— BS //—

Unterbrechungstransparenz

Betriebssysteme, ©Wolfgang Schröder-Preikschat

Überblick

Sperren von Interrupts

- „harte Synchronisation“ durch sperren von Interrupts wirft Probleme auf:

  Blockade der Sperroperation muss eine Freigabeoperation folgen:

  z.B. { `cli` / `or #0x0700,sr` } gefolgt von { `sti` / `and #0xf8ff,sr` }

  Latenz während der Sperre kann auf externe Ereignisse nicht reagiert werden:
  - die Latenz ist abhängig von der Länge der kritischen Bereiche
  - der längste kritische Bereich bestimmt die Latenz des schlimmsten Falls
  - ein Prozess muss den kritischen Bereich in begrenzter Zeit verlassen

- Interrupts zu sperren ist zwar einfach, jedoch nicht immer das adäquate Mittel

Lösungsweg zum Blockadeproblem

Systemprogrammiersprachen könnten spezielle Sprachkonstrukte aufweisen

- kritische Bereiche als zu synchronisierende Programmstrukturen auszeichnen
  - Blöcke, Funktionen/Prozeduren, Moduln, Objekte
- der Übersetzer würde die erforderlichen Ebene $_{3/2}$ Anweisungen erzeugen

Zweckentfremdung von Konstrukten herkömmlicher Programmiersprachen

- Synchronisation auf Basis von C++ Konstruktoren/Destruktoren [7]
  - der Konstruktor sperrt, der Destruktor entsperrt die Interrupts
  - Destruktoren werden automatisch beim Verlassen des Blocks ausgeführt
- auch als *scoped locking* [5] bezeichnetes Synchronisationsmuster

Lösungsweg zum Latenzproblem

strukturelle Maßnahmen „harte Synchronisation“ nur *eines* kleinen Bereichs

- die Unterbrechungsbehandlung in zwei Phasen aufteilen: Prolog und Epilog
  - abgebildet auf Strukturelemente der verwendeten Programmiersprache
  - repräsentiert durch Funktionen/Prozeduren, Moduln, Objekte
- die Phasen lose miteinander über eine dynamische Datenstruktur koppeln
  - ein kritisicher Datenbestand, mit kritischen Verwaltungsoperationen
  - als FIFO-Warteschlange in (kurzer) deterministischer Zeit verwaltbar
- kritisch ist das Einfügen und Entfernen der Elemente der Warteschlange

algorithmische Maßnahmen „weiche Synchronisation“ eben dieses Bereiches

Prolog vs. Epilog

Prolog wird asynchron zum aktiven Prozess ausgeführt

- reagiert „prompt“ auf eine asynchrone Programmunterbrechung
- läuft auf Unterbrechungsebene $i,\ 0 < i \leq max$ (IRQ $\leq i$ ist gesperrt)
- besitzt nur sehr eingeschränkte Rechte zur Nutzung von Systemfunktionen
- beauftragt ggf. (s)einen Epilog mit der weiteren Ereignisverarbeitung

Epilog wird synchron zum aktiven Prozess ausgeführt

- reagiert „verzögert“ auf eine asynchrone Programmunterbrechung
- läuft auf Unterbrechungsebene 0 (alle Interrupts sind freigegeben)
- besitzt sehr umfassende Rechte zur Nutzung von Systemfunktionen
- beauftragt ggf. eine Prozessinkarnation mit der weiteren Ereignisverarbeitung

Verwandte Konzepte

top/bottom half Unix [3]; Aktivitäten der *bottom-half* sind asynchron zu den Aktivitäten der *top-half* und dürfen keine Systemfunktionen aufrufen.

deferred procedures NT [1]; Routinen, die eine nachgeschobene Ereignisverarbeitung durchführen und verzögert ausgeführt werden.

minor/major section Mars [2]; Unterbrechungsbehandlungsroutinen, die Systemaufrufe „aufhängen“ könnten (*minor*) bzw. die verzögert werden (*major*).

ISR category 1 and 2 OSEK [4]; Behandlungsroutinen ohne (1) und mit (2) Rechten zur Ausführung von Systemfunktionen.

