

Betriebssystembau (BSB)

VL 5 – Unterbrechungssynchronisation

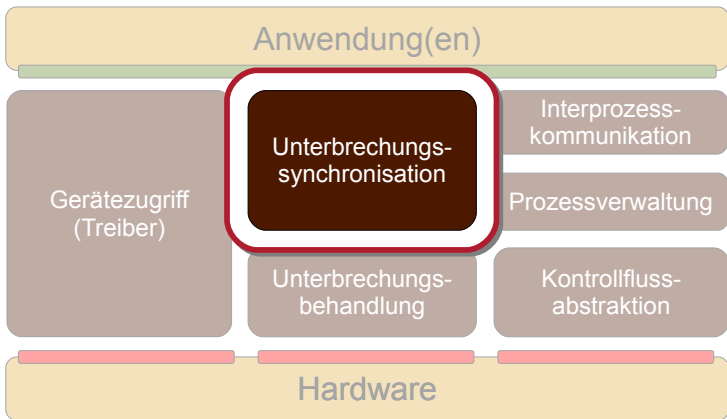
Alexander Lochmann

Lehrstuhl für Informatik 12 – Arbeitsgruppe Systemsoftware
Technische Universität Dortmund

<https://sys.cs.tu-dortmund.de/de/lehre/ws23/bsb>

WS 23 – 30. Oktober 2023

Überblick: Einordnung dieser VL



Betriebssystementwicklung

Agenda

Einleitung

Prioritätsebenenmodell

Harte Synchronisation

Weiche Synchronisation

Prolog/Epilog-Modell

Zusammenfassung

Referenzen



Einleitung

- Motivation

- Erstes Fazit

- Prioritätsebenenmodell

- Harte Synchronisation

- Weiche Synchronisation

- Prolog/Epilog-Modell

- Zusammenfassung

- Referenzen



Zustandsänderungen ...

- sind Sinn und Zweck der Unterbrechungsbehandlung
 - Gerätetreiber müssen über den Abschluss einer E/A Operation informiert werden
 - der Scheduler muss erfahren, dass eine Zeitscheibe abgelaufen ist
- müssen mit Vorsicht durchgeführt werden
 - Unterbrechungen können zu jeder Zeit auftreten
 - kritisch sind Daten/Datenstrukturen, die der normale Kontrollfluss die Unterbrechungsbehandlung sich teilen

Beispiele aus der letzten Vorlesung

Beispiel 2: Ringpuffer

auch die Pufferimplementierung ist kritisch ...

```
// 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; // Elementzähler merken
        if (elements == SIZE) return; // Element verloren
        buf[nextin] = data; // Element schreiben
        nextin++; nextin %= SIZE; // Zeiger weitersetzen
        occupied = elements + 1; // Zähler erhöhen
    }
    char consume() { // normaler Kontrollfluss:
        int elements = occupied; // Elementzähler merken
        if (elements == 0) return 0; // Puffer leer, kein Ergebnis
        char result = buf[nextout]; // Element lesen
        nextout++; nextout %= SIZE; // Lesezeiger weitersetzen
        occupied = elements - 1; // Zähler erniedrigen
        return result; // Ergebnis zurückliefern
    }
};
```

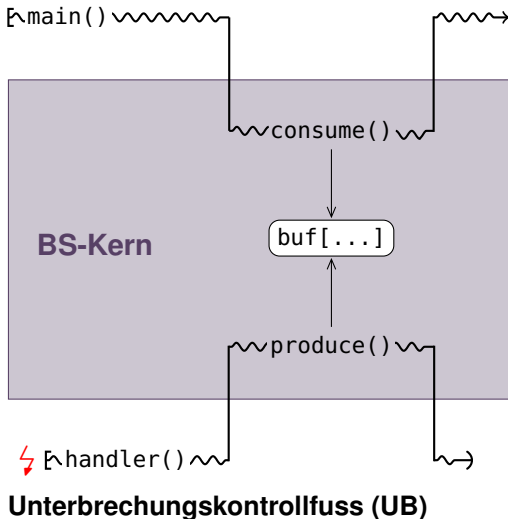
Motivation: Ursache

Kontrollflüsse
"von oben"

"begegnen"
sich im Kern

und "von unten"

