Betriebssysteme (BS)

VL 6.4 – Unterbrechungen, Synchronisation – Prolog/Epilog-Modell

Volkmar Sieh / **Daniel Lohmann**

Lehrstuhl für Informatik 4
Verteilte Systeme und Betriebssysteme

Friedrich-Alexander-Universität
Erlangen Nürnberg

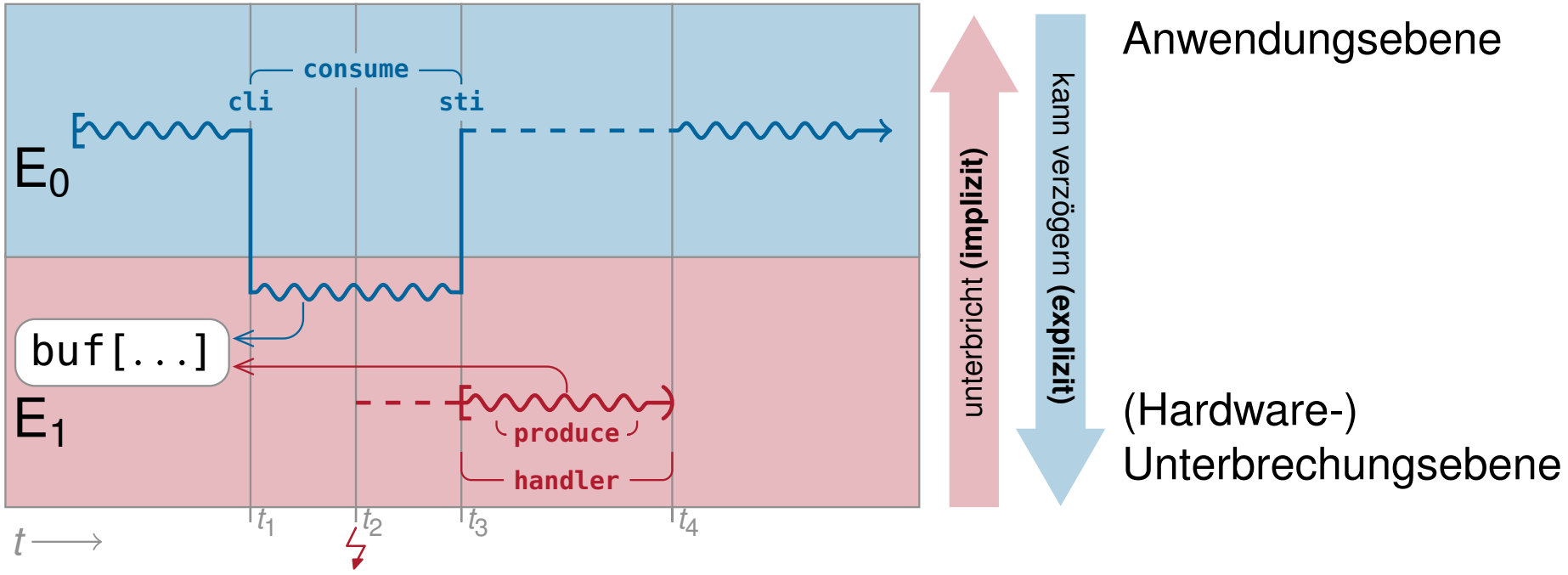
WS 20 – 30. November 2020

https://www4.cs.fau.de/Lehre/WS20/V_BS

Agenda

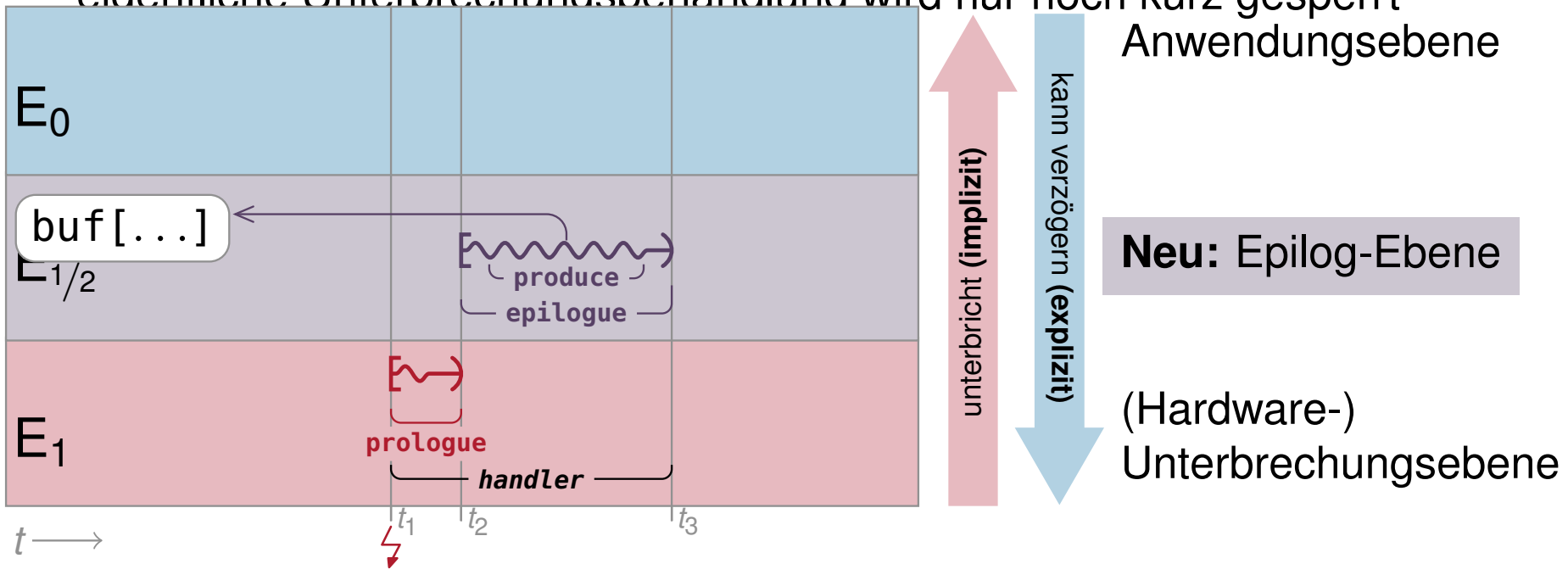
Prolog/Epilog-Modell – Motivation

- **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 $E_{1\ldots n}$ 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 $E_{1\ldots n}$
  - Unterbrechungsbehandlung wird zweigeteilt in Prolog und Epilog
    - Prolog arbeitet auf Unterbrechungsebene $E_{1\ldots n}$
    - 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.)

- Unterbrechungsbehandlungsroutinen werden zweigeteilt
  - beginnen im **Prolog** (immer)
  - werden fortgesetzt im **Epilog** (bei Bedarf)
- **Prolog** ($\rightsquigarrow$ 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 ($\rightsquigarrow$ Latenzminimierung)
    - kann bei Bedarf einen Epilog für die weitere Verarbeitung anfordern
- **Epilog** ($\rightsquigarrow$ Softwareunterbrechung)
  - läuft auf Epilogebene $E_{1/2}$ (zusätzliche Kontrollflussebene)
    - Ausführung erfolgt verzögert zum Prolog
    - erledigt die eigentliche Arbeit ($\rightsquigarrow$ 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