Prolog/Epilog PEACE [7], PURE [6] — verbessert in BS // :-)

Randbedingungen

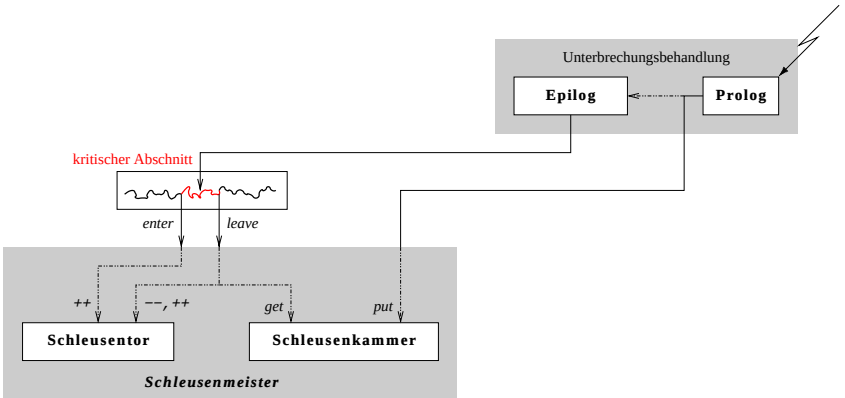
- der „normale Ablaufpfad“ soll mit minimalem Mahraufwand konfrontiert sein
  - Mehraufwand soll derjenige haben, der das Nebenläufigkeitsproblem bedingt
- es soll keine spezielle Betriebssystemarchitektur vorausgesetzt werden
  - Systemaufrufe sollen nicht zwingend nur über Traps abrufbar sein
  - die Verfahren müssen anwendbar sein innerhalb des selben Adressraums
  - sie müssen anwendbar sein im Systemmodus *und* im Benutzermodus
- Sperrzeiten für Interrupts sind zu minimieren — am besten, es fallen keine an

Synchronisation durch Schleusen

Idee Durch Interrupts hervorgerufene nebenläufige Aktivitäten werden serialisiert, genauer: die als Epiloge „verpackten“ Programmsequenzen.

- Betreten und Verlassen eines kritischen Abschnitts wird vermerkt
  - Zähler als „Schleusentor“: $\left.\begin{array}{l} tor = 0 \\ tor > 0 \end{array}\right\}$ kritischer Abschnitt ist $\left\{\begin{array}{l} \text{frei} \\ \text{belegt} \end{array}\right.$
- ein belegter/aktiver kritischer Abschnitt hält Epiloge von der Ausführung ab
  - *first-in, first-out* (FIFO) Warteschlange als „Schleusenkammer“
    - wird von den Prologen mit Epilogen aufgefüllt
    - wird beim Verlassen des kritischen Abschnitts abgearbeitet
  - jedoch nicht zwingend *first-come, first-serve* (FCFS) Verarbeitung **!**
- Zähler und Warteschlange werden von einem „Schleusenmeister“ verwaltet

Schleusenmodell (1)



Schleuse betreten

erledige *betrete einen kritischen Bereich* **:**
schließe das Schleusentor
basta

- schließen des Schleusentors bedeutet, den Zähler hochzuzählen $tor > 0$
  - minimaler Mehraufwand für den in die Schleuse „einlaufenden“ Prozess
- die Operation ist jedoch nur unkritisch im Uniprozessorfall [warum?]

Schleuse verlassen

erledige *verlasse einen kritischen Bereich* **:**
öffne das Schleusentor
wenn *die Schleusenkammer gefüllt ist*
und *das Schleusentor geöffnet ist* **dann**
arbeite anstehende Epiloge ab
basta

- öffnen des Schleusentors bedeutet, den Zähler runterzuzählen $tor \geq 0$
  - die Operation ist nur unkritisch im Uniprozessorfall
- in der Warteschlange ggf. aufgelaufene Epiloge sind abzuarbeiten [kritisch!]

Schleusen der Epiloge

erledige *arbeite anstehende Epiloge ab* **:**
  verrichte
    schließe das Schleusentor
    solange *die Schleusenkammer gefüllt ist* **verrichte**
      hole den nächsten Epilog aus der Schleusenkammer
      aktiviere den Epilog
    öffne das Schleusentor
  solange *die Schleusenkammer gefüllt ist*
basta

- das Leeren der Schleusenkammer ist ein kritischer Abschnitt [warum?]
- am Ende ist unbedingt auf „vergessene Epiloge“ zu prüfen [warum?]

