

# Betriebssysteme (BS)

## VL 5 – Unterbrechungen, Synchronisation

Volkmar Sieh / Daniel Lohmann

Lehrstuhl für Informatik 4  
Verteilte Systeme und Betriebssysteme

Friedrich-Alexander-Universität  
Erlangen Nürnberg

WS 18 – 22. November 2018



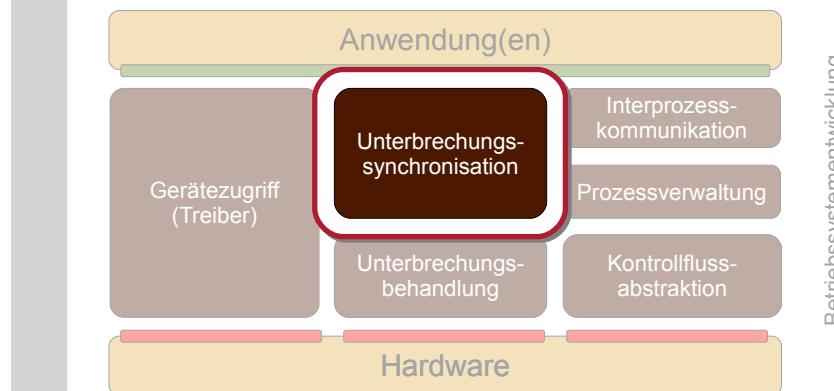
[https://www4.cs.fau.de/Lehre/WS18/V\\_BS](https://www4.cs.fau.de/Lehre/WS18/V_BS)

### Agenda

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



### Überblick: Einordnung dieser VL



Betriebssystementwicklung

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

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

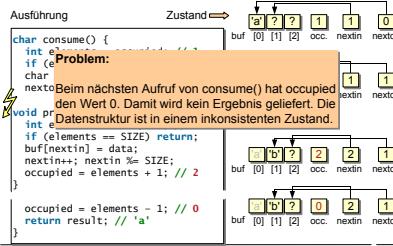


divs Betriebssysteme (VL 4 | WS 17) 4 Unterbrechungen, Software - Zustandsänderung

## Beispiele aus der letzten Vorlesung

### Beispiel 2: Ringpuffer

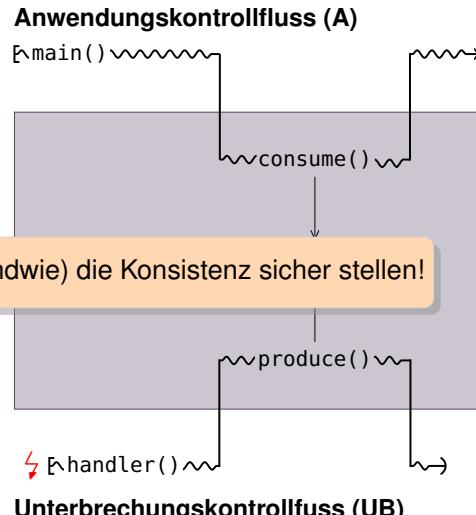
auch die Pufferimplementierung ist kritisch ...



divs Betriebssysteme (VL 4 | WS 17) 4 Unterbrechungen, Software - Zustandsänderung 4-38

## Motivation: Ursache

### Kontrollflüsse "von oben"



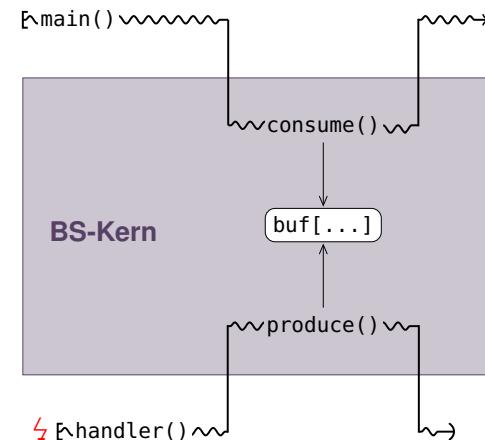
und "von unten"

vs/dl Betriebssysteme (VL 5 | WS 18) 5 Unterbrechungen, Synchronisation - Einleitung

## Motivation: Ursache

### Kontrollflüsse "von oben"

### Anwendungskontrollfluss (A)



"begegnen" sich im Kern

und "von unten"

