

# Betriebssysteme (BS)

## VL 5 – Unterbrechungen, Synchronisation

**Daniel Lohmann**

Lehrstuhl für Informatik 4  
Verteilte Systeme und Betriebssysteme

Friedrich-Alexander-Universität  
Erlangen Nürnberg

WS 13 – 13. November 2013



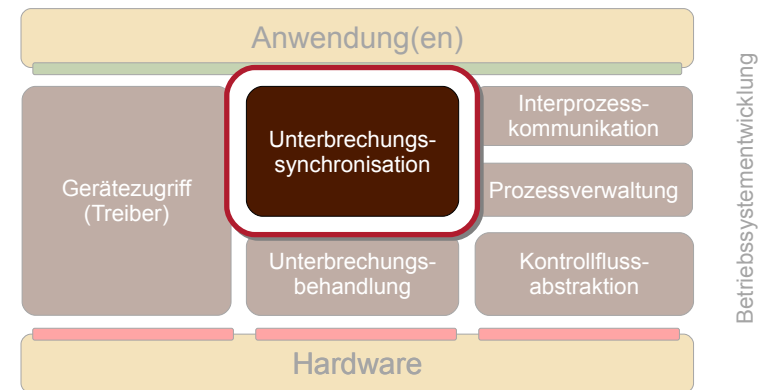
[http://www4.cs.fau.de/Lehre/WS13/V\\_BS](http://www4.cs.fau.de/Lehre/WS13/V_BS)

### Agenda

- Einleitung
- Prioritätsebenenmodell
- Harte Synchronisation
- Weiche Synchronisation
- Prolog/Epilog-Modell
- Zusammenfassung
- Referenzen



## Überblick: Einordnung dieser VL



### Agenda

- Einleitung
  - Motivation
  - Erstes Fazit
- Prioritätsebenenmodell
- Harte Synchronisation
- Weiche Synchronisation
- Prolog/Epilog-Modell
- Zusammenfassung
- Referenzen



# Motivation: Konsistenzprobleme

## Beispiel 1: Systemzeit

- hier schlummert möglicherweise ein Fehler ...
  - das Lesen von global\_time erfolgt nicht notwendigerweise atomar!

```

32-Bit-CPU:      16-Bit-CPU (little endian):
mov global_time, %eax      mov global_time, %r0; lo
                           mov global_time+2, %r1; hi
    
```

- kritisch ist eine Unterbrechung zwischen den beiden Leseinstruktionen bei der 16-Bit-CPU

Instruktion	global_time hi / lo	Resultat r1 / r0
?	002A   FFFF	?   ?
mov global_time, %r0	002A   FFFF	?   FFFF
/* Inkrementierung */	002B   0000	?   FFFF
mov global_time+2, %r1	002B   0000	002B   FFFF

## Beispiele aus der letzten Vorlesung

### Beispiel 2: Ringpuffer

auch die Pufferimplementierung ist kritisch ...

Ausführung Zustand

```

char consume() {
int elements;
char nextin;
char nextout;
void produce(char data) {
int elements;
if (elements == SIZE) return;
buf[nextin] = data;
nextin++; nextin % SIZE;
occupied = elements + 1; // 2
}
occupied = elements - 1; // 0
return result; // 'a'
}
    
```

**Problem:** Beim nächsten Aufruf von consume() hat occupied den Wert 0. Damit wird kein Ergebnis geliefert. Die Datenstruktur ist in einem inkonsistenten Zustand.

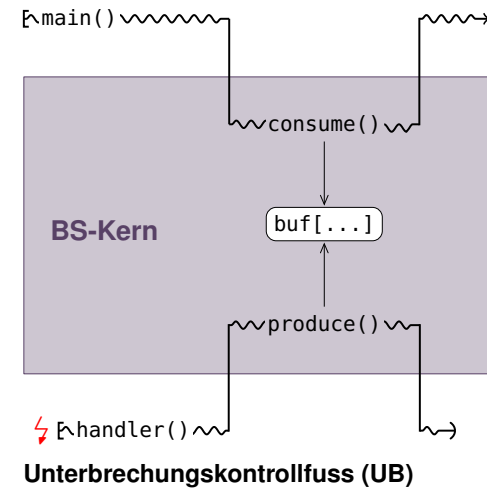
# Motivation: Ursache

Kontrollflüsse  
"von oben"

"begegnen"  
sich im Kern

und "von unten"

## Anwendungskontrollfluss (A)



## Unterbrechungskontrollfluss (UB)

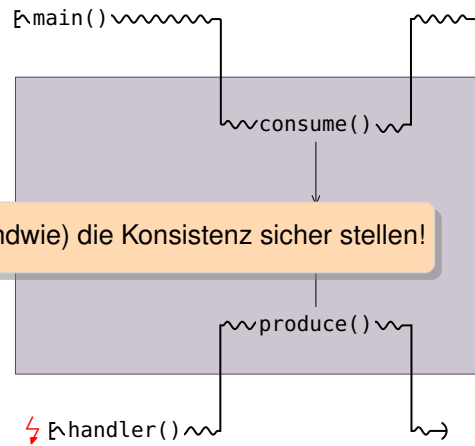
# Motivation: Ursache

Kontrollflüsse  
"von oben"

"begegnen" sich im Kern  
Wir müssen (irgendwie) die Konsistenz sicher stellen!

und "von unten"

## Anwendungskontrollfluss (A)



## Unterbrechungskontrollfluss (UB)

# Naiver Lösungsansatz

- Zweiseitige Synchronisation
  - gegenseitiger Ausschluss durch Mutex, Spin-Lock, ... (vgl. [SP])
  - wie zwischen zwei Prozessen

## Anwendungskontrollfluss (A)

```

char consume() {
mutex.lock();
...
char result = buf[nextout++];
...
mutex.unlock();
return result;
}

void produce(char data) {
mutex.lock();
...
buf[nextin++] = data;
...
mutex.unlock();
}
    
```