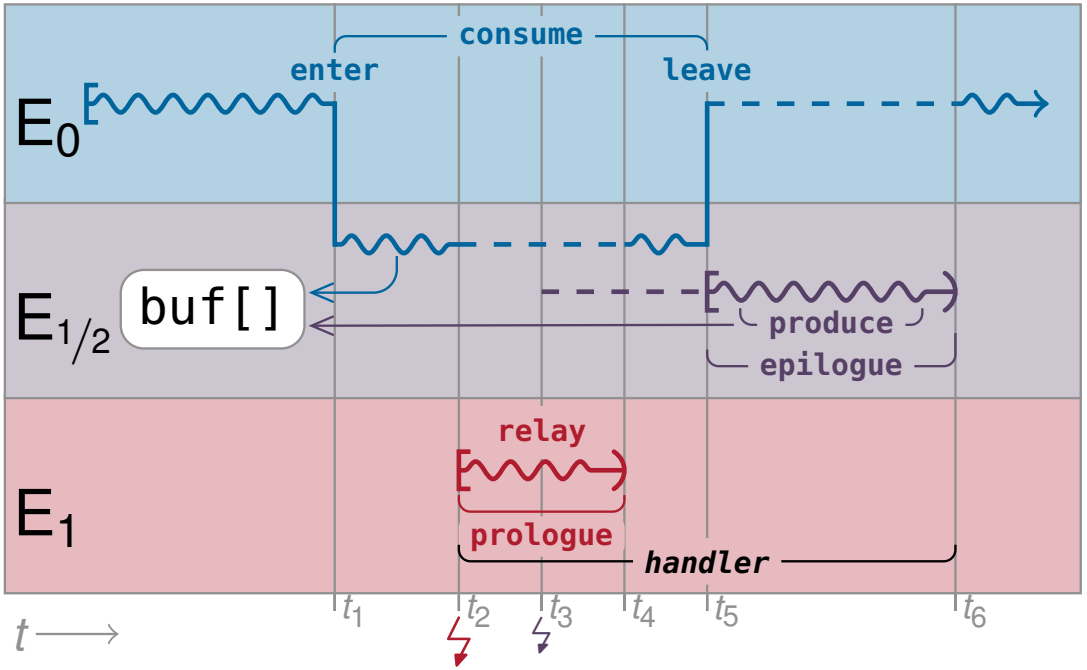
1. jederzeit unterbrochen durch Kontrollflüsse der Ebenen $E_{1\ldots n}$
   $\rightsquigarrow$ Prologe (Unterbrechungen) haben Priorität über Epiloge
2. nie unterbrochen durch Kontrollflüsse der Ebene E_0
   $\rightsquigarrow$ Epiloge haben Priorität über Anwendungskontrollflüsse
3. sequentialisiert mit anderen Kontrollflüssen von $E_{1/2}$
   $\rightsquigarrow$ Anhängige Epiloge werden nacheinander abgearbeitet.
   $\rightsquigarrow$ 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 IRQ-Leitung beim PIC

Prolog/Epilog-Modell – Ablaufbeispiel



E_1-Unterbrechungen werden nie gesperrt.

Aktivierungslatenz der Unterbrechungsbehandlung ist minimal.

- t_1 Anwendungskontrollfluss betritt Epilogebene $E_{1/2}$ (`enter()`).
- t_2 Unterbrechung ↯ auf Ebene E_1 wird signalisiert ⇝ 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 Epilogebene $E_{1/2}$ (`leave()`)
  ⇝ zwischenzeitlich aufgelaufene Epiloge werden nun abgearbeitet.
- t_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 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

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 IRQ-Leitung beim PIC
- Außerdem Mechanismen, um
  4. anhängige Epiloge zu „merken“: `queue` (z. B.)
     - entspricht dem IRR (Interrupt-Request-Register) beim PIC
  5. sicherzustellen, dass anhängige Epiloge abgearbeitet werden
     - entspricht bei der harten Synchronisation dem Protokoll zwischen CPU und PIC

Dieser Punkt muss etwas genauer betrachtet werden!

Prolog/Epilog-Modell – Implementierung

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

Wann müssen anhängige Epiloge abgearbeitet werden?

Immer unmittelbar, bevor die CPU auf E_0 zurückkehrt!

1. bei explizitem Verlassen der Epilogebene mit `leave()`
   - während der Anwendungskrontrollfluss auf $E_{1/2}$ 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 $E_{1\ldots n}$-Kontrollflüssen könnten Epiloge aufgelaufen sein (↩ Priorisierung).

Prolog/Epilog-Modell – Implementierung

- Implementierungsvarianten
  - rein softwarebasiert (⇝ **Übung**)
  - mit Hardwareunterstützung durch einen AST (⇝ [2, 3])
- Ein AST (*asynchronous system trap*) ist eine Unterbrechung, die (nur) durch Software angefordert werden kann.
  - z. B. durch Setzen eines Bits in einem bestimmten Register
  - ansonsten technisch vergleichbar mit einer Hardware-Unterbrechung
    - AST wird (im Gegensatz zu Traps/Exceptions) asynchron abgearbeitet
    - AST läuft auf eigener Unterbrechungsebene zwischen der Anwendungsebene und den Hardware-UBs (↪ unsere $E_{1/2}$)
    - Gesetzmäßigkeiten des Ebenenmodels 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_0 zurückkehrt
  - bleibt nur noch die Verwaltung der anhängigen Epiloge

