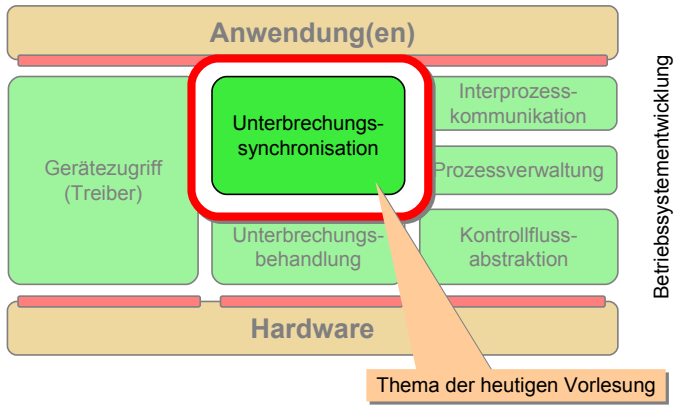
Betriebssysteme (BS)

Unterbrechungen
- Synchronisation -

Daniel Lohmann

Lehrstuhl für Informatik 4
Verteilte Systeme und Betriebssysteme

Überblick: Vorlesungen



Agenda

- Motivation / Problem
- Kontrollflussebenenmodell
- Harte Synchronisation
- Weiche Synchronisation
- Synchronisation mit dem Prolog/Epilog-Modell
- Zusammenfassung

Agenda

- **Motivation / Problem**
- Kontrollflussebenenmodell
- Harte Synchronisation
- Weiche Synchronisation
- Synchronisation mit dem Prolog/Epilog-Modell
- Zusammenfassung

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 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

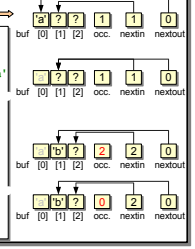
auch die Pufferimplementierung ist kritisch ...

Ausführung / Zustand

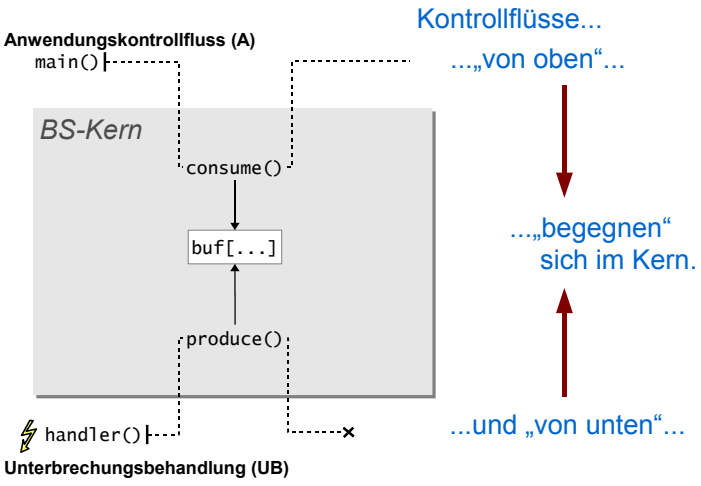
```
char consume() {
  int elements = occupied; // 1
  if (elements == 0) return 0;
  char result = buf[nextout]; //'a'
  nextout++; nextout %= SIZE;

void produce(char data) { // 'b'
  int elements = occupied; // 1!
  if (elements == SIZE) return;
  buf[nextin] = data;
  nextin++; nextin %= SIZE;
  occupied = elements + 1; // 2
}

  occupied = elements - 1; // 0
  return result; // 'a'
}
```

