Fallstudie: Warteschlangenmanipulation

NBS — minimale Erweiterung (Forts.)

```
extern void nbs_reset (queue_t *);              /* clear queue */
extern void nbs_aback (queue_t *, chain_t *);   /* append chain item */
extern chain_t *nbs_fetch (queue_t *);          /* remove chain item */
```

```
void nbs_reset (queue_t *this) {
    fad_reset(this);
}
```

```
void nbs_aback (queue_t *, chain_t *) {
    /* ... */
}
```

```
chain_t *nbs_fetch (queue_t *this) {
    chain_t *node;
    /* ... */
    return node;
}
```

NBS $\Longleftrightarrow$ FAD/ITS

reset Wiederverwendung*
- *chain_t*
- *queue_t*

aback Ersetzung

fetch Ersetzung

Beachte *

- ggf. nicht, sollte das ABA-Problem bestehen

Fallstudie: Warteschlangenmanipulation (Forts.)

Szenario 1: *aback* toleriert Überlappungen nur von sich selbst

```
void nbs_aback (queue_t *this, chain_t *item) {
    chain_t *last;

    item->link = 0;

    do last = this->tail;
    while (!CAS(&this->tail, last, &item->link));

    last->link = item;
}
```

Plausibilitätskontrolle $\hookrightarrow$ kritisch ist *tail*, der Listenfuß

- die Kopie des aktuellen Wertes von *tail* wird in *last* vermerkt
- neuer Wert von *tail* ergibt sich aus *&LINK(item)*
- CAS gelingt $\Longleftrightarrow$ *tail* hält den alten Wert $\Rightarrow$ $tail = \&LINK(item)$
- *do . . . while* terminiert $\Longleftrightarrow$ CAS gelingt

Fallstudie: Warteschlangenmanipulation (Forts.)

Szenario 2: *fetch* toleriert Überlappungen nur von sich selbst

```
chain_t *nbs_fetch (queue_t *this) {
    chain_t *node;

    do if ((node = this->head.link) == 0) return 0;
    while (!CAS(&this->head.link, node, node->link));

    if (node->link == 0)
        this->tail = &this->head;

    return node;
}
```

Plausibilitätskontrolle $\hookrightarrow$ kritisch ist *LINK(head)*, der Listenkopf

- analog zu *aback*, jedoch Sonderbehandlung wenn gilt:
  - $LINK(head) = node \wedge tail = \&LINK(node) \Rightarrow$ einziges Element
- im Falle des einzigen Elements gilt nach weiteren Verlauf:
  - $LINK(head) = LINK(node) = 0 \Rightarrow tail = \&LINK(head)$!!!

Fallstudie: Warteschlangenmanipulation (Forts.)

Szenario 3: *fetch* toleriert Überlappungen von sich selbst und (vermeintlich) von *aback*

```
chain_t *nbs_fetch (queue_t *this) {
    chain_t *node;

    do if ((node = this->head.link) == 0) return 0;
    while (!CAS(&this->head.link, node, node->link));

    if (node->link == 0)
        CAS(&this->tail, &node->link, &this->head);

    return node;
}
```

Beachte $\hookrightarrow$ hier gilt: $LINK(node) = 0 \Rightarrow LINK(head) = 0$

- überlappendes *aback* hat ggf. *item* angehängt: $tail = \&LINK(item)$
- dann müsste jedoch auch gelten: $LINK(head) = item$

$\Rightarrow$ Widerspruch: *fetch* toleriert überlappendes *aback* nicht

Fallstudie: Warteschlangenmanipulation (Forts.)

Rekapitulation der Szenarien 1 – 3

aback||*aback*
- Integrität von *tail* ist sichergestellt

fetch||*fetch*
- Integrität von *LINK(head)* ist sichergestellt

fetch||*aback*
- Integrität von *LINK(head)* bzw. *tail* ist sichergestellt
- Integrität von beiden zusammen ist nicht sichergestellt

aback||*fetch*
- Integrität $\sim$ *fetch*||*aback* $\Rightarrow$ nicht sichergestellt

Neuralgischer Punkt $\hookrightarrow$ Kopfelement wird zum Verkettungsglied

aback
- hängt *item* ggf. an ein Kopfelement (*node* = *last*) an, das ggf. schon nicht mehr auf der Liste steht
- muss die Aktualisierung von *LINK(last)* in Abhängigkeit vom Ausführungsverlauf von *fetch* vornehmen

fetch
- setzt *tail* ggf. auf *&LINK(head)*, obwohl ggf. ein weiteres (zweites) Element an die Liste angehängt wurde
- muss das Zurücksetzen des Fußzeigers (*tail*) in Abhängigkeit vom Ausführungsverlauf von *aback* vornehmen

Fallstudie: Warteschlangenmanipulation (Forts.)