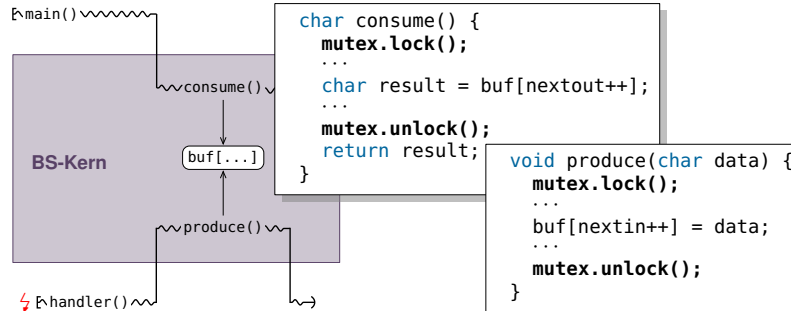
## Unterbrechungskontrollfluss (UB)

## Naiver Lösungsansatz

### Zweiseitige Synchronisation

- gegenseitiger Ausschluss durch Mutex
  - wie zwischen zwei Prozessen
- Zweiseitige Synchronisation funktioniert **natürlich nicht!**

#### Anwendungskontrollfluss (A)



#### Unterbrechungskontrollfluss (UB)

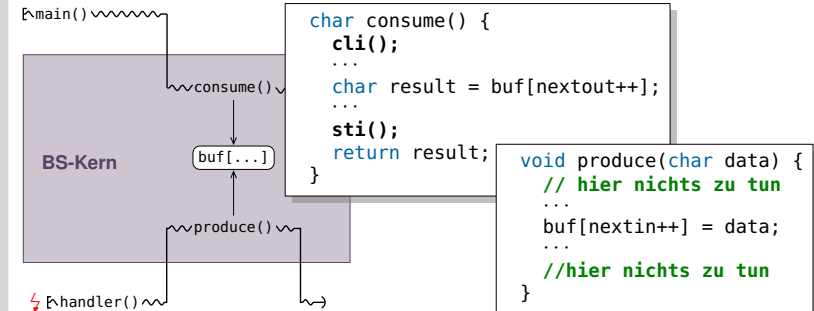


## Besserer Lösungsansatz

### Einseitige Synchronisation

- Unterdrückung der Unterbrechungsbehandlung im Verbraucher
- Operationen `disable_interrupts()` `enable_interrupts()` (im Folgenden o. B. d. A. in „Intel“-Schreibweise: `cli()` / `sti()`)

#### Anwendungskontrollfluss (A)



#### Unterbrechungskontrollfluss (UB)

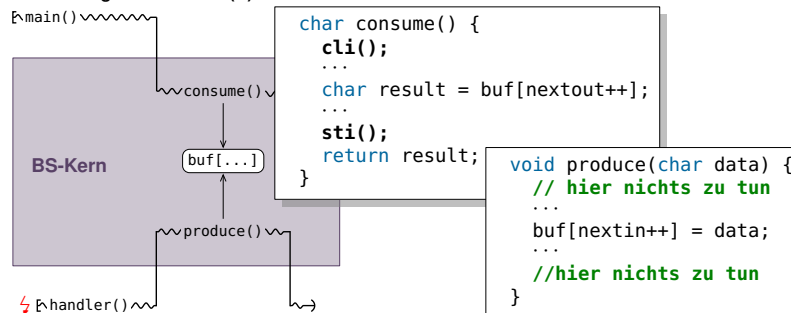


## Besserer Lösungsansatz

### Einseitige Synchronisation

- Unterdrückung der Unterbrechung: Einseitige Synchronisation funktioniert. [Warum?]
- Operationen `disable_interrupts()` `enable_interrupts()` (im Folgenden o. B. d. A. in „Intel“-Schreibweise: `cli()` / `sti()`)

#### Anwendungskontrollfluss (A)



#### Unterbrechungskontrollfluss (UB)



## Erstes Fazit

- Konsistenzsicherung zwischen
  - Anwendungskontrollfluss (A) und Unterbrechungsbehandlung (UB) muss **anders erfolgen** als zwischen Prozessen
- Die Beziehung zwischen A und UB ist **asymmetrisch**
  - Es handelt sich um „verschiedene Arten“ von Kontrollflüssen
  - UB **unterbricht** Anwendungskontrollfluss
    - implizit, an beliebiger Stelle
    - hat immer Priorität, läuft durch (*run-to-completion*)
  - A kann UB **unterdrücken** (besser: *verzögern*)
    - explizit, mit `cli/sti` (Grundannahme 5 aus VL 4)
- Synchronisation / Konsistenzsicherung erfolgt **einseitig**

Diese Tatsachen müssen wir **beachten!**

(Das heißt aber auch: Wir können sie **ausnutzen**)



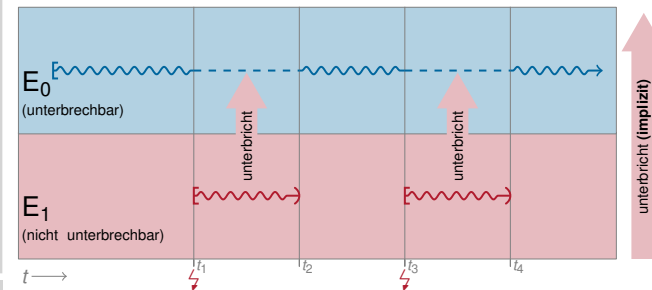
# Agenda

- Einleitung
- Prioritätsebenenmodell
  - Grundbegriffe
  - Verallgemeinerung
  - Konsistenzsicherung
- Harte Synchronisation
- Weiche Synchronisation
- Prolog/Epilog-Modell
- Zusammenfassung
- Referenzen



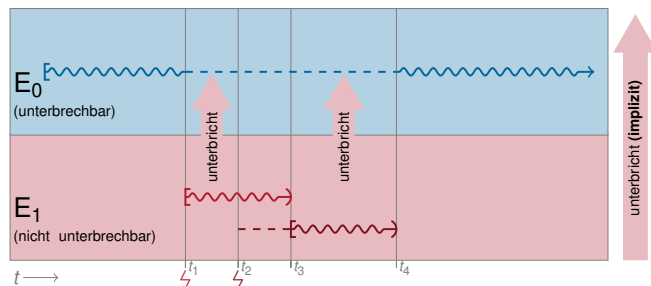
# Prioritätsebenenmodell

- $E_0$  sei die Anwendungskontrollfluss-Ebene (A)
  - Kontrollflüsse dieser Ebene sind **jederzeit unterbrechbar** (durch  $E_1$ -Kontrollflüsse, implizit)
- $E_1$  sei die Unterbrechungsbehandlungs-Ebene (UB)
  - Kontrollflüsse dieser Ebene sind **nicht unterbrechbar** (durch  $E_{0/1}$ -Kontrollflüsse, implizit)