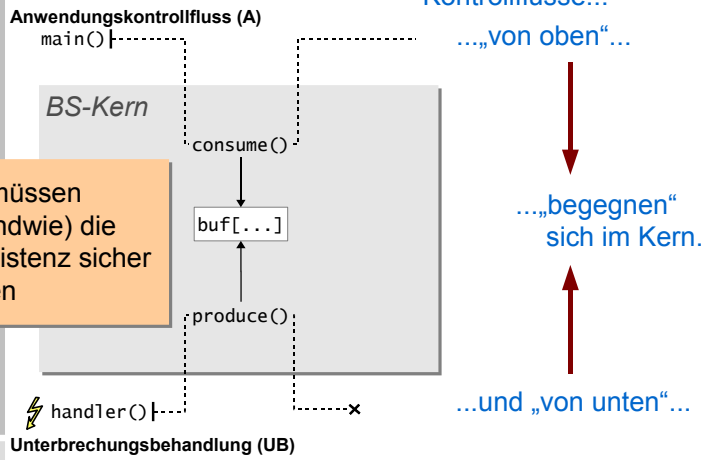


Motivation: Ursache



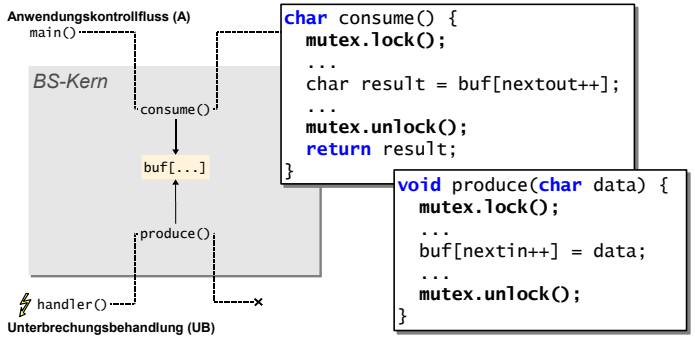
Motivation: Ursache

Wir müssen (irgendwie) die Konsistenz sicher stellen



Naiver Lösungsansatz

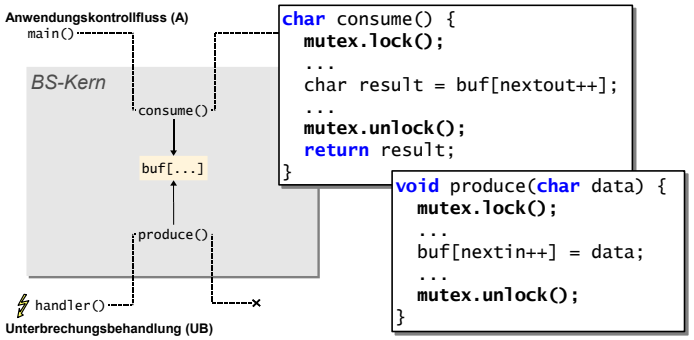
- Zweiseitige Synchronisation
  - gegenseitiger Ausschluss durch Schlossvariable (Mutex, *Spin-Lock*, ...)
  - wie zwischen zwei Prozessen



Naiver Lösungsansatz

- Zweiseitige Synchr
  - gegenseitiger Aussch (Mutex, *Spin-Lock*, ...
  - wie zwischen zwei Pr

Zweiseitige Synchronisation funktioniert **natürlich nicht**!

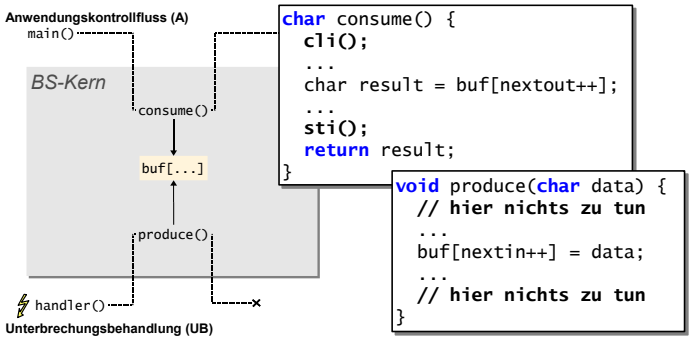



```
char consume() {
  mutex.lock();
  ...
  char result = buf[nextout++];
  ...
  mutex.unlock();
  return result;
}
```

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


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()`)