Kritischer Abschnitt „anstehende Epiloge abarbeiten“

Prozesstermination Angenommen, der Prozess, der eben den nächsten Epilog der Schleusenkammer entnommen, ihn aber noch nicht aktiviert hat, würde verdrängt werden können:

- ein anderer laufender Prozess terminiert den verdrängten Prozess
- der entnommene Epilog kann nicht mehr aktiviert werden, er ginge verloren

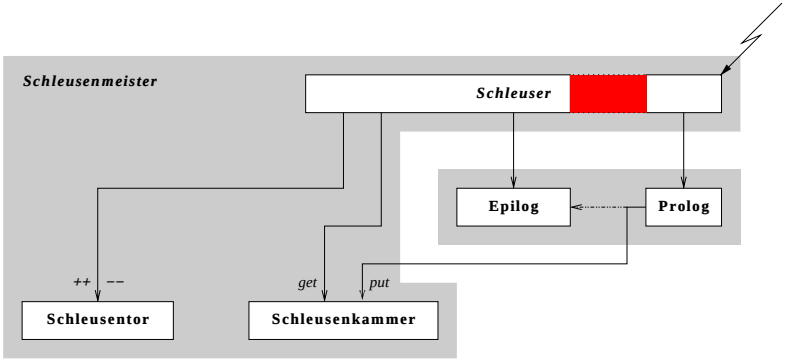
Unterbrechung Angenommen, nachdem festgestellt wurde, dass die Schleusenkammer leer ist und bevor das Schleusentor wieder geöffnet werden konnte trifft ein Interrupt ein, der einen weiteren Epilog ermöglicht (→ p. 16):

- der Epilog würde vergessen werden bis zum nächsten Schleusenvorgang
- ein nächster Schleusenvorgang kann aber nicht garantiert werden

Unterbrechungsbehandlung

- Prolog und Epilog zusammen machen die Unterbrechungsbehandlung aus
  - zuerst wird der Prolog gestartet, der dann ggf. einen Epilog anstößt
    - die Prologaktivierung erfolgt ohne Verzögerung *hardware interrupt*
    - die Epilogaktivierung wird ggf. verzögert *software interrupt*
  - die Aktivierung eines Epilogs unterliegt der „Schleusensynchronisation“
- nach Ausführung eines Prologs wird versucht, Epiloge zu propagieren
  - ein Prolog kann einen oder mehrere Epiloge ermöglichen
  - während der Ausführung eines Epilogs können weitere Epiloge auflaufen
- ¡mit Rückkehr zum unterbrochenen Programm ist die Schleusenkammer leer!

Schleusenmodell (2)



Epiloge ermöglichen

- die Ausführung eines Epilogs wird durch (s)einen Prolog eingeleitet:

> **erledige** *ermögliche einen Epilog* **:**
> *schiebe den Epilog in die Schleusenkammer*
> **basta**

- wann genau der Epilog zur Ausführung gebracht wird, ist nicht weiter definiert
  - der Synchronisationszustand entscheidet über den Ausführungszeitpunkt
- bevor ein Epilog erneut ermöglicht wird, muss dieser ausgeführt worden sein

Ermöglichte Epiloge abarbeiten (1)

> **erledige** *propagiere die Unterbrechungsbehandlung* **:**
> *aktiviere den Prolog*
> **wenn** *die Schleusenkammer gefüllt ist*
> **und** *das Schleusentor geöffnet ist* **dann**
> *arbeite anstehende Epiloge ab*
> **basta**

- entspricht dem Ablauf beim Verlassen eines kritischen Abschnitts (→ p. 11)
  - anstatt das Schleusentor zu öffnen, wird der Prolog aktiviert
- die nebenläufige Abarbeitung anstehender Epiloge ist zu verhindern [warum?]