### Unterbrechungskontrollfuss (UB)

vs/dl Betriebssysteme (VL 5 | WS 18) 5 Unterbrechungen, Synchronisation - Einleitung

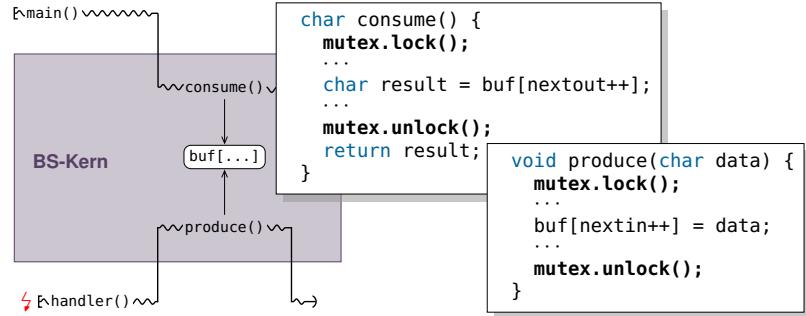
5-6

## Naiver Lösungsansatz

### Zweiseitige Synchronisation

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

### Anwendungskontrollfluss (A)



vs/dl Betriebssysteme (VL 5 | WS 18) 5 Unterbrechungen, Synchronisation - Einleitung

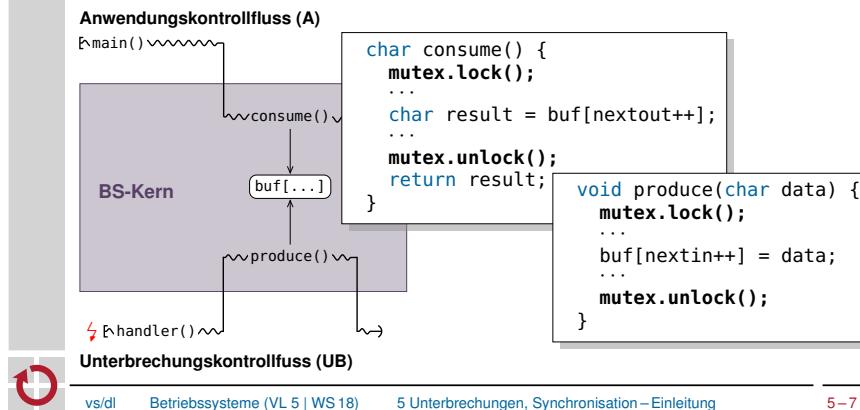
5-7

## Naiver Lösungsansatz

### Zweiseitige Synchronisation

- gegenseitiger Ausschluss durch Mutex
- wie zwischen zwei Prozessen

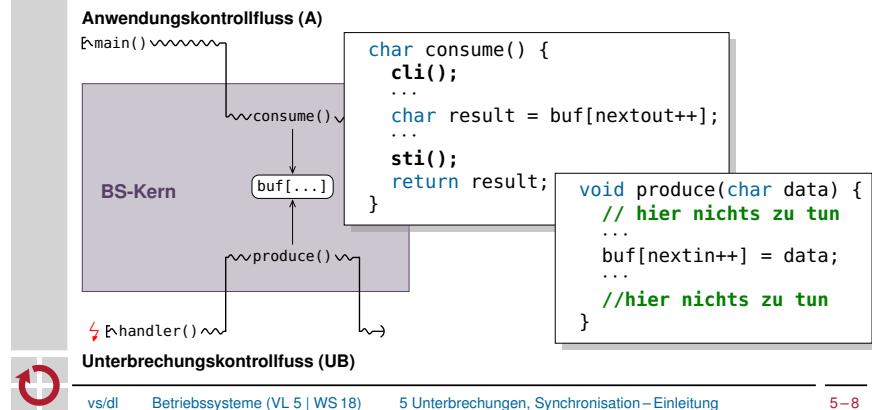
Zweiseitige Synchronisation funktioniert **natürlich nicht!**