Lösungsansatz: Problemspezifisches Protokoll zwischen *aback* und *fetch*

Idee ist es, den Verkettungszeiger (*link*) eines Listenelements auch zur „Signalisierung“ zwischen *aback* und *fetch* zu nutzen

- sei *that* Zeiger (*chain_t*$*$) auf ein Listenelement, dann soll gelten:

$$LINK(that) = \begin{cases} that & that \text{ ist gültiges Verkettungsglied} \\ 0 & \text{Verkettungsglied } that \text{ wurde entfernt} \\ \ldots & \text{Listenelement } that \text{ mit Nachfolger} \end{cases}$$

- für die beiden Funktionen lässt sich dies dann wie folgt ausnutzen:

  aback
  - hängt das neue Element nicht an $\iff LINK(that) \neq that$
  - beachte: $that \mapsto last \Rightarrow last$ ist Verkettungsglied

  fetch
  - nullt *LINK(that)* $\iff LINK(that) = that$, und
  - versucht *tail* zurückzusetzen $\iff LINK(that) = 0$
  - beachte: $that \mapsto node \Rightarrow node$ ist Verkettungsglied

- beachte: *last* = *node*, beide Funkt. sehen dasselbe Verkettungsglied

Fallstudie: Warteschlangenmanipulation (Forts.)

Korrektur des Kopfzeigers

Aufmerksamkeit ist noch dem Kopfzeiger (*LINK(head)*) zu geben, dessen Integrität im Konfliktfall (*last* = *node*) wieder herzustellen ist

fetch||*aback*
- *fetch* stellt fest, dass *node* das einzige Element war
- dann gilt $LINK(head) = 0 \wedge LINK(node) = node$
- zuvor jedoch hängt *aback* noch ein neues Element an
- dann gilt $LINK(node) \neq node \Rightarrow$ wird nicht genullt
- *LINK*(*node*) zeigt auf das soeben angehängte Element

$\Rightarrow$ *fetch*: *LINK*(*head*) ist auf *LINK*(*node*) zu korrigieren

aback||*fetch*
- *last* zeigt auf das Verkettungsglied, *tail* ist umgesetzt
- *aback* stellt fest, dass *last* soeben entfernt wurde
- dann gilt $LINK(last) = 0 \Rightarrow$ *item* hängt nicht an
- *fetch* setzt *tail* in der Situation jedoch nicht zurück
- es gilt: $tail = \&LINK(item) \wedge LINK(head) = 0$

$\Rightarrow$ *aback*: *LINK*(*head*) ist auf *item* zu korrigieren

Fallstudie: Warteschlangenmanipulation (Forts.)

Zusammenfügen: Wettlauftolerantes *aback*

```
void nbs_aback (queue_t *this, chain_t *item) {
    chain_t *last, *self;

    item->link = item;    /* item becomes new tail resp. next last */

    do self = (last = this->tail)->link;  /* draw copies as needed */
    while (!CAS(&this->tail, last, &item->link));

    if (!CAS(&last->link, self, item))    /* last removed by fetch */
        this->head.link = item;           /* item becomes new head */
}
```

Beachte $\hookrightarrow$ Problem ABA (Selbststudium)

- ggf. ist *item* ein *chain_p*$*$ und *wheel* entsprechend zu applizieren
- ebenso wäre dann *index* auf allen relevanten Stellen anzuwenden

Fallstudie: Warteschlangenmanipulation (Forts.)