Aktivzeit einer Unterbrechungsebene

- die Propagation erfolgt auf der gegenwärtig aktiven Unterbrechungsebene

  RISC unterstützen typischerweise nur zwei Ebenen:
  1. Ebene $_0$ (Benutzerebene) ermöglicht alle Interrupts
  2. Ebene $_1$ (Systemebene) sperrt alle Interrupts

  CISC sind bekannt dafür, mehr als zwei Ebenen zu unterstützen[1]
  - Ebene $_0$ (Benutzerebene) ermöglicht alle Interrupts
  - Ebene $_i$ (Systemebene), $i > 0$, sperrt Interrupts der Ebene $_j$, $0 < j \leq i$

- die **Latenzzeit** bis zur Aktivierung weiterer Prologe ist vergleichsweise lang

[1] Prozessoren der x86-Familie verhalten sich jedoch diesbezüglich wie RISC. Typische CISC-Vertreter in dem hier zu betrachtenden Fall sind z.B. Prozessoren der m68k-Familie.

Unterbrechungstransparente Propagation

Prologausführung erfolgt auf der gegenwärtig aktiven Unterbrechungsebene √

- die Ebene wird durch die *Hardware* aktiviert

Epilogausführung erfolgt auf Unterbrechungsebene $_0$ **!**

- die Ebene wird durch die *Software* aktiviert
  - alle Interrupts sind (wieder) zugelassen
  - die Latenzzeit bis zur Aktivierung weiterer Prologe ist gering
- die (Re-)Aktivierung dieser Ebene ist äußerst kritisch
  - der Prolog muss die Unterbrechung behandelt und „quittiert“ haben
  - ein *level-triggered* Interrupt muss vollständig behandelt worden sein
- den richtigen Zeitpunkt dafür zu bestimmen ist „lebenswichtig“ (→ p. 22)

Ermöglichte Epiloge abarbeiten (2)

erledige *propagiere die Unterbrechungsbehandlung* **:**
    aktiviere den Prolog
    wenn *die Schleusenkammer gefüllt ist*
      und *das Schleusentor geöffnet ist*
      und *keine Verschachtelung von Unterbrechungen vorliegt* **dann**
        arbeite anstehende Epiloge unterbrechungstransparent ab
basta

Unterbrechungstransparentes Schleusen der Epiloge

erledige *arbeite anstehende Epiloge unterbrechungstransparent ab* **:**
    verrichte
        schließe das Schleusentor
        sichere die Unterbrechungsebene und setze sie auf Null zurück
        solange *die Schleusenkammer gefüllt ist* **verrichte**
            hole den nächsten Epilog aus der Schleusenkammer
            aktiviere den Epilog
        stelle die Unterbrechungsebene wieder her
        öffne das Schleusentor
    solange *die Schleusenkammer gefüllt ist*
basta

Reaktivierungszeitpunkt für Unterbrechungsebene $_0$

- es darf keine weitere Verschachtelung von Unterbrechungen mehr vorliegen
  - d.h., das einzige noch unterbrochene Programm ist das Benutzerprogramm
- mit Beendigung der Unterbrechungsbehandlung wechselt der Arbeitsmodus
  - sofern die CPU überhaupt verschiedene Arbeitsmodi implementiert
  - logisch wird immer der nicht-privilegierte Benutzermodus reaktiviert
- die Feststellung des anstehenden Übergangs in den *user mode* ist CPU-abhängig
  - manche CPU hinterlässt Informationen dazu im Prozessorstatusregister
  - anderenfalls kann ein Zähler helfen, die nötige Information zu liefern

CPU-unterstützte Feststellung des Reaktivierungszeitpunkts

- das Prozessorstatusregister enthält Informationen zum aktiven Arbeitsmodus
  - es reicht ein Statusbit, das anzeigt, in welchem Modus die CPU operiert

Fallstudie m68k-Familie von Motorola:

- Bit $_{13}$ des Statusregisters ist das *supervisor/user state* Bit
  - Interrupts aktivieren den *supervisor state*, d.h. Bit $_{13} = 1$
  - zuvor wird eine Kopie des Statusregisters angelegt, die gestapelt wird[2]
  - das so gesicherte Bit $_{13}$ identifiziert den zu reaktivierenden Arbeitsmodus
- Bit $_{13} = 0$ zeigt den *user state* an: alle Interrupts sind zugelassen

[2] Bit $_{13}$ zeigt auch an, welcher Stapelspeicher/-zeiger gegenwärtig aktiv ist. Die Statussicherung bei der Ausnahmebehandlung erfolgt im *supervisor state* und muss daher den *supervisor state* Stapel verwenden. Je nach Bit $_{12}$ kann dies ab dem MC68020 der *master* (Bit $_{12} = 1$) oder *interrupt stack* (Bit $_{12} = 0$) sein.