## 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()`)

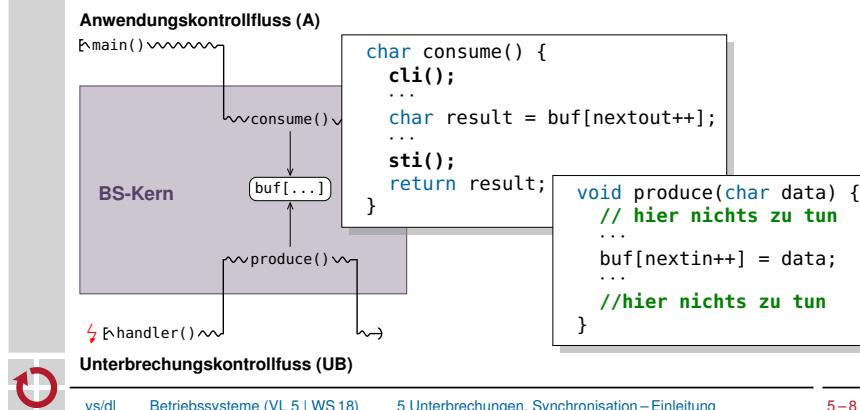


## Besserer Lösungsansatz

### Einseitige Synchronisation

- Unterdrückung der Unterbrechung
- Operationen `disable_interrupts()` funktioniert. [Warum?]

Einseitige Synchronisation funktioniert [Warum?]



## 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



vs/dl

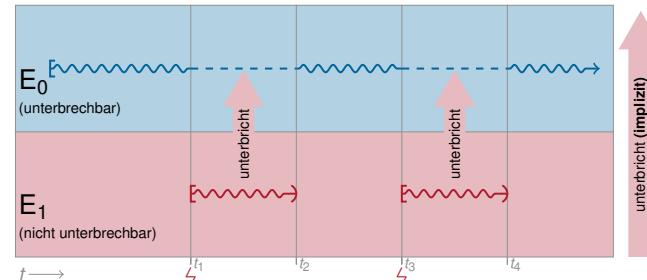
Betriebssysteme (VL 5 | WS 18)

5 Unterbrechungen, Synchronisation – Prioritätsebenenmodell

5–10

## 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)



vs/dl

Betriebssysteme (VL 5 | WS 18)

5 Unterbrechungen, Synchronisation – Prioritätsebenenmodell

5–11

## 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

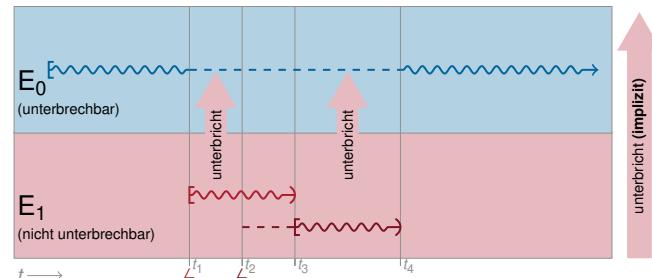


vs/dl

Betriebssysteme (VL 5 | WS 18)

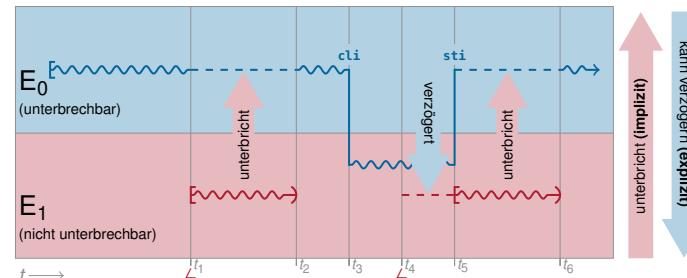
5 Unterbrechungen, Synchronisation – Prioritätsebenenmodell

5–12



## 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
  - Mit **sti** wechselt ein  $E_1$ -Kontrollfluss explizit auf  $E_0$ 
    - er ist ab dann (wieder) unterbrechbar
    - anhängige  $E_1$ -Kontrollflüsse „schlagen durch“



vs/dl

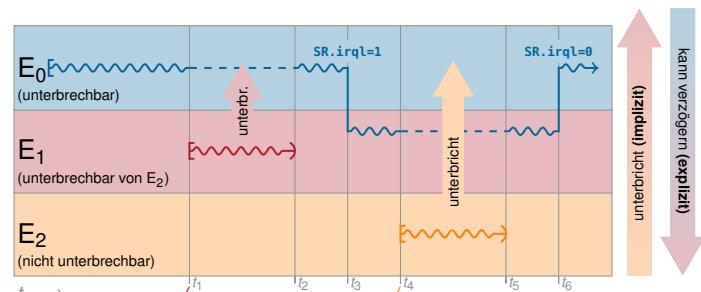
Betriebssysteme (VL 5 | WS 18)