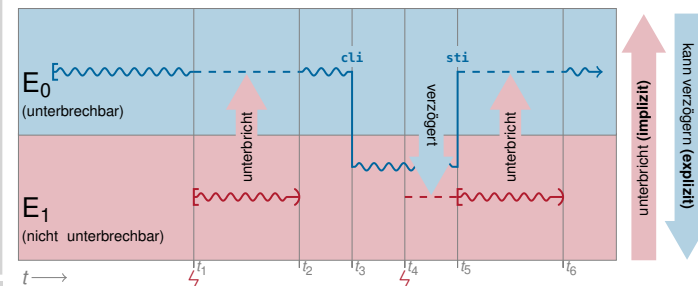
# Prioritätsebenenmodell

- Kontrollflüsse derselben Ebene werden **sequentialisiert**
  - Sind mehrere Kontrollflüsse in einer Ebene anhängig, so werden diese **nacheinander** abgearbeitet (*run-to-completion*)
    - damit ist auf jeder Ebene höchstens ein Kontrollfluss aktiv
  - Die Sequentialisierungsstrategie selber ist dabei beliebig
    - FIFO, LIFO, nach Priorität, zufällig, ...
    - Für  $E_1$ -Kontrollflüsse auf dem PC implementiert der (A)PIC die Strategie



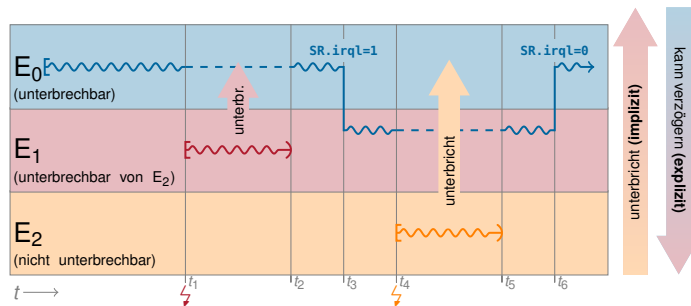
# Prioritätsebenenmodell

- Kontrollflüsse können die Ebene wechseln
  - Mit **cli** **wechselt** ein  $E_0$ -Kontrollfluss explizit auf  $E_1$ 
    - er ist ab dann nicht mehr unterbrechbar
    - andere  $E_1$ -Kontrollflüsse werden verzögert ( $\leftrightarrow$  Sequentialisierung)
  - Mit **sti** **wechselt** ein  $E_1$ -Kontrollfluss explizit auf  $E_0$ 
    - er ist ab dann (wieder) unterbrechbar
    - anhängige  $E_1$ -Kontrollflüsse „schlagen durch“ ( $\leftrightarrow$  Sequentialisierung)



## Prioritätsebenenmodell

- Verallgemeinerung für mehrere Unterbrechungsebenen:
  - Kontrollflüsse auf  $E_l$  werden
    - jederzeit **unterbrochen** durch Kontrollflüsse von  $E_m$  (für  $m > l$ )
    - nie unterbrochen** durch Kontrollflüsse von  $E_k$  (für  $k \leq l$ )
    - sequentialisiert** mit weiteren Kontrollflüssen von  $E_l$
  - Kontrollflüsse können die Ebene **wechseln**
    - durch spezielle Operationen (hier: Modifizieren des Statusregisters)



## Prioritätsebenenmodell: Konsistenzsicherung

- Jede Zustandsvariable ist (logisch) genau einer Ebene  $E_l$  zugeordnet
  - Zugriffe aus  $E_l$  sind implizit konsistent ( $\leftrightarrow$  Sequentialisierung)
  - Konsistenz bei Zugriff aus höheren / tieferen Ebenen muss explizit sichergestellt werden
- Maßnahmen zur Konsistenzsicherung bei Zugriffen:
  - „von oben“ (aus  $E_k$  mit  $k < l$ ) durch **harte Synchronisation**
    - explizit die Ebene auf  $E_l$  wechseln beim Zugriff (Verzögerung)
    - damit erfolgt der Zugriff aus derselben Ebene ( $\rightarrow$  Sequentialisierung)
  - „von unten“ (aus  $E_m$  mit  $m > l$ ) durch **weiche Synchronisation**
    - algorithmisch sicherstellen, dass Unterbrechungen nicht stören
    - erfordert unterbrechungstransparente Algorithmen

## Agenda

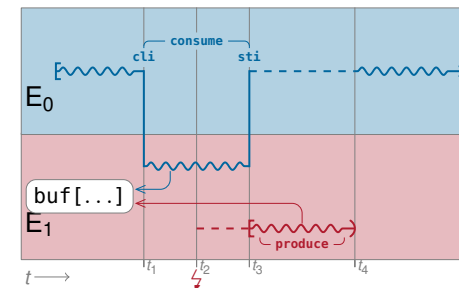
- Einleitung
- Prioritätsebenenmodell
- Harte Synchronisation
  - Ansatz
  - Bewertung
- Weiche Synchronisation
- Prolog/Epilog-Modell
- Zusammenfassung
- Referenzen

## Bounded Buffer – Lösung mit harter Synchronisation

Zugriff „von oben“ wird hart synchronisiert: Für die Ausführung von `consume()` wechselt der Kontrollfluss auf  $E_1$

```
char consume() {
    cli();
    ...
    char result = buf[nextout++];
    ...
    sti();
    return result;
}
```

```
void produce(char data) {
    // hier nichts zu tun
    ...
    buf[nextin++] = data;
    ...
    //hier nichts zu tun
}
```



## Harte Synchronisation: Bewertung

### ■ Vorteile

- Konsistenz ist sicher gestellt
  - auch bei komplexen Datenstrukturen und Zugriffsmustern
  - unabhängig davon, was der Compiler macht
- einfach anzuwenden, „funktioniert immer“
  - im Zweifelsfall legt man einfach sämtlichen Zustand auf die höchstpriorie Ebene

### ■ Nachteile

- Breitbandwirkung
  - Es werden pauschal alle Unterbrechungsbehandlungen (Kontrollflüsse) auf und unterhalb der Zustandsebene verzögert
- Prioritätsverletzung
  - Es werden Kontrollflüsse höherer Priorität verzögert
- prophylaktisches Verfahren
  - Nachteile werden in Kauf genommen, obwohl die Wahrscheinlichkeit, dass tatsächlich eine relevante Unterbrechung eintrifft, sehr klein ist.