Zählerbasierte Feststellung des Reaktivierungszeitpunkts

```
interrupt:
    incl _inca
    ...
    decl _inca
    iret
```

- zwei entscheidende Voraussetzungen müssen erfüllt sein:
  1. `incl` und `decl` sind unteilbar
  2. kein Interrupt verzögert die Ausführung von `incl`
- ist abhängig von der CPU und ihrem Interrupt-Modell

- im RISC-Fall und z.B. im CISC-Fall x86 wären die Voraussetzungen erfüllt
  - zum fraglichen Zeitpunkt sind alle Interrupts gesperrt[3]
- besonders problematisch dabei bleiben jedoch die *level-triggered* Interrupts

[3] `incl` und `decl` könnten demnach auch ruhig teilbar sein.

Intermezzo

Oder nimm eine „Gemeinschaftskarte“ :-)

Gehe in das „Gefängniss“! Begib Dich direkt dorthin. → p. 29
Gehe nicht über „Los“. Ziehe nicht DM 4000,- ein. pp. 26–28

Level-Triggered Interrupts

- das Interrupt-Signal liegt solange an, bis es vom Gerät zurückgenommen wird
  - der Gerätetreiber (genauer: sein Prolog) veranlasst dies beim Gerät
  - zu frühes reaktivieren von Unterbrechungsebene $_0$ führt zur Rekursion
  - eine saubere Abbruchbedingung zu garantieren, ist nicht (immer) möglich
- eine Unterbrechung zwischen „`interrupt:`“ und „`incl _inca`“ wäre kritisch
  - der geschachtelte Interrupt würde nicht rechtzeitig gezählt werden
  - der Zählerwert 1 entspräche dann nicht der Unterbrechungsebene $_1$
  - aktivieren der Unterbrechungsebene $_0$ löst erneut den ersten Interrupt aus
  - durch Software hervorgerufene *spurious interrupts* sind die Folge
- theoretisch betrachtet kann eine nicht endende Rekursion die Folge sein

Level- vs. *Edge-Triggered Interrupts*

- mit *edge-triggered* Interrupts bestehen diese Probleme nicht:
  - das Interrupt-Signal liegt nur für eine sehr kurze Zeit (einen Takt) an
  - die CPU prüft das Signal mit der nächsten (steigenden/fallenden) Taktflanke
  - ein Zurücksetzen der Unterbrechungsebene wäre damit unkritisch
- dafür werfen *edge-triggered* Interrupts andere Probleme auf:
  - Interrupts können verloren gehen, wenn die CPU „zu langsam ist“
  - „Rauschen“ (*noise*) kann Interrupts erzeugen, die keine sind
  - ebenso Spannungsschwankungen — *spurious interrupts* sind die Folge
- *level-triggered* Interrupts sind sicherer — und sie sind das „geringere Übel“

Spurious Interrupts

- „falsche“, „unechte“, „unberechtigte“ Interrupts gehören zum Alltag
  - mit ihnen ist jederzeit zu rechnen
  - ihr Auftreten darf nicht zum Systemabsturz führen
  - dabei ist es letztlich egal, wodurch sie ausgelöst worden sind
- Unterbrechungsbehandlung muss immer ein stückweit Fehlertoleranz zeigen
  - Interrupts, die keine sind, müssen ignoriert werden können
  - erst ihr gehäuftes Auftreten darf Anlass zu weiteren Schritten[4] geben
- robuste Treibersoftware muss Bestandteil eines jeden Betriebssystems sein

[4] Das System z.B. in einen *fail-stop*-Zustand bringen und automatisch „sauber herunterfahren“.

Da gibt es doch aber noch ein Problem (→ p. 21) . . .