5 Unterbrechungen, Synchronisation – Prioritätsebenenmodell

5–13

## Prioritätsebenenmodell

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



## Agenda

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



## Prioritätsebenenmodell: Konsistenzsicherung

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

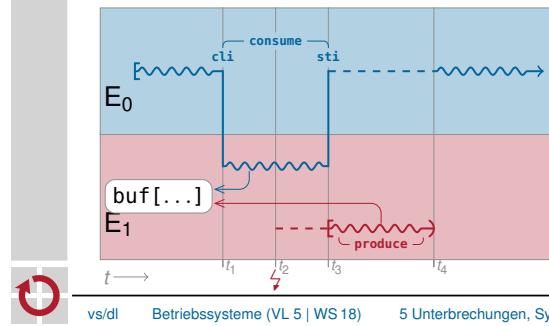


## Bounded Buffer – Lösung mit harter Synchronisation

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
}
```



Zustand liegt (logisch) auf  $E_1$



## 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 höchstpriore 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.



vs/dl

Betriebssysteme (VL 5 | WS 18)

5 Unterbrechungen, Synchronisation – Harte Synchronisation

5–18

## Agenda

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



vs/dl

Betriebssysteme (VL 5 | WS 18)

5 Unterbrechungen, Synchronisation – Weiche Synchronisation

5–20

## Harte Synchronisation: Bewertung (Forts.)

### Ob die Nachteile erheblich sind, hängt ab von

- Häufigkeit,
  - durchschnittlicher Dauer,
  - maximaler Dauer
- der Verzögerung.

### Kritisch ist vor allem die **maximale Dauer**

- hat direkten Einfluss auf die annehmende 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**



vs/dl

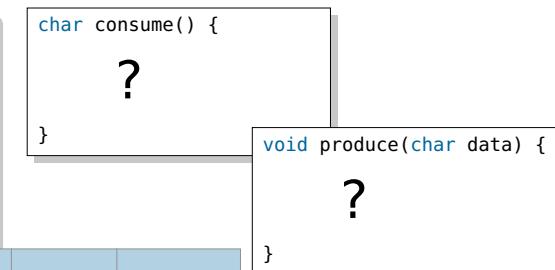
Betriebssysteme (VL 5 | WS 18)

5 Unterbrechungen, Synchronisation – Harte Synchronisation

5–19

## 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.



Zustand liegt (logisch) auf  $E_0$



vs/dl

Betriebssysteme (VL 5 | WS 18)

5 Unterbrechungen, Synchronisation – Weiche Synchronisation

5–21

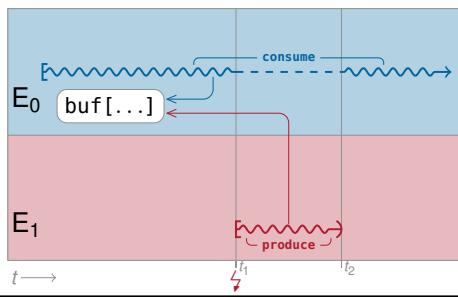
## Bounded Buffer – Konsistenzbedingungen, Annahmen

### 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 erhöhen
    }
    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 – 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 erhöhen
    }
    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 – 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;
    }
};
```

Diese alternative Implementierung kommt ohne gemeinsam beschriebenen Zustand aus.



## 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**
- nach dem Lesen von **nextin**

- ⇒ consume() nach produce()
- ⇒ consume() vor produce()

- aus der Sicht von produce()

- vor dem Schreiben von **nextout**
- nach dem Schreiben von **nextout**

- ⇒ produce() vor consume()
- ⇒ produce() nach consume()