## Harte Synchronisation: Bewertung (Forts.)

- Ob die Nachteile erheblich sind, hängt ab von
  - Häufigkeit,
  - durchschnittlicher Dauer,
  - maximaler Dauerder Verzögerung.
- Kritisch ist vor allem die **maximale Dauer**
  - hat direkten Einfluss auf die anzunehmende Latenz
  - Wird die Latenz zu hoch, können Daten verloren gehen
    - *edge-triggered* Unterbrechungen gehen verloren
    - Daten werden zu langsam von EA-Gerät abgeholt

### Fazit

Harte Synchronisation ist eher **ungeeignet** für die Konsistenzsicherung **komplexer Datenstrukturen**



## Agenda

- Einleitung
- Prioritätsebenenmodell
- Harte Synchronisation
- Weiche Synchronisation**
  - Ansatz
  - Implementierungsbeispiele
  - Bewertung
- Prolog/Epilog-Modell
- Zusammenfassung
- Referenzen

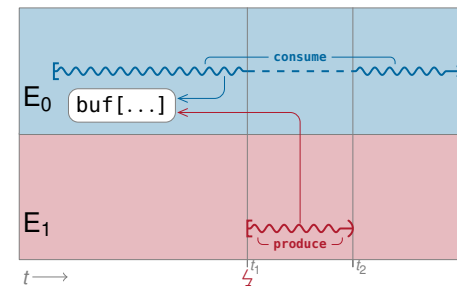


## Bounded Buffer – Ansatz mit weicher Synchronisation

Zugriff „von unten“ wird weich synchronisiert: consume() liefert ein korrektes Ergebnis, auch wenn während der Abarbeitung produce() ausgeführt wurde.

```
char consume() {  
    ?  
}
```

```
void produce(char data) {  
    ?  
}
```

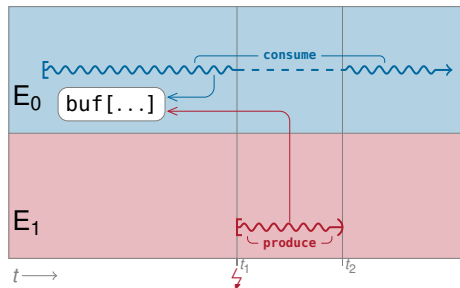


Zustand liegt (logisch) auf E<sub>0</sub>



## Bounded Buffer – Konsistenzbedingungen, Annahmen

- **Konsistenzbedingung**
  - Ergebnis einer unterbrochenen Ausführung soll äquivalent sein zu dem einer sequentiellen Ausführung der Operation
    - entweder consume() vor produce() oder consume() nach produce()
- **Annahmen**
  - produce() unterbricht consume()
    - alle anderen Kombinationen kommen nicht vor
  - produce() läuft immer durch (*run-to-completion*)



## Bounded Buffer – Implementierung aus der letzten VL

Kritisch ist der gemeinsam verwendete Zustand

```
// Pufferklasse in C++
class BoundedBuffer {
    char buf[SIZE]; int occupied; int nextin, nextout;
public:
    BoundedBuffer(): occupied(0), nextin(0), nextout(0) {}
    void produce(char data) { // Unterbrechungsbehandlung:
        int elements = occupied; // Elementzaehler merken
        if (elements == SIZE) return; // Element verloren
        buf[nextin] = data; // Element schreiben
        nextin++; nextin %= SIZE; // Zeiger weitersetzen
        occupied = elements + 1; // Zaehler erhoehen
    }
    char consume() { // normaler Kontrollfluss:
        int elements = occupied; // Elementzaehler merken
        if (elements == 0) return 0; // Puffer leer, kein Ergebnis
        char result = buf[nextout]; // Element lesen
        nextout++; nextout %= SIZE; // Lesezeiger weitersetzen
        occupied = elements - 1; // Zaehler erniedrigen
        return result; // Ergebnis zurueckliefern
    }
};
```



## Bounded Buffer – Implementierung aus der letzten VL

Kritisch ist der gemeinsam verwendete Zustand

```
// Pufferklasse in C++
class BoundedBuffer {
    char buf[SIZE]; int occupied; int nextin, nextout;
public:
    BoundedBuffer(): occupied(0), nextin(0), nextout(0) {}
    void produce(char data) { // Unterbrechungsbehandlung:
        int elements = occupied; // Elementzaehler merken
        if (elements == SIZE) return; // Element verloren
        buf[nextin] = data; // Element schreiben
        nextin++; nextin %= SIZE; // Zeiger weitersetzen
        occupied = elements + 1; // Zaehler erhoehen
    }
    char consume() { // normaler Kontrollfluss:
        int elements = occupied; // Elementzaehler merken
        if (elements == 0) return 0; // Puffer leer, kein Ergebnis
        char result = buf[nextout]; // Element lesen
        nextout++; nextout %= SIZE; // Lesezeiger weitersetzen
        occupied = elements - 1; // Zaehler erniedrigen
        return result; // Ergebnis zurueckliefern
    }
};
```

Insbesondere Zustand, auf den von beiden Seiten **schreibend** zugegriffen wird.



## Bounded Buffer – Alternative Implementierung

Diese alternative Implementierung kommt ohne gemeinsam beschriebenen Zustand aus.

```
// Pufferklasse in C++ (alternativ)
class BoundedBuffer {
    char buf[SIZE]; int nextin, nextout;
public:
    BoundedBuffer(): nextin(0), nextout(0) {}
    void produce(char data) {
        if ((nextin + 1) % SIZE == nextout) return;
        buf[nextin] = data;
        nextin = (nextin + 1) % SIZE;
    }
    char consume() {
        if (nextout == nextin) return 0;
        char result = buf[nextout];
        nextout = (nextout + 1) % SIZE;
        return result;
    }
};
```



## Bounded Buffer – Alternative Implementierung

```
// Pufferklasse in C++ (alternativ)
class BoundedBuffer {
    char buf[SIZE]; int nextin, nextout;
public:
    BoundedBuffer(): nextin(0), nextout(0) {}
    void produce(char data) {

        if ((nextin + 1) % SIZE == nextout) return;
        buf[nextin] = data;
        nextin = (nextin + 1) % SIZE;
    }
    char consume() {

        if (nextout == nextin) return 0;
        char result = buf[nextout];
        nextout = (nextout + 1) % SIZE;

        return result;
    }
};
```

Allerdings gibt es hier jetzt Zustand, der von einer Seite gelesen und von der jeweils anderen beschrieben wird.