Das Szenario ein Interrupt zwischen `solange` und `basta`, der einen Epilog ermöglicht und die Schleusenkammer gefüllt hinterlässt.

- Epiloge können vergessen werden, ihre Abarbeitung ist nicht gesichert
  - ein weiterer Interrupt müsste aufteten, um die Epiloge abzuarbeiten **!**
  - ein kritischer Bereich müsste betreten und wieder verlassen werden **!**
- das Problem besteht nur bei verschachtelten Unterbrechungen [warum?]
  - das Unterbrechungsmodell der m68k-Familie kann es z.B. hervorrufen
  - RISC und die x86-Familie sind dagegen „harmlos“, weil einfach gestrickt[5]
- ob es sich „katastrophal“ auswirkt, hängt von der Rechnerbetriebsart ab

[5] Sofern ein PIC (*programmable interrupt controller*) nicht zum Einsatz kommt.

. . . das jedoch in den Griff zu bekommen ist

Die Lösung entweder

1. vor Abfrage der Terminationsbedingung der Schleife alle Interrupts sperren, so auch die Schleife verlassen und zum unterbrochenen Programm zurückkehren, wenn die Schleusenkammer leer ist oder
2. wenn keine Unterbrechungsverschachtelung vorliegt, einen **AST** (*asynchronous system trap*) ermöglichen, anstatt direkt die Schleusenkammer zu leeren. Die AST-Behandlung leert dann die Schleusenkammer, d.h. bewirkt ein nachträgliches Verlassen eines „kritischen Abschnitts“.[6]

[6] Den AST zu ermöglichen, ohne auf Unterstützung durch die CPU bauen zu können, ist eine „haarige“ Angelegenheit. Es muss der *exception-stack frame* des unterbrochenen Programms (wie?) manipuliert werden.

Einen AST ermöglichen

erledige *propagiere die Unterbrechungsbehandlung* **:**
  aktiviere den Prolog
  wenn *die Schleusenkammer gefüllt ist*
  und *das Schleusentor geöffnet ist*
  und *keine Verschachtelung von Unterbrechungen vorliegt* **dann**
    ermögliche einen AST
basta

- die letzte Abfrage ist überflüssig, wenn die CPU einen AST unterstützt
  - ein AST wird von der CPU bei Rückkehr in den Benutzermodus ausgelöst
  - die CPU stellt also selbst sicher, dass keine Verschachtelungen vorliegen

Ermöglichte Epiloge als AST abarbeiten

erledige *AST* **:**
    arbeite anstehende Epiloge ab
basta

- der AST entspricht einem („normalen“) Systemaufruf zur Epilogabarbeitung
  - alle Interrupts sind zugelassen
  - ggf. ist dies erst explizit („von Hand“) zu ermöglichen, je nach CPU
- die ggf. gefüllte Schleusenkammer wird (nachträglich) geleert (→ p. 11)

„Schleusenkammer“ — Epilogwarteschlange

- Prologe hinterlassen ausführungsbereite Epiloge in einer Warteschlange
  - wenn die Nachbearbeitung einer Unterbrechungsbehandlung erfolgen soll
- der Zeitpunkt der Abarbeitung der Warteschlange ist bestimmt durch . . .
  1. die Länge/Dauer des unterbrochenen kritischen Bereiches, sofern zutreffend
     - beim Verlassen der Schleuse (→ p. 11) wird die Warteschlange abgebaut
  2. die Verschachtelungstiefe des Interrupts, d.h., den propagierenden Prolog
     - Unterbrechungsinkarnation $_1$ (→ p. 21)/AST baut die Warteschlange ab
- der „schlimmste Fall“ (*worst case*) der Abarbeitung muss bestimmbar sein

Synchronisation der Epilogwarteschlange

- die Epilogwarteschlange („Schleusenkammer“) ist ein kritischer Datenbestand

  sie wird { aufgebaut im Zuge der *Prologverarbeitung*
  { abgebaut beim Verlassen des { *kritischen Abschnitts*
  { *Interrupt-Modus*

- die Notwendigkeit zur Koordination der nebenläufigen Zugriffe ist offensichtlich
  - harte Synchronisation durch sperren der Interrupts ist zu vermeiden
  - nicht-blockierende Synchronisation ohne spezielle CPU-Elops ist das Ziel