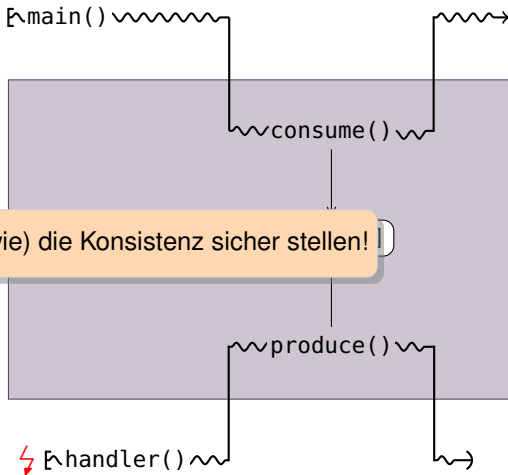
Anwendungskontrollfluss (A)



Motivation: Ursache

Kontrollflüsse
"von oben"

Anwendungskontrollfluss (A)



Wir müssen (irgendwie) die Konsistenz sicher stellen!

und "von unten"

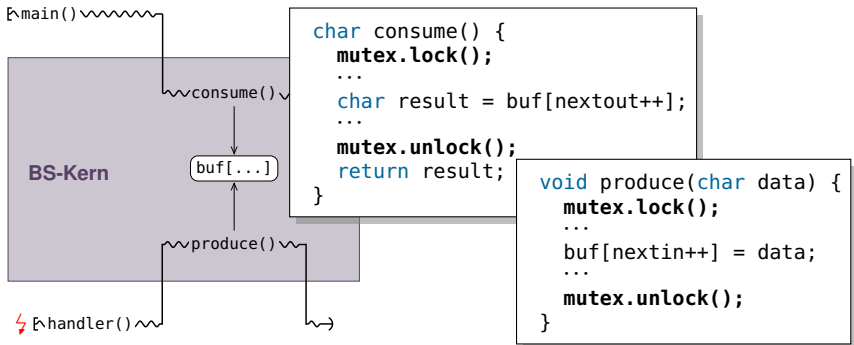
Unterbrechungskontrollfluss (UB)

Naiver Lösungsansatz

■ Zweiseitige Synchronisation

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

Anwendungskontrollfluss (A)



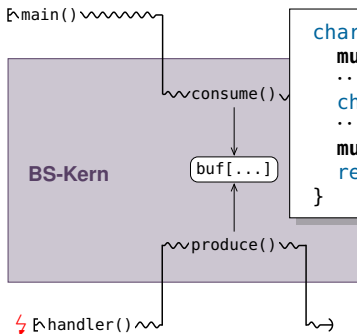
Unterbreckungskontrollfluss (UB)

Naiver Lösungsansatz

Zweiseitige Synchronisation

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

Anwendungskontrollfluss (A)



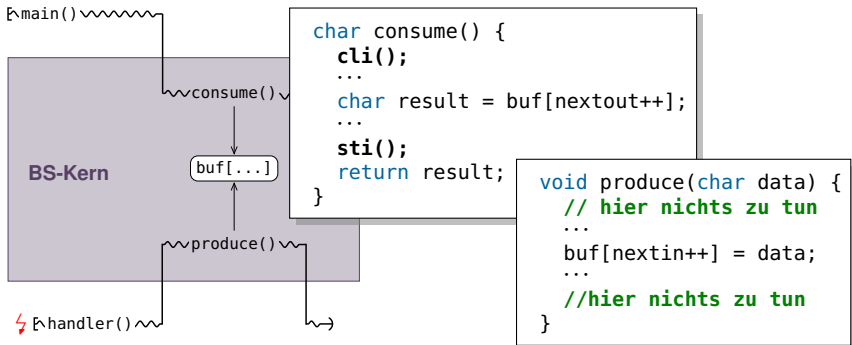
```
char consume() {  
    mutex.lock();  
    ...  
    char result = buf[nextout++];  
    ...  
    mutex.unlock();  
    return result;  
}
```

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

Unterbreckungskontrollfluss (UB)

- 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

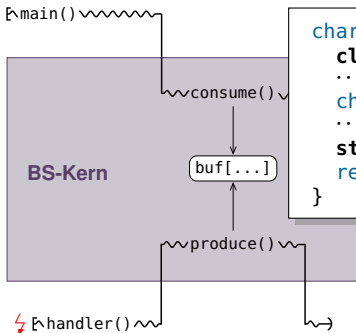
- Unterdrückung der Unterbrechungsbehandlung
- Operationen `disable_interrupts()` (im Folgenden o. B. d. A. in „Intel“-Schreibweise: `cli()/sti()`)

Einseitige Synchronisation

funktioniert.