Zusammenfügen: Wettlauftolerantes *fetch*

```
chain_t *nbs_fetch (queue_t *this) {
    chain_t *node, *next;

    do if ((node = this->head.link) == 0) return 0;
    while (!CAS(&this->head.link, node,
                ((next = node->link) == node ? 0 : next)));

    if (next == node) {    /* last element just removed, be careful */
        if (!CAS(&node->link, next, 0)) this->head.link = node->link;
        else CAS(&this->tail, &node->link, &this->head);
    }

    return node;
}
```

Beachte ↪ Problem ABA (Selbststudium)

- ggf. ist *node* ein *chain_p∗* und *index* entsprechend zu applizieren

Fallstudie: Warteschlangenmanipulation (Forts.)

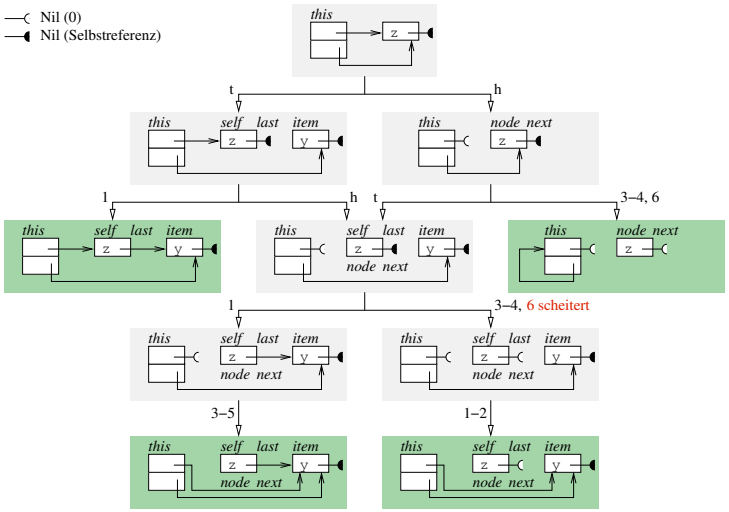
Plausibilitätskontrolle: Neuralgische Punkte

```
void nbs_aback (queue_t *this, chain_t *item) {
    chain_t *last, *self;

    item->link = item;

t   do self = (last = this->tail)->link;
t   while (!CAS(&this->tail, last, &item->link));

1   if (!CAS(&last->link, self, item))
2       this->head.link = item;
}

chain_t *nbs_fetch (queue_t *this) {
    chain_t *node, *next;

h   do if ((node = this->head.link) == 0) return 0;
h   while (!CAS(&this->head.link, node,
h             ((next = node->link) == node ? 0 : next)));

3   if (next == node) {
4       if (!CAS(&node->link, next, 0))
5           this->head.link = node->link;
6       else CAS(&this->tail, &node->link, &this->head);
    }

    return node;
}
```

- t ▶ *tail* weitersetzen ✓
- h ▶ *head* weitersetzen ✓
- 1 ▶ Verkettungsglied gültig?
  - ▶ ja, *item* angehängt
  - ▶ *fetch* wurde signalisiert
- 2 ▶ nein, *fetch* kam vorbei
  - ▶ *head* korrigieren
- 3 ▶ letztes Element raus?
- 4 ▶ wirklich keins mehr da?
  - ▶ ja, *aback* signalisiert: 6
- 5 ▶ nein, *aback* kam vorbei
  - ▶ *head* korrigieren
- 6 ▶ *tail* ggf. zurücksetzen

Fallstudie: Warteschlangenmanipulation (Forts.)

Plausibilitätskontrolle: Datenstrukturentwicklung je nach Überlappungsfall



Nichtsequentielle Programme sind nicht einfach. . .

Rekapitulation am Beispiel wichtiger, noch nicht behandelter Aspekte:

Ausführungspfadanalyse
- ▶ welcher Aufwand trifft den *Normalfall*?
- ▶ welcher Zusatzaufwand den *Ausnahmefall*?
- ▶ wann „lohnt“ sich welches Verfahren?

Fortschrittsgarantie
- ▶ ist das Vorankommen nichtsequentieller Programmausführung sichergestellt?