An genau diesen Stellen müssen wir prüfen, ob die Konsistenzbedingung gilt.



## Bounded Buffer – Analyse der neuen Implementierung

### ■ Angenommen, die Unterbrechung von consume() erfolgt:

- aus der Sicht von consume()
  - vor dem Lesen von **nextin**  $\Leftrightarrow$  consume() nach produce() ✓
  - nach dem Lesen von **nextin**  $\Leftrightarrow$  consume() vor produce() ✓
- aus der Sicht von produce()
  - vor dem Schreiben von **nextout**  $\Leftrightarrow$  produce() vor consume() ✓
  - nach dem Schreiben von **nextout**  $\Leftrightarrow$  produce() nach consume() ✓

```
char consume() {
    if (nextout == nextin) return 0;
    char result = buf[nextout];
    nextout = (nextout + 1) % SIZE;
    return result;
}
```

Konsistenzbedingung ist in jedem Fall erfüllt!

```
void produce(char data) {
    if ((nextin + 1) % SIZE == nextout) return;
    buf[nextin] = data;
    nextin = (nextin + 1) % SIZE;
}
```



## Systemzeit – Implementierung aus der letzten Vorlesung

```
/* globale Zeitvariable */
extern volatile time_t global_time;
```

```
/* Systemzeit abfragen */
time_t time () {
    return global_time;
}
```

```
/* Unterbrechungs- *
 * * behandlung */
void timerHandler () {
    global_time++;
}
```

h8300-hms-g++ (16-Bit-Architektur)

```
time:
    mov global_time, %r0; lo
    mov global_time+2, %r1; hi
    ret
```

**Problem:**  
Daten werden nicht atomar gelesen.



## Systemzeit – Konsistenzbedingungen, Annahmen, Ansatz

### ■ Konsistenzbedingung

- Ergebnis einer unterbrochenen Ausführung soll äquivalent sein zu dem einer sequentiellen Ausführung der Operation
  - entweder time() vor timerHandler() oder umgekehrt

### ■ Annahmen

- timerHandler() unterbricht time()
  - alle anderen Kombinationen kommen nicht vor
- timerHandler() läuft immer durch (run-to-completion)

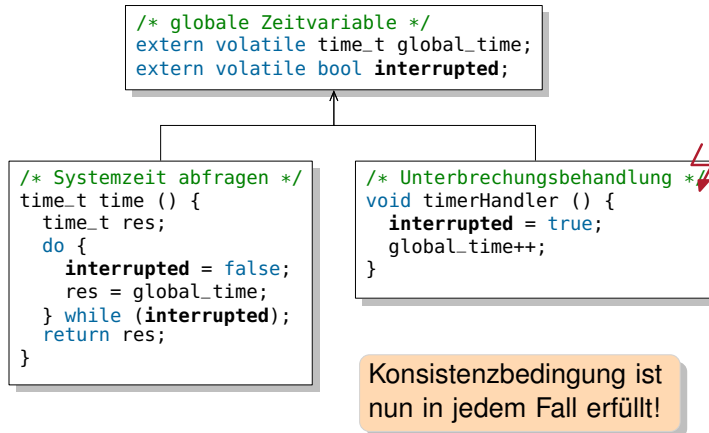
### ■ Lösungsansatz: In time() optimistisch herangehen

1. lese Daten unter der Annahme, nicht unterbrochen zu werden
2. überprüfe, ob Annahme zutraf – wurden wir unterbrochen?
3. falls unterbrochen, setze neu auf ab Schritt 1





## Systemzeit – Neue Implementierung



## Weiche Synchronisation: Bewertung

### ■ Vorteile

- Konsistenz ist sichergestellt (durch Unterbrechungstransparenz)
- Priorität wird nie verletzt
  - Kontrollflüsse der höherpriorien Ebenen kommen immer durch
- Kosten entstehen entweder gar nicht oder nur im Konfliktfall
  - gar nicht  $\rightsquigarrow$  Beispiel Bounded Buffer
  - im Konfliktfall  $\rightsquigarrow$  optimistische Verfahren, Beispiel Systemzeit (zusätzliche Kosten durch Wiederaufsetzen)

### ■ Nachteile

- Lösungen häufig sehr komplex
  - Wenn man überhaupt eine Lösung findet, ist diese in der Regel schwer zu verstehen – und noch schwieriger zu verifizieren
- Lösungen häufig sehr fragil (bezüglich Randbedingungen)
  - Kleinste Änderungen können die Konsistenzgarantie zerstören
  - Codegenerierung des Compilers ist zu beachten
- Bei größeren Datenmengen steigen die Wiederaufsetz-Kosten



## Weiche Synchronisation: Bewertung (Forts.)

### Fazit

- Weiche Synchronisation durch Unterbrechungstransparenz ist **grundsätzlich erstrebenswert!**
- Es handelt sich bei den Algorithmen jedoch immer um **Speziallösungen** für **Spezialfälle**.
- Als allgemein verwendbares Mittel für die Sicherung **beliebiger Datenstrukturen** ist sie **nicht geeignet**.



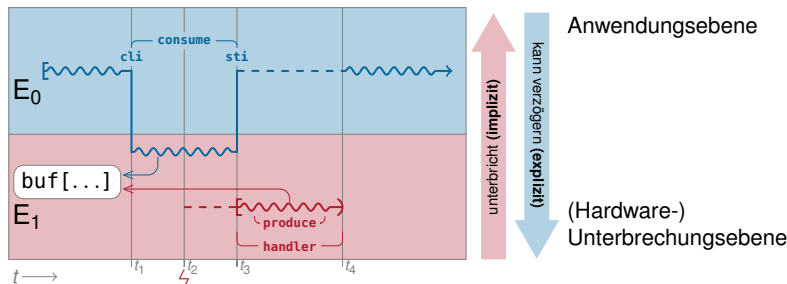
## Agenda

Einleitung  
Prioritätsebenenmodell  
Harte Synchronisation  
Weiche Synchronisation  
Prolog/Epilog-Modell  
  Ansatz  
  Implementierung  
  Bewertung  
  Verwandte Konzepte  
Zusammenfassung  
Referenzen