[Warum?]

Anwendungskontrollfluss (A)



```
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
}
```

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**)

Einleitung

Prioritätsebenenmodell

Grundbegriffe

Verallgemeinerung

Konsistenzsicherung

Harte Synchronisation

Weiche Synchronisation

Prolog/Epilog-Modell

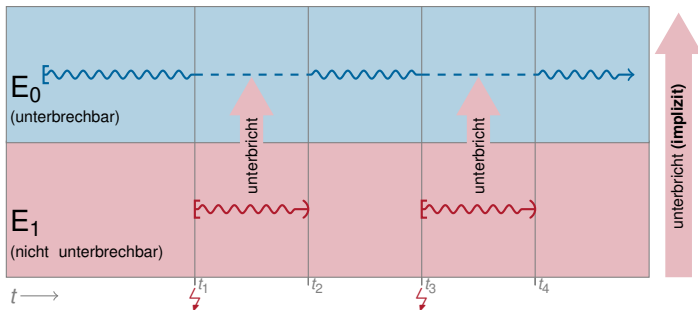
Zusammenfassung

Referenzen



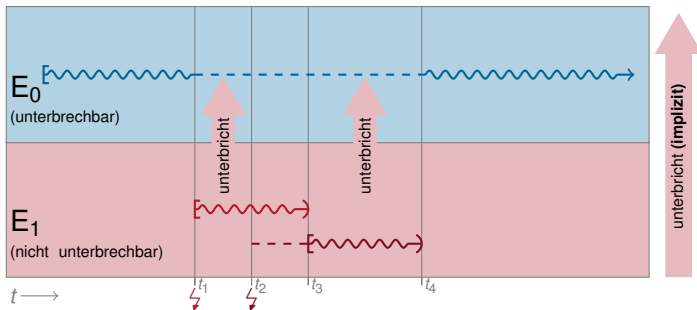
Prioritätsebenenmodell

- 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)



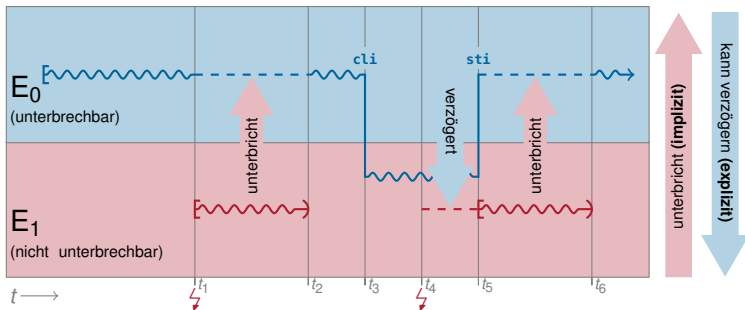
Prioritätsebenenmodell

- 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 (↔ Sequentialisierung)
 - Mit `sti` **wechselt** ein E_1 -Kontrollfluss explizit auf E_0
 - er ist ab dann (wieder) unterbrechbar
 - abhängige E_1 -Kontrollflüsse „schlagen durch“ (↔ Sequentialisierung)



Prioritätsebenenmodell

■ Verallgemeinerung für mehrere Unterbrechungsebenen:

■ Kontrollflüsse auf E_l werden

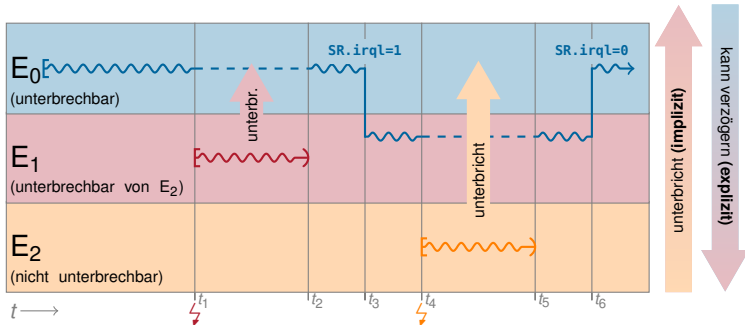
1. **jederzeit unterbrochen** durch Kontrollflüsse von E_m
2. **nie unterbrochen** durch Kontrollflüsse von E_k
3. **sequentialisiert** mit weiteren Kontrollflüssen von E_l

(für $m > l$)

(für $k \leq l$)

■ Kontrollflüsse können die Ebene **wechseln**

- durch spezielle Operationen (hier: Modifizieren des Statusregisters)