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

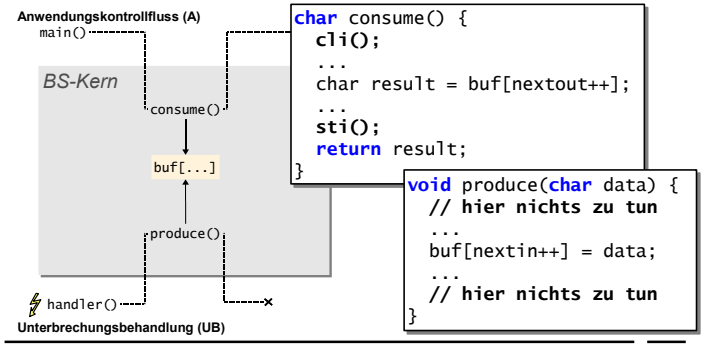
```
void produce(char data) {
  // hier nichts zu tun
  ...
  buf[nextin++] = data;
  ...
  // hier nichts zu tun
}
```


Besserer Lösungsansatz

- Einseitige Synchro
  - Unterdrückung der U
  - Operationen `disable` (im Folgenden o.B.d.A. in „

Mit einseitiger Synchronisation funktioniert es...

[warum eigentlich?]




```
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
}
```


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 der letzten Vorlesung)
- Synchronisation / Konsistenzsicherung erfolgt **einseitig**

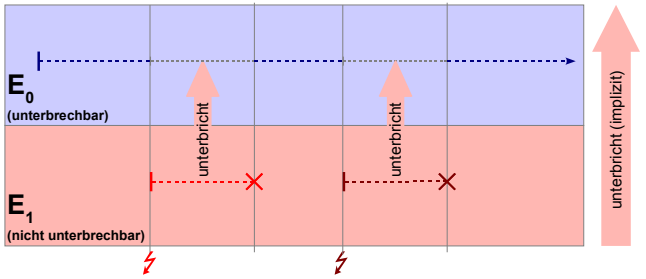
Diese Tatsachen müssen wir **beachten**!
(Das heißt aber auch: Wir können sie **ausnutzen**)

Agenda

- Motivation / Problem
- **Kontrollflussebenenmodell**
- Harte Synchronisation
- Weiche Synchronisation
- Synchronisation mit dem Prolog/Epilog-Modell
- Zusammenfassung

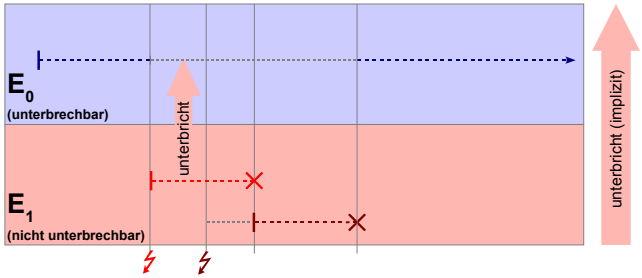
Kontrollflussebenenmodell

- 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 andere $E_{0/1}$-Kontrollflüsse)
  - E_1-Kontrollflüsse haben Priorität über E_0-Kontrollflüsse



