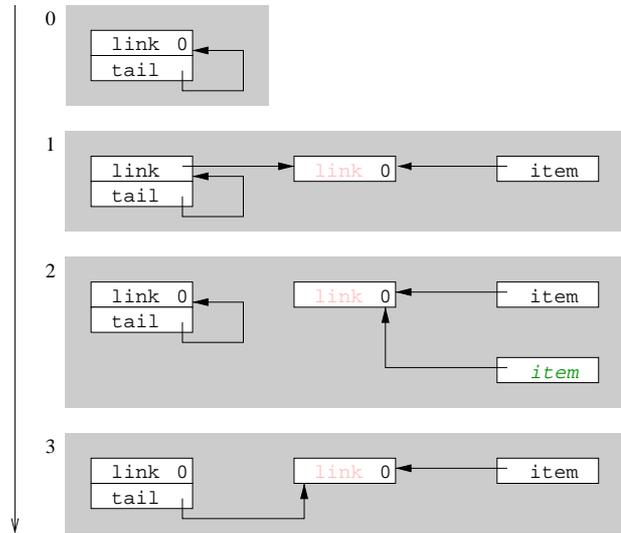
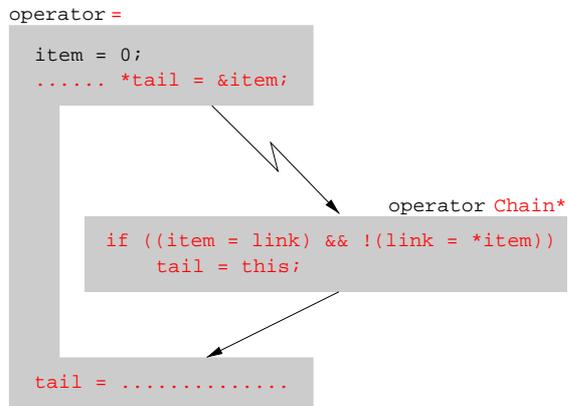
= unterbricht/überlappt Chain*



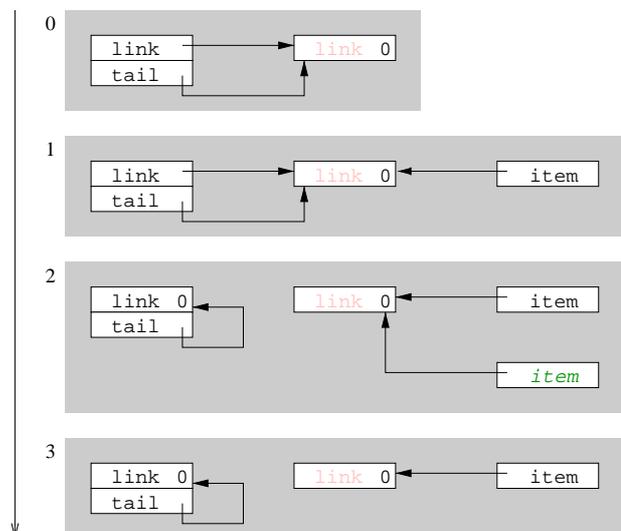
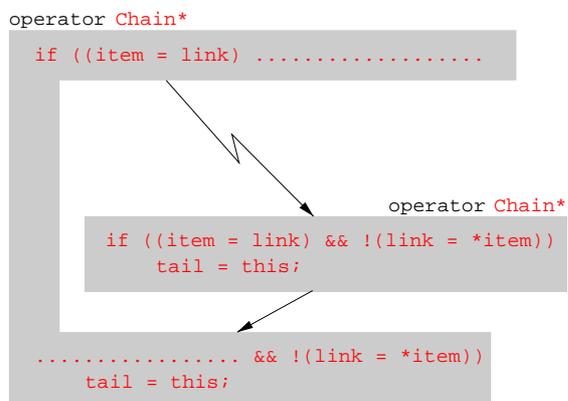
= unterbricht/überlappt sich selbst



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, wartebefahfete Synchronisation
 - nicht-blockierende, wartefreie Synchronisation
 - Synchronisation mit und ohne speziellen Ebene₂-Operationen
 - Lösungen für Uni- und/oder (*shared memory*) Multiprozessorsysteme
- die „Qual der Wahl“ ist bestimmt durch den jeweiligen Anwendungsfall