- 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 (\leftrightarrow Sequentialisierung)
 - „von unten“ (aus E_m mit $m > l$) durch **weiche Synchronisation**
 - **algorithmisch sicherstellen**, dass Unterbrechungen nicht stören
 - erfordert unterbrechungstransparente Algorithmen



Einleitung

Prioritätsebenenmodell

Harte Synchronisation

Ansatz

Bewertung

Weiche Synchronisation

Prolog/Epilog-Modell

Zusammenfassung

Referenzen

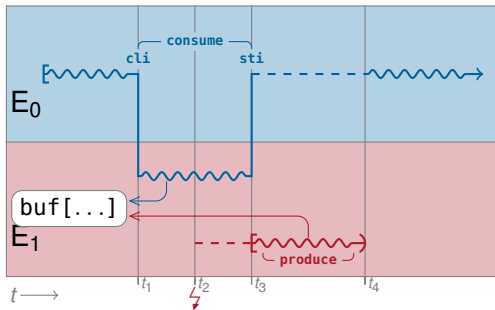


Bounded Buffer – Lösung mit harter Synchronisation

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  
}
```



Zustand liegt
(logisch) auf E_1

■ 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 eintritt, sehr klein ist.

- 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



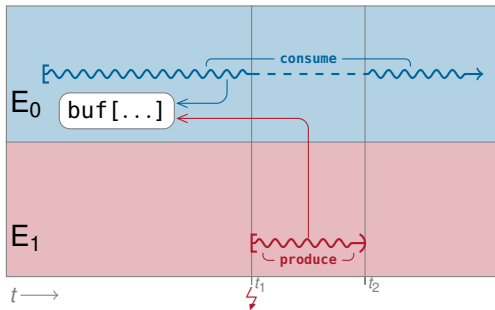
Bounded Buffer – Ansatz mit weicher Synchronisation

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_0



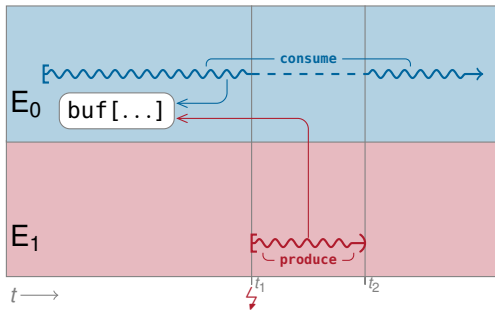
Bounded Buffer – Konsistenzbedingungen, Annahmen

■ 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*)



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
    }
};
```

Kritisch ist der gemeinsam verwendete Zustand

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

```
// 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
    }
};
```

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.

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

■ aus der Sicht von `consume()`

- vor dem Lesen von **nextin**
- nach dem Lesen von **nextin**

\Leftrightarrow `consume()` nach `produce()`



\Leftrightarrow `consume()` vor `produce()`



■ aus der Sicht von `produce()`

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

\Leftrightarrow `produce()` vor `consume()`



\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.

■ 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 zutrif – wurden wir unterbrochen?
3. falls unterbrochen, setze neu auf ab Schritt 1



Systemzeit – Neue Implementierung

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

```
/* Systemzeit abfragen */  
time_t time () {  
    time_t res;  
    do {  
        interrupted = false;  
        res = global_time;  
    } while (interrupted);  
    return res;  
}
```

```
/* Unterbrechungsbehandlung */  
void timerHandler () {  
    interrupted = true;  
    global_time++;  
}
```

Konsistenzbedingung ist
nun in jedem Fall erfüllt!

■ 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 ~> 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)
 - Kleinste Änderungen können die Konsistenzgarantie zerstören
 - Codegenerierung des Compilers ist zu beachten
- Bei größeren Datenmengen steigen die Wiederaufsetzkosten