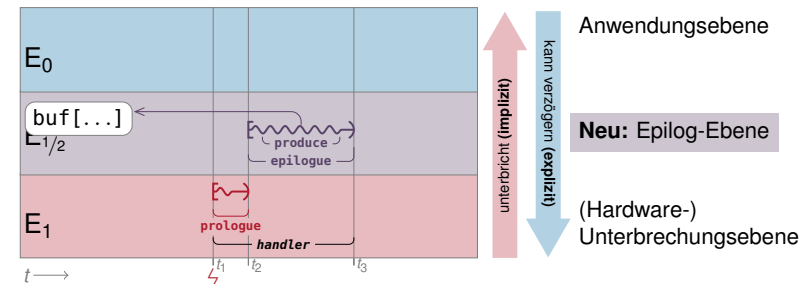
## Prolog/Epilog-Modell – Motivation

- **Reprise:** Harte Synchronisation
  - einfach, korrekt, „funktioniert immer“ ✓
  - Hauptproblem ist die hohe Latenz ✗
    - Verzögerung bei **Zugriff auf den Zustand** aus höheren Ebenen
    - Verzögerung bei **Bearbeitung des Zustands** in der UB selbst
  - letztlich dadurch verursacht, dass der Zustand (logisch) auf der/einer Hardwareunterbrechungsebene  $E_{1...n}$  liegt.



## Prolog/Epilog-Modell – Ansatz

- **Ansatz:** Latenzverbergung durch zusätzliche Ebene
  - Wir fügen eine weitere **logische Ebene** ein:  $E_{1/2}$ 
    - $E_{1/2}$  liegt zwischen der Anwendungsebene  $E_0$  und den UB-Ebenen  $E_{1...n}$
  - Unterbrechungsbehandlung wird **zweigeteilt** in **Prolog** und **Epilog**
    - **Prolog** arbeitet auf Unterbrechungsebene  $E_{1...n}$
    - **Epilog** arbeitet auf der neuen (Software-)Ebene  $E_{1/2}$  (**Epiloge**)
  - Zustand liegt (so weit wie möglich) auf der Epilogebeene
    - eigentliche Unterbrechungsbehandlung wird nur noch kurz gesperrt



## Prolog/Epilog-Modell – Ansatz (Forts.)

- Unterbrechungsbehandlungsroutinen werden zweigeteilt
  - beginnen im **Prolog** (immer)
  - werden fortgesetzt im **Epilog** (bei Bedarf)
- **Prolog** ( $\leadsto$  Hardwareunterbrechung)
  - läuft auf Hardwareunterbrechungsebene
    - hat damit Priorität über Anwendungsebene und Epilogebeene
  - ist **kurz**, fast wenig oder gar keinen Zustand an
    - Üblicherweise wird nur der Hardware-Zustand gesichert und bestätigt
    - Unterbrechungen bleiben nur kurz gesperrt ( $\leadsto$  Latenzminimierung)
    - kann bei Bedarf einen Epilog für die weitere Verarbeitung anfordern
- **Epilog** ( $\leadsto$  Softwareunterbrechung)
  - läuft auf Epilogebeene  $E_{1/2}$  (zusätzliche Kontrollflussebene)
    - Ausführung erfolgt verzögert zum Prolog
    - erledigt die eigentliche Arbeit ( $\leadsto$  Latenzverbergung)
  - hat Zugriff auf größten Teil des Zustands
    - Zustand wird auf Epilogebeene synchronisiert

## Prolog/Epilog-Modell – Epilogebeene

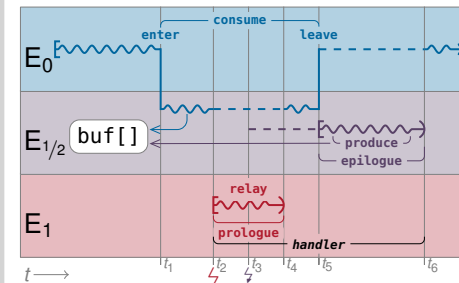
- Die Epilogebeene wird (ganz oder teilweise) in **Software** implementiert
  - trotzdem handelt es sich um eine ganz normale Prioritätsebene des Ebenenmodells
  - es müssen daher auch dieselben Gesetzmäßigkeiten gelten
- Es gilt: Kontrollflüsse auf der Epilogebeene  $E_{1/2}$  werden
  1. **jederzeit unterbrochen** durch Kontrollflüsse der Ebenen  $E_{1...n}$ 
    - $\leadsto$  Prologe (Unterbrechungen) haben Priorität über Epiloge
  2. **nie unterbrochen** durch Kontrollflüsse der Ebene  $E_0$ 
    - $\leadsto$  Epiloge haben Priorität über Anwendungskontrollflüsse
  3. **sequenzialisiert** mit anderen Kontrollflüssen von  $E_{1/2}$ 
    - $\leadsto$  Anhängige Epiloge werden nacheinander abgearbeitet.
    - $\leadsto$  Bei Rückkehr zur Anwendungsebene sind alle Epiloge abgearbeitet.

## Prolog/Epilog-Modell – Implementierung

- Benötigt werden Operationen, um
  1. explizit die Epilogebeine zu betreten: **enter()**
    - entspricht dem `cli` bei der harten Synchronisation
  2. explizit die Epilogebeine zu verlassen: **leave()**
    - entspricht dem `sti` bei der harten Synchronisation
  3. einen Epilog anzufordern: **relay()**
    - entspricht dem Hochziehen der IRQ-Leitung beim PIC



## Prolog/Epilog-Modell – Ablaufbeispiel



E<sub>1</sub>-Unterbrechungen werden nie gesperrt.

Aktivierungslatenz der Unterbrechungsbehandlung ist minimal.

- t<sub>1</sub> Anwendungskontrollfluss betritt Epilogebeine E<sub>1/2</sub> (`enter()`).
- t<sub>2</sub> Unterbrechung ⚡ auf Ebene E<sub>1</sub> wird signalisiert → Prolog wird ausgeführt.
- t<sub>3</sub> Prolog fordert Epilog für die nachgeordnete Bearbeitung an (`relay()` ⚡).
- t<sub>4</sub> Prolog terminiert, unterbrochener E<sub>1/2</sub>-Kontrollfluss läuft weiter.
- t<sub>5</sub> Anwendungskontrollfluss verlässt die Epilogebeine E<sub>1/2</sub> (`leave()`)  
→ zwischenzeitlich aufgelaufene Epiloge werden nun abgearbeitet.
- t<sub>6</sub> Epilog terminiert, Anwendungskontrollfluss fährt auf E<sub>0</sub> fort.