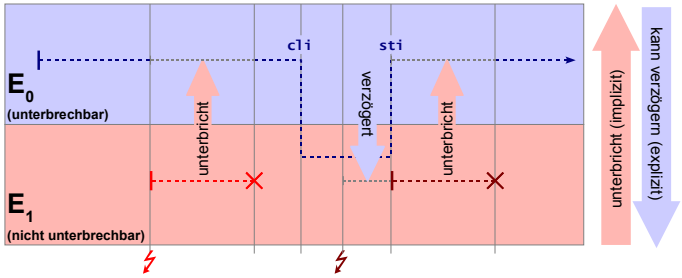
Kontrollflussebenenmodell

- 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 PIC die Strategie



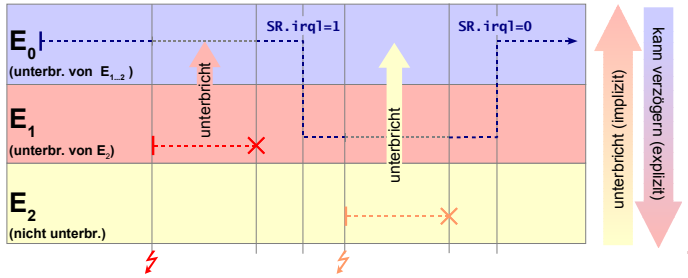
Kontrollflussebenenmodell

- 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
    - anhängige E_1-Kontrollflüsse „schlagen durch“ (→Sequentialisierung)



Kontrollflussebenenmodell

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



Kontrollflussebenen: Konsistenzsicherung

- Jede Zustandsvariable ist (logisch) genau einer Ebene E_l zugeordnet
  - Zugriffe aus E_l sind implizit konsistent (→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 (→Sequentialisierung)
  - „von unten“ (aus E_m mit $m > l$) durch **weiche Synchronisation**
    - **algorithmisch sicherstellen**, dass Unterbrechungen nicht stören
    - erfordert unterbrechungstransparente Algorithmen

Agenda

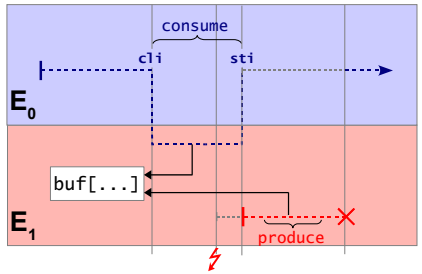
- Motivation / Problem
- Kontrollflussebenenmodell
- **Harte Synchronisation**
- Weiche Synchronisation
- Synchronisation mit dem Prolog/Epilog-Modell
- Zusammenfassung

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
    - man ist (weitgehend) unabhängig davon, was der Compiler macht
  - einfach anzuwenden (für den Entwickler), funktioniert immer
    - im Zweifelsfall legt man sämtlichen Zustand auf die höchstpriore Ebene
- **Nachteile**
  - Breitbandwirkung
    - es werden pauschal alle Unterbrechungsbehandlungen (Kontrollflüsse) auf und unterhalb der Zustandsebene verzögert
  - Prioritätsverletzung
    - es werden Kontrollflüsse verzögert, die eine höhere Priorität haben
  - prophylaktisches Verfahren
    - Nachteile werden in Kauf genommen, obwohl die Wahrscheinlichkeit, dass tatsächlich eine relevante Unterbrechung eintrifft, sehr klein ist.

Harte Synchronisation: Bewertung

- 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 (anzunehmende) **Latenz**
  - Wird die Latenz zu hoch, können Daten verloren gehen.
    - *edge-triggered* Unterbrechungen werden nicht bemerkt
    - Daten werden zu langsam von EA-Gerät abgeholt

Fazit: Harte Synchronisation ist eher **ungeeignet** für die Konsistenzsicherung **komplexer Datenstrukturen**.

Agenda

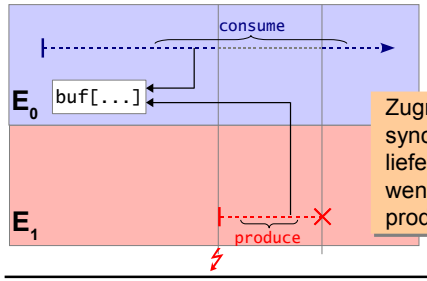
- Motivation / Problem
- Kontrollflussebenenmodell
- Harte Synchronisation
- **Weiche Synchronisation**
- Synchronisation mit dem Prolog/Epilog-Modell
- Zusammenfassung