Prolog/Epilog-Modell – Implementierung

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

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

Prolog/Epilog-Modell – Implementierung

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

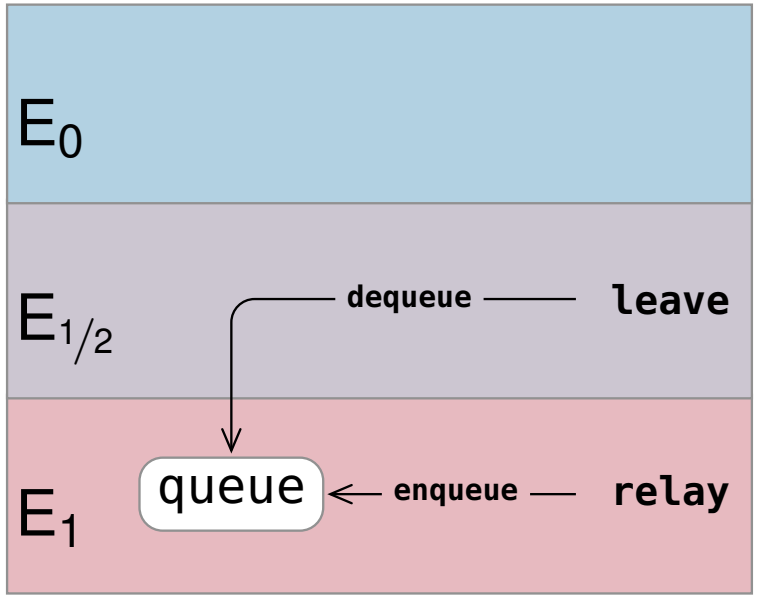
```
void enter() {
  CPU::setIRQL(1);          // b
}
void leave() {
  CPU::setIRQL(0);          // e
}                           // A
void relay(<Epilog>) {
  <haenge Epilog an queue an>
  CPU_SRC1::trigger();      // aktiviere Level-1 IRQ (AST)
}
void __attribute__((interrupt_handler)) irq1Handler()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

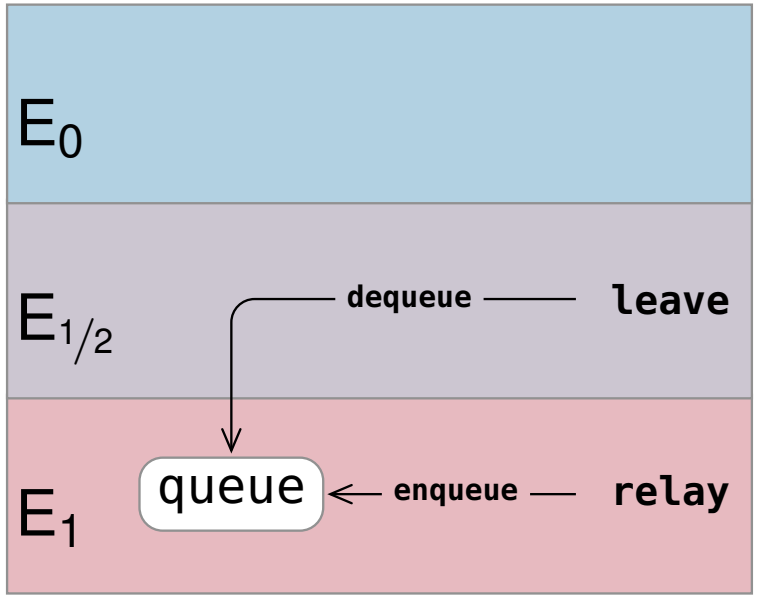
Prolog/Epilog-Modell – Ziel erreicht?

- 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



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. [7]) wäre natürlich schöner!

Prolog/Epilog-Modell: Bewertung

Vorteile

- Konsistenz ist sichergestellt
  (durch Synchronisation auf Epilogebene)
- 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 Epilogebene 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 (Forts.)

Fazit

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

Agenda

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

Nachtrag: Mehrkernsysteme

Beachte: Unterbrechungsbehandlung $\neq$ Parallelität

- Techniken funktionieren (so) nur bei echter Unterbrechungssemantik: 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” $\mapsto$ zweiseitig blockierend, z. B. mit *Spin-Locks* $\rightsquigarrow$ Übung
  - „weich” $\mapsto$ algorithmisch nichtblockierend (**schwer!**) $\rightsquigarrow$ [CS]