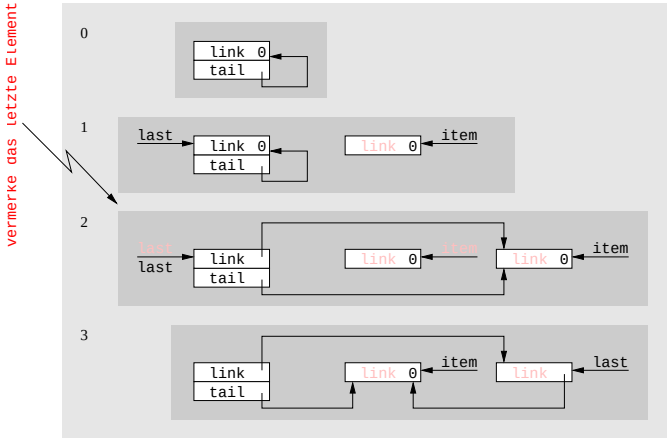
Unterbrechungsmuster

- die kritischen Bereich sind nur vor Nebenläufigkeit durch Interrupts zu schützen:
  1. der Interrupt trifft einen Prozess beim Abarbeiten der Epilogwarteschlange
     - ein `put()`$_{Prolog}$ überlappt `get()`$_{Epiloge\ abarbeiten}$
  2. es treten verschachtelte Interrupts auf
     - ein `put()`$_{Prolog}$ überlappt { `put()`$_{Prolog}$
     { `get()`$_{Epiloge\ abarbeiten}$
- ein `get()` unterbricht niemals sich selbst oder ein `put()` . . . . . **ausnutzen!**

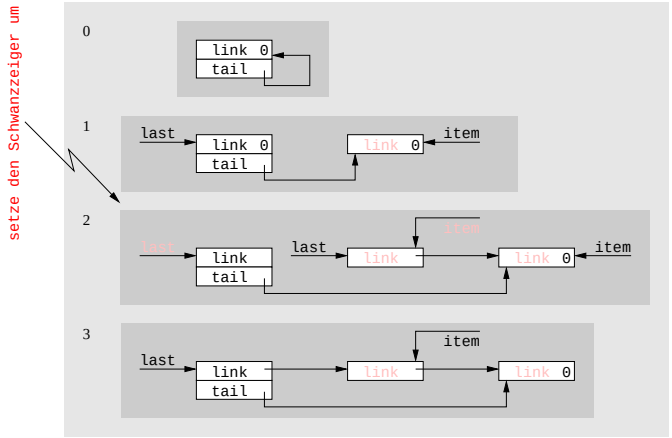
Unterbrechungstransparentes Einfügeverfahren

erledige *hänge das Element an die Warteschlange an* **:**
setzte den Verkettungszeiger des anzuhängenden Elements auf NIL
vermerke das letzte Element (d.h., lege eine Schwanzzeigerkopie an)
setzte den Schwanzzeiger auf das anzuhängende Element um
solange *dem letzten Element ein weiteres Element nachfolgt* **verrichte**
vermerke das nachfolgende Element als (vorerst) letztes Element
mache das anzuhängende Element zum Nachfolger des letzten Elements
basta

`put()` überlappt `put()` (1)



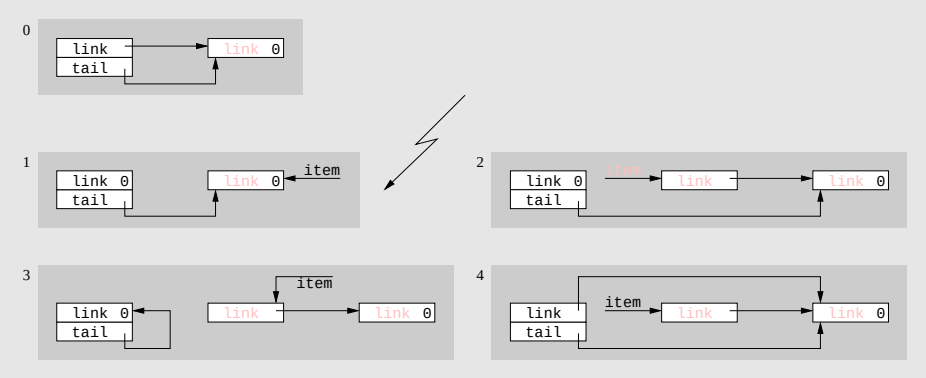
`put()` überlappt `put()` (2)