Bounded Buffer – Ansatz mit weicher Synchronisation

Zustand liegt (logisch) auf E_0

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

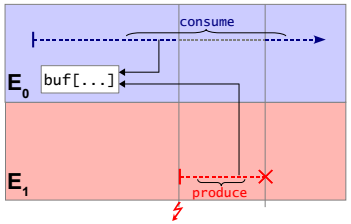
```
void produce(char data) {
  ?
}
```



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

Bounded Buffer – Konsistenzbedingungen, Annahmen

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



Bounded Buffer – Implementierung aus der letzten VL

- Kritisch ist der gemeinsam verwendete Zustand

```
// Pufferklasse in C++ (bekannt aus der letzten Vorlesung)
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
} };
```


Bounded Buffer – Implementierung aus der letzten VL

- Kristisch ist der gemeinsam verwendete Zustand

```
// Pufferklasse in C++ (bekannt aus der letzten Vorlesung)
class BoundedBuffer {
  char buf[SIZE]; int occupied; in
public:
  BoundedBuffer(): occupied(0), ne
  void produce(char data) {
    int elements = occupied;
    if (elements == SIZE) return;
    buf[nextin] = data;
    nextin++; nextin %= SIZE;
    occupied = elements + 1;       // Zähler erhöhen
  }
  char consume() {
    int elements = occupied;
    if (elements == 0) return 0;
    char result = buf[nextout];
    nextout++; nextout %= SIZE;
    occupied = elements - 1;       // Zähler erniedrigen
    return result;
} };
```

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

Bounded Buffer – Neue 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 – Neue 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.

Bounded Buffer – Neue 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 in `consume()` erfolgt...
  - aus der Sicht von `consume()`
    - vor dem Lesen von **nextin** ⇔ `consume()` nach `produce()` ✔
    - nach dem Lesen von **nextin** ⇔ `consume()` vor `produce()` ✔
  - aus der Sicht von `produce()`
    - vor dem Schreiben von **nextout** ⇔ `produce()` vor `consume()` ✔
    - nach dem Schreiben von **nextout** ⇔ `produce()` nach `consume()` ✔

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

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

Konsistenzbedingung ist in jedem Fall erfüllt.

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++;
}
```

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 beliebigen sequentiellen Ausführung der Operationen
    - *entweder* `time()` vor `timerHandler()` *oder umgekehrt*
- Annahmen
  - `timerHandler()` unterbricht `time()`
    - alle anderen Kombinationen kommen nicht vor
  - `timerHandler()` läuft immer komplett durch
- Lösungansatz: In `time()` **optimistisch** herangehen
  1. lese Daten unter der Annahme, nicht unterbrochen zu werden
  2. überprüfe, ob Annahme korrekt war (wurden wir unterbrochen?)
  3. falls unterbrochen, setze wieder 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;
}
```

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

Konsistenzbedingung wird nun in jedem Fall erfüllt.

Weiche Synchronisation: Bewertung

- **Vorteile**
  - Konsistenz ist sicher gestellt (durch Unterbrechungstransparenz)
  - Priorität wird nie verletzt
    - Kontrollflüsse der höherprioren 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 denn überhaupt eine Lösung findet, ist diese in der Regel schwer zu verstehen und noch schwieriger zu verifizieren
  - Lösungen funktionieren nur unter zahlreichen Randbedingungen
    - kleinste Änderungen im Code können die Konsistenzgarantie zerstören
    - Codegenerierung des Compilers ist zu beachten
  - Bei größeren Datenmengen steigen die Kosten für Wiederaufsetzen

Weiche Synchronisation: Bewertung

Fazit:
Weiche Synchronisation durch Unterbrechungstransparenz ist eine feine Sache. Es handelt sich bei den Algorithmen jedoch immer um **Speziallösungen** für **Spezialfälle**.

Als allgemein verwendbares Mittel für die Konsistenzsicherung **komplexer Datenstrukturen** ist sie damit **nicht geeignet**.