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

Zusammenfassung

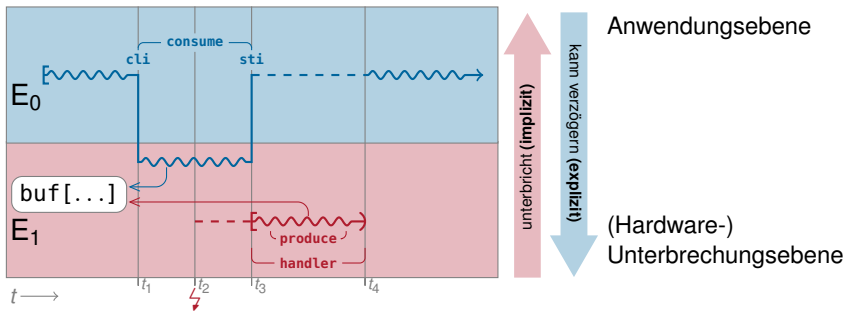
Referenzen



Prolog/Epilog-Modell – Motivation

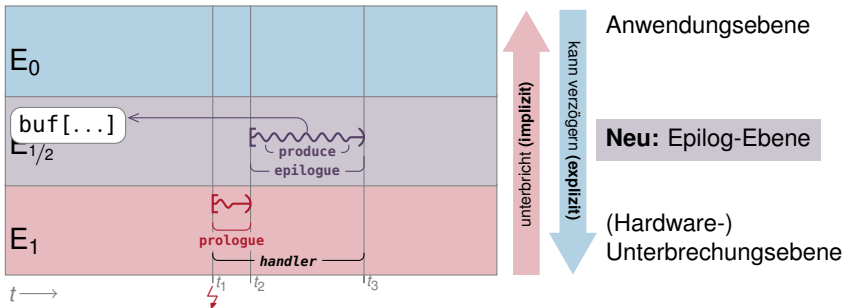
■ Reprise: Harte Synchronisation

- 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 $E_{1\dots n}$
 - Unterbrechungsbehandlung wird **zweigeteilt** in **Prolog** und **Epilog**
 - **Prolog** arbeitet auf Unterbrechungsebene $E_{1\dots n}$
 - **Epilog** arbeitet auf der neuen (Software-)Ebene $E_{1/2}$ (**Epilogebe**ne)
 - Zustand liegt (so weit wie möglich) auf der Epilogebe n e
 - eigentliche Unterbrechungsbehandlung wird nur noch kurz gesperrt



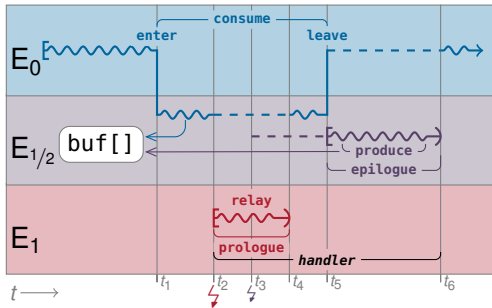
- Unterbrechungsbehandlungsroutrinen werden zweigeteilt (vgl. VL5)
 - beginnen im **Prolog** (fast immer)
 - werden fortgesetzt im **Epilog** (bei Bedarf)
- **Prolog** (↪ Hardwareunterbrechung, vgl. VL4)
 - läuft auf Hardwareunterbrechungsebene
 - hat damit Priorität über Anwendungsebene und Epilogebe
 - 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, vgl. SoftIRQ, VL5)
 - läuft auf Epilogebe $E_{1/2}$ (zusätzliche Kontrollflussebene)
 - Ausführung erfolgt verzögert zum Prolog
 - erledigt die eigentliche Arbeit (↪ Latenzverbergung)
 - hat Zugriff auf größten Teil des Zustands
 - Zustand wird auf Epilogebe

- Die Epilogebeine 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 Epilogebeine $E_{1/2}$ werden
 1. **jederzeit unterbrochen** durch Kontrollflüsse der Ebenen $E_{1...n}$
 - ↪ Prologe (Unterbrechungen) haben Priorität über Epiloge
 2. **nie unterbrochen** durch Kontrollflüsse der Ebene E_0
 - ↪ Epiloge 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.

- Benötigt werden Operationen, um
 1. explizit die Epilogebeene zu betreten: **enter()**
 - entspricht dem `cli` bei der harten Synchronisation
 2. explizit die Epilogebeene 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_1 -Unterbrechungen werden nie gesperrt.

Aktivierungslatenz der Unterbrechungsbehandlung ist minimal.

- t_1 Anwendungskontrollfluss betritt Epilogebe $E_{1/2}$ (`enter()`).
- t_2 Unterbrechung ⚡ auf Ebene E_1 wird signalisiert \rightsquigarrow Prolog wird ausgeführt.
- t_3 Prolog fordert Epilog für die nachgeordnete Bearbeitung an (`relay()` ⚡).
- t_4 Prolog terminiert, unterbrochener $E_{1/2}$ -Kontrollfluss läuft weiter.
- t_5 Anwendungskontrollfluss verlässt die Epilogebe $E_{1/2}$ (`leave()`)
 \rightsquigarrow zwischenzeitlich aufgelaufene Epiloge werden nun abgearbeitet.
- t_6 Epilog terminiert, Anwendungskontrollfluss fährt auf E_0 fort.