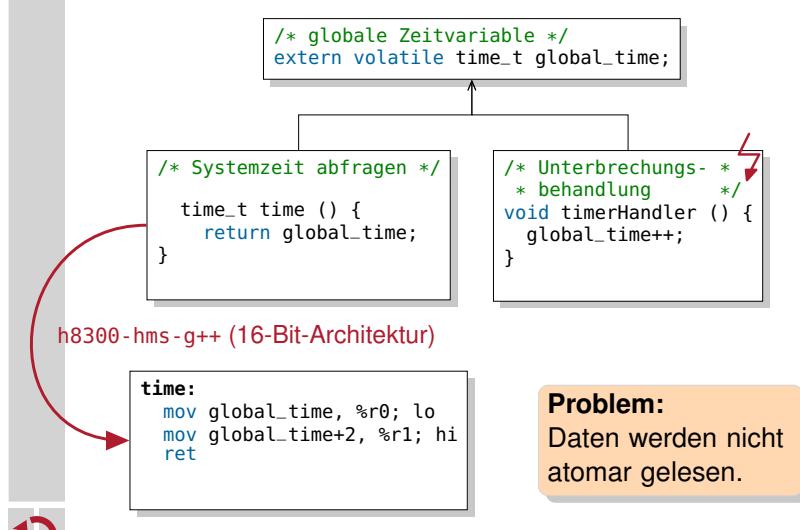
```
char consume() {
    if (nextout == nextin) return 0;
    char result = buf[nextout];
    nextout = (nextout + 1) % SIZE;
    return result;
}
```

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



## 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

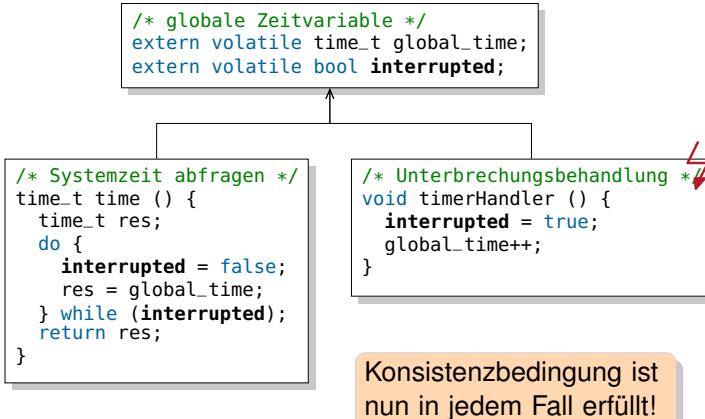
### Annahmen

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

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

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

## Systemzeit – Neue Implementierung



## 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.**



## Weiche Synchronisation: Bewertung

### Vorteile

- Konsistenz ist sichergestellt (durch Unterbrechungstransparenz)
- Priorität wird nie verletzt
  - Kontrollflüsse der höheren Ebenen kommen immer durch
- Kosten entstehen entweder gar nicht oder nur im Konfliktfall
  - gar nicht ↗ Beispiel Bounded Buffer
  - im Konfliktfall ↗ 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)
  - Kleinsten Änderungen können die Konsistenzgarantie zerstören
  - Codegenerierung des Compilers ist zu beachten
- Bei größeren Datenmengen steigen die Wiederaufsetzkosten



## Agenda

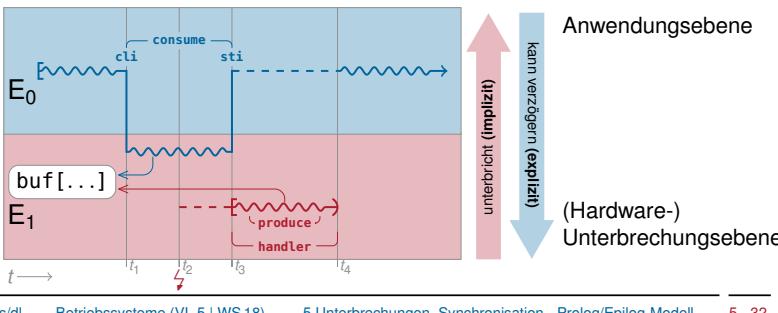
Einleitung  
Prioritätsebenenmodell  
Harte Synchronisation  
Weiche Synchronisation  
Prolog/Epilog-Modell  
Ansatz  
Implementierung  
Bewertung  
Verwandte Konzepte  
Zusammenfassung  
Referenzen