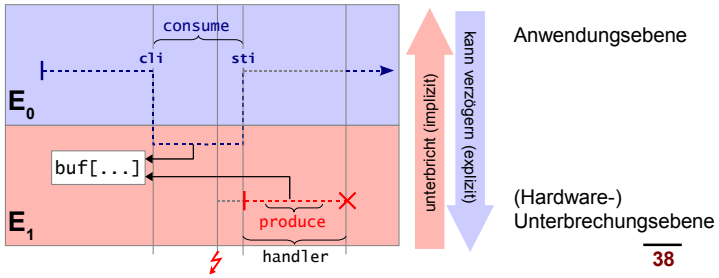
- **Nachteile**
  - Lösungen häufig sehr komplex
    - wenn man denn überhaupt eine Lösung findet, ist diese in der Regel schwer zu verstehen und noch schwieriger zu verifzieren
  - Lösungen funktionieren nur unter zahlreichen Randbedingungen
    - kleinste Änderungen im Code können die Konsistenzgarantie zerstören
    - Codegenerierung des Compilers beachten
  - Bei größeren Datenmengen steigen die Kosten für Wiederaufsetzen

Agenda

- Motivation / Problem
- Kontrollflussebenenmodell
- Harte Synchronisation
- Weiche Synchronisation
- **Synchronisation mit dem Prolog/Epilog-Modell**
- Zusammenfassung

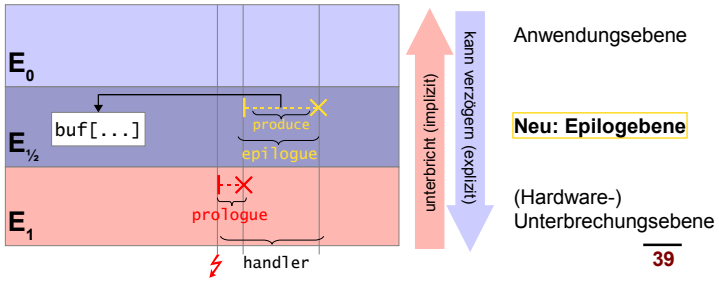
Prolog/Epilog-Modell – Motivation

- Noch einmal: Harte Synchronisation
  - einfach, korrekt, „funktioniert immer“ ✔
  - Hauptproblem ist die hohe Latenz ✘
    - lange Verzögerung bei **Zugriff auf den Zustand** aus den höheren Ebenen
    - lange 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

- **Idee:** Wir fügen eine **weitere Ebene $E_{½}$** zwischen der Anwendungsebene E_0 und den UB-Ebenen $E_{1...n}$ ein
  - UB wird **zweigeteilt** in ***Prolog*** und ***Epilog***
    - **Prolog** arbeitet auf Unterbrechungsebene $E_{1...n}$
    - **Epilog** arbeitet auf der neuen (Software-)Ebene $E_{½}$ (**Epilogebene**)
  - Zustand liegt (so weit wie möglich) auf der Epilogebene
    - eigentliche Unterbrechungsbehandlung wird nur noch kurz gesperrt



Prolog/Epilog-Modell – Ansatz

- Unterbrechungsbehandlungsroutinen werden zweigeteilt
  - beginnen in ihrem Prolog (immer)
  - werden fortgesetzt in ihrem Epilog (bei Bedarf)
- Prolog
  - läuft auf Hardwareunterbrechungsebene
    - Priorität über Anwendungsebene und Epilogebene
  - ist kurz, fasst wenig oder gar keinen Zustand an
    - **Hardwareunterbrechungen bleiben nur kurz gesperrt**
  - kann bei Bedarf einen Epilog anfordern
- Epilog
  - läuft auf Epilogebene (zusätzliche Kontrollflussebene)
    - Ausführung erfolgt verzögert zum Prolog
  - erledigt die eigentliche Arbeit
  - hat Zugriff auf größten Teil des Zustands
    - Zustand wird auf Epilogebene synchronisiert