## Prolog/Epilog-Modell – Implementierung

- Benötigt werden Operationen, um
  1. explizit die Epilogebeine zu betreten: **enter()**
    - entspricht dem `cli` bei der harten Synchronisation
  2. explizit die Epilogebeine zu verlassen: **leave()**
    - entspricht dem `sti` bei der harten Synchronisation
  3. einen Epilog anzufordern: **relay()**
    - entspricht dem Hochziehen der IRQ-Leitung beim PIC
- Außerdem Mechanismen, um
  4. anhängige Epiloge zu „merken“: **queue** (z. B.)
    - entspricht dem IRR (Interrupt-Request-Register) beim PIC
  5. sicherzustellen, dass anhängige Epiloge abgearbeitet werden
    - entspricht bei der harten Synchronisation dem Protokoll zwischen CPU und PIC

Dieser Punkt muss etwas genauer betrachtet werden!



## Prolog/Epilog-Modell – Implementierung

5. sicherzustellen, dass anhängige Epiloge abgearbeitet werden
  - entspricht bei der harten Synchronisation dem Protokoll zwischen CPU und PIC

Wann müssen anhängige Epiloge abgearbeitet werden?

Immer unmittelbar, bevor die CPU auf E<sub>0</sub> zurückkehrt!

1. bei explizitem Verlassen der Epilogebeine mit `leave()`
  - während der Anwendungskontrollfluss auf E<sub>1/2</sub> gearbeitet hat könnten Epiloge aufgelaufen sein (↔ Sequentialisierung).
2. nach Abarbeitung des letzten Epilogs
  - während der Epilogabarbeitung könnten weitere Epiloge aufgelaufen sein (↔ Sequentialisierung).
3. wenn der **letzte** Unterbrechungsbehandler terminiert
  - während der Abarbeitung von E<sub>1...n</sub>-Kontrollflüssen könnten Epiloge aufgelaufen sein (↔ Priorisierung).



## Prolog/Epilog-Modell – Implementierung

- Implementierungsvarianten
  - rein softwarebasiert (↔ Übung)
  - mit Hardwareunterstützung durch einen AST (↔ [2, 3])
- Ein AST (*asynchronous system trap*) ist eine Unterbrechung, die (nur) durch Software angefordert werden kann.
  - z. B. durch Setzen eines Bits in einem bestimmten Register
  - ansonsten technisch vergleichbar mit einer Hardware-Unterbrechung
    - AST wird (im Gegensatz zu Traps/Exceptions) *asynchron* abgearbeitet
    - AST läuft auf eigener Unterbrechungsebene zwischen der Anwendungsebene und den Hardware-UBs (↔ unsere  $E_{1/2}$ )
    - Gesetzmäßigkeiten des Ebenenmodells gelten (AST-Ausführung ist verzögerbar, wird automatisch aktiviert, ...)
- Sicherstellung der Epilogbearbeitung wird damit sehr einfach!
  - Abarbeitung der Epiloge erfolgt im AST
    - ↔ und damit automatisch, bevor die CPU auf  $E_0$  zurückkehrt
  - bleibt nur noch die Verwaltung der anhängigen Epiloge



## Prolog/Epilog-Modell – Implementierung

- Beispiel TriCore: Implementierung mit AST
  - AST hier als Unterbrechung der  $E_1$  konfiguriert (↔ unsere  $E_{1/2}$ )
  - Geräteunterbrechungen laufen auf  $E_{2...n}$

```
void enter() {
    CPU::setIRQL(1);           // betrete E1, verzögere AST
}
void leave() {
    CPU::setIRQL(0);          // erlaube AST (anhaengiger
                              // AST wurde jetzt abgearbeitet)
}
void relay(<Epilog>) {
    <haenge Epilog an queue an>
    CPU_SRC1::trigger();      // aktiviere Level-1 IRQ (AST)
}
void __attribute__((interrupt_handler)) irq1Handler() {
    while(<Epilog in queue>) {
        <entferne Epilog aus queue>
        <arbeite Epilog ab>
    }
}
```



## Prolog/Epilog-Modell – Implementierung

- Beispiel TriCore: Implementierung mit AST
  - AST hier als Unterbrechung der  $E_1$  konfiguriert (↔ unsere  $E_{1/2}$ )
  - Geräteunterbrechungen laufen auf  $E_{2...n}$

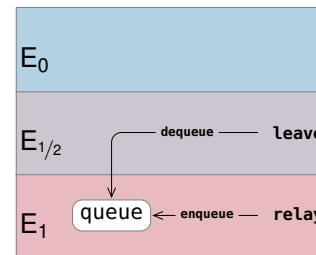
```
void enter() {
    CPU::setIRQL(1);           // b
}
void leave() {
    CPU::setIRQL(0);          // e
}
void relay(<Epilog>) {
    <haenge Epilog an queue an>
    CPU_SRC1::trigger();      // aktiviere Level-1 IRQ (AST)
}
void __attribute__((interrupt_handler)) irq1Handler() {
    while(<Epilog in queue>) {
        <entferne Epilog aus queue>
        <arbeite Epilog ab>
    }
}
```

Bietet die Hardware (wie z. B. IA-32) kein AST-Konzept, so kann man dieses in Software nachbilden.  
Näheres dazu in der Übung.



## Prolog/Epilog-Modell – Ziel erreicht?

- Kern-Zustand kann jetzt auf Epilog-Ebene verwaltet und synchronisiert werden.
  - Hardware-UBs müssen nicht (mehr) gesperrt werden!
- Ein Problem bleibt noch: Die Epilog-Warteschlange
  - Zugriff erfolgt aus Prologon und der Epilog-Ebene
    - muss also entweder hart synchronisiert werden (im Bild)
    - oder man sucht eine Speziallösung mit weicher Synchronisation



Harte Synchronisation erscheint hier *akzeptabel*, da die Sperrzeit (↔ Ausführungszeit von `dequeue()`) *kurz* und *deterministisch* ist.

Eine Lösung mit *weicher Synchronisation* (z. B. [6]) wäre natürlich schöner!