Anmerkung
- ▶ die Untersuchungen sind nicht exakt: keine Taktzyklen/Laufzeiten
- ▶ um ein Gefühl zu bekommen reicht es, Maschinenbefehle zu zählen
- ▶ auch soll es genügen, sich Umgebungseinflüsse vor Augen zu halten

Ausführungspfadanalyse

Referenz zum Vergleich mit NBS: FAD mit „annotierten“ kritischen Abschnitten

```
void fad_aback (queue_t *this, chain_t *item) {
    item->link = 0;

    ENTER(fad);
    this->tail->link = item;
    this->tail = item;
    LEAVE(fad);
}
```

Schutzoptionen

- nil unsynchronisiert
- ice Fortsetzungssperre
- irq Unterbrechungssperre

```
chain_t *fad_fetch (queue_t *this) {
    chain_t *item;

    ENTER(fad);
    if ((item = this->head.link) && !(this->head.link = item->link))
        this->tail = &this->head;
    LEAVE(fad);

    return item;
}
```


Ausführungspfadanalyse (Forts.)

Anzahl der Maschinenbefehle (x86): gesamt/davon im kritischen Abschnitt

	aback		*fetch*	
Konfiguration	-	—	-	—
unsynchronisert	7/0	7/0	10/0	11/0
Unterbrechungssperre	9/4	9/4	12/8	14/10
Fortsetzungssperre	$12/4+\delta_1$	$12/4+\delta_1$	$19/8+\delta_1$	$21/10+\delta_1$
NBS	$24/0+\delta_2$	$25/0+\delta_2$	$25/0+\delta_2$	$35/0+\delta_2$

- kürzester Pfad, — längster Pfad

δ_1 Aufwand für die Abarbeitung zurückgestellter Fortsetzungen: *clear*

- 2 Befehle für die Aufrufsequenz
- 12 Befehle Verwaltungsgemeinkosten
- 6 Befehle für Entnahme und Aktivierung *einer* Fortsetzung
  - Entnahme: plus 10/11 Befehle des unsynchronisierten *fetch*

δ_2 Aufwand für die Wiederholungsversuche bei gescheitertem CAS

- 8/14 Befehle pro Versuch bei *aback*/*fetch*

Fortschrittsgarantie

Aussagen zur Lebendigkeit (engl. *liveliness*)[7] von Programmen zu treffen, die nichtsequentiell ablaufen, braucht Umgebungswissen

- alle hier betrachteten Fälle zeigten nichtsequentielle Programme
- in ihnen spiegeln sich Annahmen zu Überlappungsmustern wider
- die durch das Operationsprinzip des Rechensystems begründet sind
  - Unterbrecher- und Uni- bzw. Multiprozessorbetrieb

Ansätze zum Nachweis von Lebendigkeitseigenschaften nichtsequentieller Programme gibt es bereits seit geraumer Zeit

> *There is a rather large body of sad experience to indicate that a concurrent program can withstand very careful scrutiny without revealing its errors. [10, S. 456]*

- formale Methoden sind eigentlich der einzig richtige Weg zum Ziel
  - wären Betriebssysteme (universal/spezial) nur nicht so komplex

[7] In der Informatik (fälschlicherweise) auch: *liveness*.

Unterbrechungs-/Fortsetzungssperren

Fortschrittsgarantie für die Weiterleitung zurückgestellter Arbeitsaufträge (d.h., Nachspänne oder Fortsetzungen/Einfädelungen)

- leitet sich ab aus der WCET des gesperrten kritischen Abschnitts, für den eine Eintrittsanforderung vorliegt[8]
- ergibt sich aus der Annahme, dass die diesen Abschnitt ausmachende Anweisungsfolge des nichtsequentiellen Programms terminiert

Beachte $\hookrightarrow$ δ_1 verlängert die Laufzeit serialisierter KA nicht

- die betrachteten Sperrverfahren garantieren Fortschritt, da ihre Korrektheit nicht von der der kritischen Abschnitte abhängt
- die Sperrverfahren „benutzen“ die kritischen Abschnitte nicht

[8] Die in Kap. 5 vorgestellte Unterbrechungsweitergabe sowie die damit verbundene Nachspannfreistellung (*clear*) zeigt Merkmal eines kritischen Abschnitts, der nämlich zu einem Zeitpunkt nur einmal aktiv sein darf, um Stapelüberlauf vorzubeugen.

Unterbrechungstransparente Synchronisation

Fortschrittsgarantie für Warteschlangenoperationen in einer Umgebung, die ein rollenspezifisches bzw. -verteiltes Überlappungsmuster vorgibt:

aback, *fetch* } ist nur von Überlappung durch *aback* betroffen, ggf.