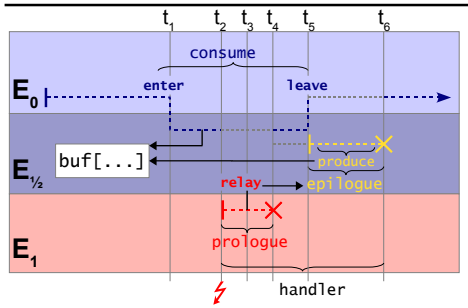
Prolog/Epilog-Modell – Epilogebene

- Die Epilogebene wird (ganz oder teilweise) in **Software** implementiert
  - trotzdem handelt es sich um eine ganz normale Kontrollflussebene des Ebenenmodells
  - es müssen daher auch dieselben Gesetzmäßigkeiten gelten
- Es gilt: Kontrollflüsse auf der Epilogebene $E_{½}$ 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 $E_{½}$
     → Anhängige Epiloge werden nacheinander abgearbeitet.
     → Bei Rückkehr zur Anwendungsebene sind alle Epiloge abgearbeitet.

Prolog/Epilog-Modell – Implementierung

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

Prolog/Epilog-Modell – Ablaufbeispiel



E_1-Unterbrechungen werden nie gesperrt.

Aktivierungslatenz der Unterbrechungs-behandlung ist minimal.

1 Anwendungskontrollfluss betritt Epilogebene $E_{½}$ (enter).
2 Unterbrechung auf Ebene E_1 wird signalisiert, Prolog wird ausgeführt.
3 Prolog fordert Epilog für die nachgeordnete Bearbeitung an (relay).
4 Prolog terminiert, unterbrochener $E_{½}$-Kontrollfluss (Anwendung) läuft weiter.
5 Anwendungskontrollfluss verlässt die Epilogebene $E_{½}$ (leave), zwischenzeitlich aufgelaufene Epiloge werden abgearbeitet.
6 Epilog terminiert, Anwendungskontrollfluss fährt auf E_0 fort.

Prolog/Epilog-Modell – Implementierung

- Benötigt werden Operationen, um
  1. explizit die Epilogebene zu betreten: `enter()`
     - entspricht dem `cli` bei der harten Synchronisation
  2. explizit die Epilogebene zu verlassen: `leave()`
     - entspricht dem `sti` bei der harten Synchronisation
  3. einen Epilog anzufordern: `relay()`
     - entspricht dem Hochziehen der entsprechenden 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 Teil muss etwas genauer betrachtet werden.

Prolog/Epilog-Modell – Implementierung

- Wann müssen anhängige Epiloge abgearbeitet werden?

Immer unmittelbar, bevor die CPU auf E_0 zurückkehrt!

1. bei explizitem Verlassen der Epilogebene mit `leave()`
   - während der Anwendungskontrollfluss auf Epilogebene gearbeitet hat könnten weitere Epiloge aufgelaufen sein (→ Sequentialisierung)
2. nach Abarbeitung des letzten Epilogs
   - während der Epilogabarbeitung könnten weitere Epiloge aufgelaufen sein
3. wenn der **letzte** Unterbrechungsbehandler terminiert
   - während die CPU Kontrollflüsse der Ebenen $E_{1...n}$ ausgeführt hat, könnten Epiloge aufgelaufen sein (→ Priorisierung)

- Zwei Implementierungsvarianten
  - mit Hardwareunterstützung durch einen AST (im Folgenden)
  - rein softwarebasiert (in der Übung)

Prolog/Epilog-Modell – Implementierung

- 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
- Im Unterschied zu Traps / Exceptions / Softwareinterrupts wird ein angeforderter AST **asynchron** abgearbeitet.
  - AST läuft auf einer eigenen Unterbrechungsebene zwischen der Anwendungsebene und den Hardware-UBs (unsere $E_{½}$)
  - Gesetzmäßigkeiten des Ebenenmodells gelten (AST-Ausführung ist verzögerbar, wird automatisch aktiviert, ...)
- Sicherstellung der Epilogabarbeitung damit sehr einfach!
  - Abarbeitung der Epiloge erfolgt im AST
    - und damit automatisch, bevor die CPU auf E_0 zurückkehrt
  - bleibt nur die Verwaltung der anhängigen Epiloge