## Prolog/Epilog-Modell – Motivation

### ■ Reprise: Harte Synchronisation

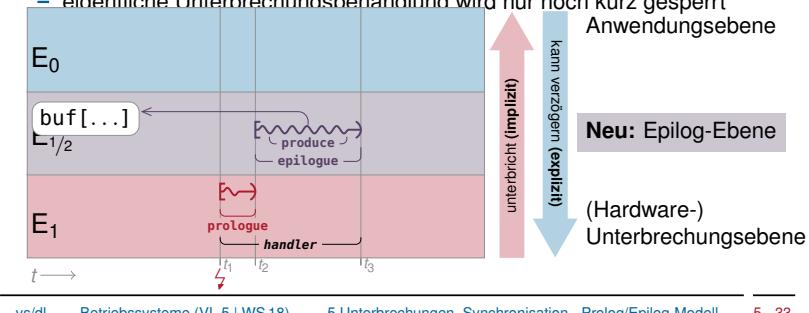
- einfach, korrekt, „funktioniert immer“ ✓
- Hauptproblem ist die hohe Latenz ✗
  - Verzögerung bei **Zugriff auf den Zustand** aus höheren Ebenen
  - Verzögerung bei **Bearbeitung des Zustands** in der UB selbst
- letztlich dadurch verursacht, dass der Zustand (logisch) auf der/einer Hardwareunterbrechungsebene  $E_{1\dots n}$  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  $E_{1\dots n}$
- Unterbrechungsbehandlung wird **zweigeteilt** in **Prolog** und **Epilog**
  - Prolog arbeitet auf Unterbrechungsebene  $E_{1\dots n}$
  - Epilog arbeitet auf der neuen (Software-)Ebene  $E_{1/2}$  (Epilogebene)
- Zustand liegt (so weit wie möglich) auf der Epilogebene
  - eigentliche Unterbrechungsbehandlung wird nur noch kurz gesperrt



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

### ■ Unterbrechungsbehandlungs Routinen werden zweigeteilt

- beginnen im **Prolog** (immer)
- werden fortgesetzt im **Epilog** (bei Bedarf)

### ■ Prolog (~ Hardwareunterbrechung)

- läuft auf Hardwareunterbrechungsebene
  - hat damit Priorität über Anwendungsebene und Epilogebene
- ist **kurz**, fasst wenig oder gar keinen Zustand an
  - Üblicherweise wird nur der Hardware-Zustand gesichert und bestätigt
  - Unterbrechungen bleiben nur kurz gesperrt (~ Latenzminimierung)
  - kann bei Bedarf einen Epilog für die weitere Verarbeitung anfordern

### ■ Epilog (~ Softwareunterbrechung)

- läuft auf Epilogebene  $E_{1/2}$  (zusätzliche Kontrollflussebene)
  - Ausführung erfolgt verzögert zum Prolog
  - erledigt die eigentliche Arbeit (~ Latenzverbergung)
- hat Zugriff auf größten Teil des Zustands
  - Zustand wird auf Epilogebene synchronisiert



## Prolog/Epilog-Modell – Epilogebene

### ■ Die Epilogebene 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 Epilogebene $E_{1/2}$ werden

1. **jederzeit unterbrochen** durch Kontrollflüsse der Ebenen  $E_{1\dots n}$ 
  - ~ Prolog (Unterbrechungen) haben Priorität über Epilog
2. **nie unterbrochen** durch Kontrollflüsse der Ebene  $E_0$ 
  - ~ Epilog haben Priorität über Anwendungskontrollflüsse
3. **sequentialisiert** mit anderen Kontrollflüssen von  $E_{1/2}$ 
  - ~ Anhängige Epiloge werden nacheinander abgearbeitet.
  - ~ Bei Rückkehr zur Anwendungsebene sind alle Epiloge abgearbeitet.



## Prolog/Epilog-Modell – Implementierung

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



vs/dl

Betriebssysteme (VL 5 | WS 18)

5 Unterbrechungen, Synchronisation – Prolog/Epilog-Modell

5-36

## Prolog/Epilog-Modell – Implementierung

- Benötigt werden Operationen, um
  1. explizit die Epiloge Ebene zu betreten: **enter()**
    - entspricht dem cli bei der harten Synchronisation
  2. explizit die Epiloge Ebene 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



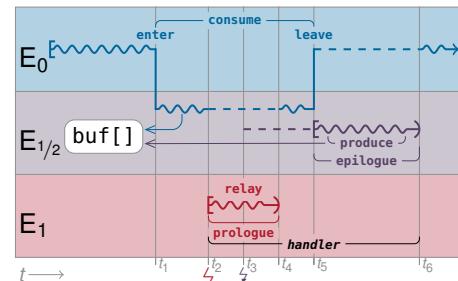
vs/dl

Betriebssysteme (VL 5 | WS 18)

5 Unterbrechungen, Synchronisation – Prolog/Epilog-Modell

5-38

## Prolog/Epilog-Modell – Ablaufbeispiel



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

Aktivierungslatenz der Unterbrechungsbehandlung ist minimal.