■ Benötigt werden Operationen, um

1. explizit die Epilogebeene zu betreten: **enter()**
 - entspricht dem `cli` bei der harten Synchronisation
2. explizit die Epilogebeene 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!

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_0 zurückkehrt!

1. bei explizitem Verlassen der Epilogebeene mit `leave()`
 - während der Anwendungskontrollfluss auf $E_{1/2}$ gearbeitet hat könnten Epiloge aufgelaufen sein (\leftrightarrow Sequentialisierung).
2. nach Abarbeitung des letzten Epilogs
 - während der Epilogabarbeitung könnten weitere Epiloge aufgelaufen sein (\leftrightarrow Sequentialisierung).
3. wenn der **letzte** Unterbrechungsbehandler terminiert
 - während der Abarbeitung von $E_{1\dots n}$ -Kontrollflüssen könnten Epiloge aufgelaufen sein (\leftrightarrow Priorisierung).

■ Implementierungsvarianten

- rein softwarebasiert
- mit Hardwareunterstützung durch einen **AST**

(\rightsquigarrow Übung)

(\rightsquigarrow [1, 2])

■ 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 (\leftrightarrow unsere $E_{1/2}$)
 - Gesetzmäßigkeiten des Ebenenmodells gelten
(AST-Ausführung ist verzögerbar, wird automatisch aktiviert, ...)

■ Sicherstellung der Epilogabarbeitung wird damit sehr einfach!

- Abarbeitung der Epilog erfolgt im AST
 - \rightsquigarrow und damit automatisch, bevor die CPU auf E_0 zurückkehrt
- bleibt nur noch die Verwaltung der anhängigen Epilog

- Beispiel TriCore: Implementierung mit AST
 - AST hier als Unterbrechung der E_1 konfiguriert (\Leftrightarrow 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 (anhangiger
}                               // 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>
    }
}
```

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

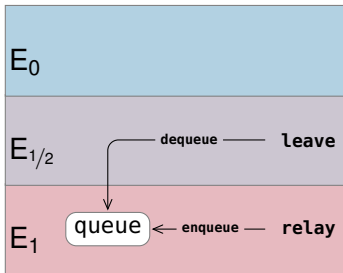
```
void enter() {
    CPU::setIRQL(1);           // be
}
void leave() {
    CPU::setIRQL(0);          // en
}                               // AS
void relay(<Epilog>) {
    <haenge Epilog an queue an>
    CPU_SRC1::trigger();      // aktiviere Level-1 IRQ (AST)
}
void __attribute__((interrupt_handler)) 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?

- 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 (\Leftrightarrow Ausführungszeit von `dequeue()`) **kurz** und **deterministisch** ist.

Eine Lösung mit **weicher Synchronisation** (z. B. [3]) wäre natürlich schöner!

■ Vorteile

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

■ Nachteile

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

Fazit

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



Agenda

Einleitung

Prioritätsebenenmodell

Harte Synchronisation

Weiche Synchronisation

Prolog/Epilog-Modell

Zusammenfassung

Referenzen



- **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



Beachte: Unterbrechungsbehandlung \neq Parallelität

- Techniken funktionieren (so) nur bei echter Unterbrechungsemantik: 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“ \rightarrow zweiseitig blockierend, z. B. mit *Spin-Locks* \rightsquigarrow Übung
 - „weich“ \rightarrow algorithmisch nichtblockierend (**schwer!**)

- [1] Digital Equipment Corporation. *VAX-11 Architecture Reference Manual*. Document Number EK-VAXAR-RM-001. Digital Equipment Corporation. Maynard, MA, USA: Digital Press, Mai 1982.
- [2] Samuel J. Leffler, Marshall Kirk McKusick, Michael J. Karels u. a. *The Design and Implementation of the 4.3BSD UNIX Operating System*. Addison-Wesley, Mai 1989. isbn: 0-201-06196-1.
- [3] 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: <http://doi.ieeecomputersociety.org/10.1109/ISORC.2000.839540>.
- [BS] Peter Ulbrich. *Betriebssysteme*. Vorlesung mit Übung. Technische Universität Dortmund, Lehrstuhl für Informatik 12, 2021 (jährlich). url: <https://sys-sideshow.cs.tu-dortmund.de/Teaching/SS2021/BS/>.