Prolog/Epilog-Modell – Implementierung

- Beispiel TriCore: Implementierung mit AST
  - AST hier als Unterbrechung der E_1 implementiert (⇔ unsere $E_{½}$)
  - Hardwareunterbrechungen laufen auf $E_{2...n}$

```
void enter() {
  CPU::setIRQL(1);              // betrete E1, verzögere AST
}
void leave() {
  CPU::setIRQL(0);              // erlaube AST (anhängiger
}                               // AST würde jetzt abgearbeitet)
void relay(<Epilog>) {
  <hänge 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 implementiert (⇔ unsere $E_{½}$)
  - Hardwareunterbrechungen laufen auf $E_{2...n}$

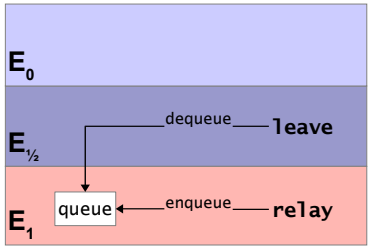
```
void enter() {
  CPU::setIRQL(1);
}
void leave() {
  CPU::setIRQL(0);
}                               // AST würde jetzt abgearbeitet)
void relay(<Epilog>) {
  <hänge Epilog an queue an>
  CPU_SRC1::trigger();          // aktiviere Level-1 IRQ (AST)
}
void __attribute__((interrupt_handler)) irq1Handler()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

Bietet die Hardware, wie z.B. Intel 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 Prologen und der Epilog-Ebene
    - muss entweder hart synchronisiert werden (im Bild dargestellt)
    - 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 wäre natürlich trozdem schön!

Prolog/Epilog-Modell – Verwandte Konzepte

- Windows: *ISRs* / *deferred procedure calls* (DPCs)
  - Unterbrechungsbehandler (ISRs) können DPCs ein eine Warteschlange einhängen. Diese wird verzögert abgearbeitet, bevor die CPU auf Faden-Ebene zurückkehrt
- Linux: *ISRs* / *bottom halves* (BHs)
  - Zu jedem Unterbrechungsbehandler (ISR) gibt es ein Bit in einer Bitmaske, durch das eine verzögerte *bottom half* angefordert werden kann. Die BHs werden vor dem Verlassen des Kerns ausgeführt.
  - Neuere Linux-Kerne verwenden ein DPC-ähnliches Konzept mit Warteschlangen von *tasklets*.
- eCos: *ISRs* / *deferred service routines (DSRs)*
- …

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

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
  - Unterbrechungsperren lassen sich nicht vollständig vermeiden
    - Gemeinsamer Zustand von Pro- und Epilog muss weiter hart oder weich synchronisiert werden

Prolog/Epilog-Modell: Bewertung

Fazit:
Das Prolog/Epilog-Modell ist ein **guter Kompromiss** für die Synchronisation bei Zugriffen auf den Kernzustand.

Es ist auch für die Konsistenzsicherung **komplexer Datenstrukturen geeignet**.

- 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
  - Unterbrechungsperren lassen sich nicht vollständig vermeiden
    - Gemeinsamer Zustand von Pro- und Epilog muss weiter hart oder weich synchronisiert werden

Agenda

- Motivation / Problem
- Kontrollflussebenenmodell
- Harte Synchronisation
- Weiche Synchronisation
- Synchronisation mit dem Prolog/Epilog-Modell
- **Zusammenfassung**

Zusammenfassung: Unterbrechungssynchronisation

- Konsistenzsicherung im BS-Kern
  - muss anders erfolgen als zwischen Prozessen → einseitig
  - Kontrollflüsse arbeiten auf verschiedenen Ebenen
- 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
  - Pro-/Epilog-basierte Synchronisation (durch Zweiteilung der Unterbrechungsbehandlung)
    - guter Kompromiss, Synchronisation ohne Auswirkungen auf die Latenz