- t<sub>1</sub> Anwendungskontrollfluss betritt Epiloge Ebene 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 Epiloge Ebene 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.



vs/dl

Betriebssysteme (VL 5 | WS 18)

5 Unterbrechungen, Synchronisation – Prolog/Epilog-Modell

5-37

## 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 Epiloge Ebene 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).



vs/dl

Betriebssysteme (VL 5 | WS 18)

5 Unterbrechungen, Synchronisation – Prolog/Epilog-Modell

5-39

## Prolog/Epilog-Modell – Implementierung

- Implementierungsvarianten
  - rein softwarebasiert (☞ Übung) (☞ [2, 3])
  - mit Hardwareunterstützung durch einen AST
- 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<sub>1/2</sub>)
    - Gesetzmäßigkeiten des Ebenenmodells gelten  
(AST-Ausführung ist verzögerbar, wird automatisch aktiviert, ...)
- Sicherstellung der Epilogabarbeitung wird damit sehr einfach!
  - Abarbeitung der Epiloge erfolgt im AST
    - ☞ und damit automatisch, bevor die CPU auf E<sub>0</sub> zurückkehrt
  - bleibt nur noch die Verwaltung der anhängigen Epiloge



## Prolog/Epilog-Modell – Implementierung

- Beispiel TriCore: Implementierung mit AST
  - AST hier als Unterbrechung der E<sub>1</sub> konfiguriert (☞ unsere E<sub>1/2</sub>)
  - Geräteunterbrechungen laufen auf E<sub>2...n</sub>

```
void enter() {          // b
    CPU::setIRQL(1);
}
void leave() {           // e
    CPU::setIRQL(0);
}
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 – Implementierung

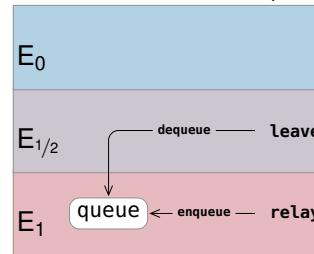
- Beispiel TriCore: Implementierung mit AST
  - AST hier als Unterbrechung der E<sub>1</sub> konfiguriert (☞ unsere E<sub>1/2</sub>)
  - Geräteunterbrechungen laufen auf E<sub>2...n</sub>

```
void enter() {
    CPU::setIRQL(1);           // betrete E1, verzoegere 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 – Ziel erreicht?

- Kernzustand kann jetzt auf Epilogebene verwaltet und synchronisiert werden.
  - Hardware-UBs müssen nicht (mehr) gesperrt werden!
- Ein Problem bleibt noch: Die Epilog-Warteschlange
  - Zugriff erfolgt aus Prologen und der Epilogebene
    - 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. [7]) wäre natürlich schöner!



### Vorteile

- Konsistenz ist sichergestellt  
(durch Synchronisation auf Epiloge 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 Epiloge Ebene zu schützen

### Nachteile

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



vs/dl

Betriebssysteme (VL 5 | WS 18)

5 Unterbrechungen, Synchronisation – Prolog/Epilog-Modell

5–43

### Fazit

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



vs/dl

Betriebssysteme (VL 5 | WS 18)

5 Unterbrechungen, Synchronisation – Prolog/Epilog-Modell

5–44

## 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) [8]
  - Unterbrechungsbehandler ( $\rightarrow$  Prologue) können DPCs ( $\rightarrow$  Epiloge) in eine Warteschlange einhängen. Diese wird verzögert abgearbeitet, bevor die CPU auf Faden-Ebene zurückkehrt
- Linux: top halves / bottom halves, tasklets, irq threads [1, 5, 6]
  - Klassisch: Unterbrechungsbehandler (ISR) setzt Bit, durch das eine verzögerte bottom half (BH  $\rightarrow$  Epilog) angefordert werden kann.
  - Aktuell: BH  $\rightarrow$  Softirqs, dazu kommen tasklets (vgl. mit Windows DPCs) und interrupt threads.
- eCos: ISRs / deferred service routines (DSRs)
- ...