## Prolog/Epilog-Modell: Bewertung

### ■ Vorteile

- Konsistenz ist sichergestellt (durch Synchronisation auf Epilog-Ebene)
- Programmiermodell entspricht dem (einfach verständlichen) Modell der harten Synchronisation
- Auch komplexer Zustand kann synchronisiert werden
  - ohne das dabei Unterbrechungsanforderungen verloren gehen
  - ermöglicht es, den gesamte BS-Kern auf Epilog-Ebene zu schützen

### ■ Nachteile

- Zusätzliche Ebene führt zu zusätzlichem Overhead
  - Epilogaktivierung könnte länger dauern als direkte Behandlung
  - Komplexität für den BS-Entwickler wird erhöht
- Unterbrechungssperren lassen sich nicht vollständig vermeiden
  - Gemeinsamer Zustand von Pro- und Epilog muss weiter hart oder weich synchronisiert werden



## Prolog/Epilog-Modell: Bewertung (Forts.)

### Fazit

- Das Prolog/Epilog-Modell ist ein **guter Kompromiss** für die Synchronisation des Kernzustands.
- Es ist auch für die Konsistenzsicherung **komplexer Datenstrukturen geeignet**



## Prolog/Epilog-Modell – Verwandte Konzepte

- UNIX: top/bottom half [4]
  - Aktivitäten der bottom half ( $\rightarrow E_1$ ) sind asynchron zu den Aktivitäten der top half ( $\rightarrow E_{1/2}$ ) und dürfen keine Systemfunktionen aufrufen
- Windows: ISRs / deferred procedure calls (DPCs) [7]
  - Unterbrechungsbehandler ( $\rightarrow$  Prologe) können DPCs ( $\rightarrow$  Epiloge) in eine Warteschlange einhängen. Diese wird verzögert abgearbeitet, bevor die CPU auf Faden-Ebene zurückkehrt
- Linux: top halbes / bottom halbes, tasklets [1, 5]
  - Zu jedem Unterbrechungsbehandler (ISR) gibt es ein Bit in einer Bitmaske, durch das eine verzögerte bottom half (BH  $\rightarrow$  Epilog) angefordert werden kann. Die BHs werden vor dem Verlassen des Kerns ausgeführt.
  - Neuere Linux-Kerne verwenden tasklets, vgl. mit Windows DPCs
- eCos: ISRs / deferred service routines (DSRs)

■ ...

Nahezu alle Betriebssysteme, die Unterbrechungsbehandlung verwenden, bieten auch eine „Epilog-Ebene“.



## Agenda

Einleitung  
Prioritätsebenenmodell  
Harte Synchronisation  
Weiche Synchronisation  
Prolog/Epilog-Modell  
**Zusammenfassung**  
Referenzen



## Zusammenfassung: Unterbrechungssynchronisation

- Konsistenzsicherung im BS-Kern
  - muss anders erfolgen als zwischen Prozessen – einseitig
  - Kontrollflüsse arbeiten auf verschiedenen Prioritätsebenen
- Maßnahmen zur Konsistenzsicherung
  - harte Synchronisation (durch Unterbrechungssperren)
    - einfach, jedoch negative Auswirkungen auf Latenz
    - Unterbrechungsanforderungen können verloren gehen
  - weiche Synchronisation (durch Unterbrechungstransparenz)
    - gut und effizient, jedoch nur in Spezialfällen möglich
    - Implementierung kann sehr komplex werden
  - Prolog/Epilog-basierte Synchronisation (Zweiteilung der Unterbrechungsbehandlung)
    - guter Kompromiss
    - Stand der Technik in heutigen Betriebssystemen



## Referenzen

- [1] Daniel P. Bovet and Marco Cesati. *Understanding the Linux Kernel*. O'Reilly, 2001. ISBN: 0-596-00002-2.
- [2] Digital Equipment Corporation. *VAX-11 Architecture Reference Manual*. Document Number EK-VAXAR-RM-001. Digital Equipment Corporation. Maynard, MA, USA: Digital Press, May 1982.
- [3] Samuel J. Leffler, Marshall Kirk McKusick, Michael J. Karels, et al. *The Design and Implementation of the 4.3 BSD UNIX Operating System*. Addison-Wesley, May 1989. ISBN: 0-201-06196-1.
- [4] John Lions. *Lions' Commentary on UNIX (6th Edition)*. Peer-to-Peer Communications Inc., 1977. ISBN: 978-1573980135.
- [5] Robert Love. *Linux Kernel Development (2nd Edition)*. Novell Press, 2005. ISBN: 978-0672327209.
- [6] Friedrich Schön, Wolfgang Schröder-Preikschat, Olaf Spinczyk, et al. "On Interrupt-Transparent Synchronization in an Embedded Object-Oriented Operating System". In: *Proceedings of the 3rd IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC '00)*. (Newport Beach, CA, USA). IEEE Computer Society Press, Mar. 2000, pp. 270–277. DOI: 10.1109/ISORC.2000.839540.



## Nachtrag: Mehrkernsysteme

### Beachte: Unterbrechungsbehandlung $\neq$ Parallelität

- Techniken funktionieren (so) nur bei echter Unterbrechungssensitivität: A und UB werden auf **demselben** Prozessor ausgeführt
- Wird die UB „echt parallel“ (auf einem weiteren Prozessor) ausgeführt, kommt es zu Problemen
  - Annahmen des Prioritätsebenenmodells gelten nicht mehr! (Sequenzialisierung, Priorisierung, *run-to-completion*)
  - Asymmetrie (UB unterbricht A) ist nicht länger gegeben (weiche Synchronisation wird dadurch viel schwieriger)
- Zusätzlich erforderlich: **Inter-Prozessor-Synchronisation**
  - „hart“  $\rightarrow$  zweiseitig blockierend, z. B. mit *Spin-Locks*  $\rightsquigarrow$  Übung
  - „weich“  $\rightarrow$  algorithmisch nichtblockierend (**schwer!**)



## Referenzen (Forts.)

- [SP] Wolfgang Schröder-Preikschat. *Systemprogrammierung*. Vorlesung mit Übung. Friedrich-Alexander-Universität Erlangen-Nürnberg, Lehrstuhl für Informatik 4, 2009 (jährlich). URL: [http://www4.informatik.uni-erlangen.de/Lehre/WS09/V\\_SP](http://www4.informatik.uni-erlangen.de/Lehre/WS09/V_SP).
- [7] David A. Solomon and Mark Russinovich. *Inside Microsoft Windows 2000 (3rd Edition)*. Microsoft Press, 2000. ISBN: 3-86063-630-8.