Unterbrechungstransparentes Entnahmeverfahren

erledige *entnehme der Warteschlange das nächste Element* **:**
ermittle das nächste Element über den Kopfzeiger
wenn *ein Element vorhanden ist* **dann**
setze den Kopfzeiger auf das nachfolgende Element um
wenn *ein nachfolgendes Element nicht existiert* **dann**
setze den Schwanzzeiger auf den Kopfzeiger um
wenn *das ermittelte Element (jetzt) einen Nachfolger hat* **dann**
hänge den/die Nachfolgerelemente in die Warteschlange ein
liefere *das ermittelte Element*
basta

`put()` überlappt `get()`



Determiniertheit

- der „**Aufwand des schlimmsten Falls**“ hängt ab von zwei Faktoren:
  1. dem Interrupt-Modell der verwendeten CPU # Unterbrechungsebenen
  2. der Implementierung von Gerätetreibern # ermöglichter Epiloge
- dieser *worst-case overhead* ist berechenbar und lässt sich wie folgt bestimmen:

$$WCO_{put} = O_{ins} + (N_{epi} - 1) * O_{ski}$$
$$WCO_{get} = O_{rem} + O_{pre} + N_{epi} * O_{req}$$

N_{epi}	max. Anzahl anstehender Epiloge
O_{ins}	einfügen eines einzelnen Epilogs
O_{ski}	überspringen eines Epilogs
O_{rem}	löschen des letzten Epilogs
O_{pre}	vorbereiten, Epiloge umzuhängen
O_{req}	umhängen eines einzelnen Epilogs

- alle Parameter sind statisch bestimmbar auf Basis einer Quelltextanalyse

Zusammenfassung

- Unterbrechungstransparenz bedeutet, Interrupts jederzeit zu ermöglichen
  - die Hardware gibt die jeweilige Unterbrechungsebene vor
  - die Software wird niemals eine höhere Unterbrechungsebene einlegen
- die Reaktionszeit auf externe Ereignisse ist nicht durch Software bestimmt
  - Interrupts werden nicht weiter als durch die Hardware vorgegeben gesperrt
  - durch Hardware ausgelöste Ereignisse sind in Software prompt bedienbar
- Entwurf und Implementierung unterliegen — wie überall — einem *tradeoff*
  - die Schleusensynchronisation, z.B., ist nicht in allen Fällen zweckmäßig
  - das Anwendungsszenario kann Interrupt-Sperren sehr wohl bevorzugen

Referenzen

[1] H. Custer. *Inside WINDOWS-NT*. Microsoft Press, 1993.

[2] A. Damm, J. Reisinger, W. Schwabl, and H. Kopetz. The Real-Time Operating System of MARS. *Operating Systems Review*, 23(3):141–157, 1989.

[3] J. Lions. “UNIX Operating System Source Code Level Six” and “A Commentary on the UNIX Operating System”. Technical report, Department of Computer Science, The University of New South Wales, Australia, 1977. Second Printing.

[4] OSEK/VDX Steering Committee. OSEK/VDX Operating System, Sept. 2001. Specification 2.2.

[5] D. C. Schmidt. Strategized Locking, Thread-safe Decorator, and Scoped Locking: Patterns and Idioms for Simplifying Multi-threaded C++ Components. *C++ Report*, 11(9), Sept. 1999.

[6] F. Schön, W. Schröder-Preikschat, O. Spinczyk, and U. Spinczyk. On Interrupt-Transparent Synchronization in an Embedded Object-Oriented Operating System. In *The Third IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2000)*, pages 270–277, Newport Beach, California, March 15–17, 2000. IEEE Computer Society. ISBN 0-7695-0607-0.

[7] W. Schröder-Preikschat. *The Logical Design of Parallel Operating Systems*. Prentice Hall International, 1994. ISBN 0-13-183369-3.