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



vs/dl

Betriebssysteme (VL 5 | WS 18)

5 Unterbrechungen, Synchronisation – Prolog/Epilog-Modell

5–45

## Agenda

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



vs/dl

Betriebssysteme (VL 5 | WS 18)

5 Unterbrechungen, Synchronisation – Zusammenfassung

5–46

## 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



vs/dl

Betriebssysteme (VL 5 | WS 18)

5 Unterbrechungen, Synchronisation – Zusammenfassung

5–47

## Referenzen

- Daniel P. Bovet und Marco Cesati. *Understanding the Linux Kernel*. O'Reilly, 2001. ISBN: 0-596-00002-2.
- Digital Equipment Corporation. *VAX-11 Architecture Reference Manual*. Document Number EK-VAXAR-RM-001. Digital Equipment Corporation. Maynard, MA, USA: Digital Press, Mai 1982.
- Samuel J. Leffler, Marshall Kirk McKusick, Michael J. Karels u. a. *The Design and Implementation of the 4.3 BSD UNIX Operating System*. Addison-Wesley, Mai 1989. ISBN: 0-201-06196-1.
- John Lions. *Lions' Commentary on UNIX (6th Edition)*. Peer-to-Peer Communications Inc., 1977. ISBN: 978-1573980135.
- Robert Love. *Linux Kernel Development (2nd Edition)*. Novell Press, 2005. ISBN: 978-0672327209.
- Valentin Rothberg. *Interrupt Handling in Linux*. Technical Report. Friedrich-Alexander-Universität Erlangen-Nürnberg, Lehrstuhl für Informatik 4, 2015. URL: [https://www4.cs.fau.de/~vrothberg/Interrupt\\_Handling\\_in\\_Linux.pdf](https://www4.cs.fau.de/~vrothberg/Interrupt_Handling_in_Linux.pdf).



vs/dl

Betriebssysteme (VL 5 | WS 18)

5 Unterbrechungen, Synchronisation – Referenzen

5–49

## Nachtrag: Mehrkernsysteme

### Beachte: Unterbrechungsbehandlung ≠ Parallelität

- Techniken funktionieren (so) nur bei echter Unterbrechungsmantik: 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! (Sequentialisierung, Priorisierung, *run-to-completion*)
  - Asymmetrie (UB unterbricht A) ist nicht länger gegeben (weiche Synchronisation wird dadurch viel schwieriger)
- Zusätzlich erforderlich: **Interprozessor-Synchronisation**
  - „hart“ ↪ zweiseitig blockierend, z. B. mit *Spin-Locks* ↪ Übung
  - „weich“ ↪ algorithmisch nichtblockierend (**schwer!**) ↪ [CS]



vs/dl

Betriebssysteme (VL 5 | WS 18)

5 Unterbrechungen, Synchronisation – Zusammenfassung

5–48

## Referenzen (Forts.)

- Friedrich Schön, Wolfgang Schröder-Preikschat, Olaf Spinczyk u. a. „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, März 2000, S. 270–277. DOI: 10.1109/ISORC.2000.839540.
- Wolfgang Schröder-Preikschat. *Concurrent Systems*. Vorlesung mit Übung. Friedrich-Alexander-Universität Erlangen-Nürnberg, Lehrstuhl für Informatik 4, 2015 (jährlich). URL: [https://www4.cs.fau.de/Lehre/WS15/V\\_CS](https://www4.cs.fau.de/Lehre/WS15/V_CS).
- Wolfgang Schröder-Preikschat. *Systemprogrammierung*. Vorlesung mit Übung. Friedrich-Alexander-Universität Erlangen-Nürnberg, Lehrstuhl für Informatik 4, 2015 (jährlich). URL: [https://www4.cs.fau.de/Lehre/WS15/V\\_SP](https://www4.cs.fau.de/Lehre/WS15/V_SP).
- David A. Solomon und Mark Russinovich. *Inside Microsoft Windows 2000 (3rd Edition)*. Microsoft Press, 2000. ISBN: 3-86063-630-8.



vs/dl

Betriebssysteme (VL 5 | WS 18)

5 Unterbrechungen, Synchronisation – Referenzen

5–50