- die Ausführung von *aback* wird ereignisbedingt, mehrstufig ausgelöst
- und: nichtsequentielle Programmausführung im Uniprozessorbetrieb

Beachte ↪ Programmschleifen (S. 6-26/6-27)

- aus den Algorithmen allein heraus lässt sich keine WCET bestimmen
- sie ist erst durch Analyse des Unterbrecherbetriebs abschätzbar

Unterbrechungstransparente Synchronisation (Forts.)

Analyse des Unterbrecher- und Treiberbetriebs

IPL_{max}
- höchst mögliche Unterbrechungsebene der Hardware
- $0 < max < N$
- $0 \leadsto 1 \leadsto 2 \leadsto \cdots \leadsto N$ ist schlimmster Fall
- d.h., vertikal kompletter Durchlauf, von unten nach oben

IPL_{tip}
- Anzahl der Gerätetreiber einer Ebene $_{tip}$
- $0 \leq tip \leq IPL_{max}$
- $0 \leadsto 1 \leadsto 2 \leadsto \cdots \leadsto IPL_{tip}$ ist schlimmster Fall in Ebene $_{tip}$
- d.h., horizontal kompletter Durchlauf (Abfragebetrieb)

TIP_{dev}
- Nachspannauslösungen von Treiber $_{dev}$ pro Unterbrechung
- $0 \leq dev \leq M$: $M = 1$ bei Einfachauslösung, $\delta > 1$ sonst

TIP_{max}
- maximale Länge der Warteschlange von Nachspännen
- $\sum TIP_{dev}$ aller Treiber aller Unterbrechungsebenen

Beachte ↪ δ_1

- TIP_{max} ist auch untere Grenze für die Anzahl von Fortsetzungen

Unterbrechungstransparente Synchronisation (Forts.)

Obergrenze für die Anzahl der Durchläufe der kritischen Programmschleifen

Konfiguration	IPL_{max}	$\sum IPL_{tip}$	TIP_{dev}	TIP_{max}
Ø PC_{Linux}	15	17	1	17
≈ MacBook	15	20	1	20
AX*	7	7	1	7
$PEACE^*_{SUPRENUM}$	8	8	1	8
$PEACE_{MANNA}$	1	3	1	3
$PURE_{LKW}$	∼	10	1	10
$PURE_{Wetterstation}$	2	2	1	2
CiAO-AS	2	2	1	2
OOStuBS	15	2	1	2
MPStuBS	15	3	1	3
EZStubs	15	1	1	1

* Nicht jede Unterbrechungsebene hat Treiber, mindestens aber einen Abfänger

Nichtblockierende Synchronisation

Fortschrittsgarantie für Warteschlangenoperationen in einer Umgebung, die ein rollenunspezifisches Überlappungsmuster vorgibt:

- Operationsausführungen können sich beliebig überlappen
  - Wartestapel ↦ *ahead* und *strip*
  - Warteschlange ↦ *aback* und *fetch*
- nichtsequentielle Programmausführung: Uni-/Multiprozessorbetrieb

Beachte ↪ δ_2 und Programmschleifen (S. 6-37 bzw. 6-55/6-56)

- aus den Algorithmen allein heraus lässt sich keine WCET bestimmen
  - sie ist erst durch Analyse der Programmabläufe abschätzbar
- festgehalten werden kann, dass sich die Verfahren sperrfrei verhalten
  - die Ausführung des nichtsequentiellen Programms schreitet voran
  - allerdings lässt die Aushungerung von Fäden nicht ausschließen
- bei bekannter WCET aller Fadenabläufe wären sie sogar wartefrei

Nichtblockierende Synchronisation (Forts.)

Differenzierungen von Fortschrittsgarantien

behinderungsfrei (engl. *obstruction-free*)

- ein einzelner, in Isolation ablaufender Faden wird seine Operation in begrenzter Anzahl von Schritten beenden
- ein Faden läuft in Isolation ab, wenn alle ihn behindern könnenden anderen Fäden in der Ausführung zurückgestellt sind

sperrfrei (engl. *lock-free*), umfasst Behinderungsfreiheit

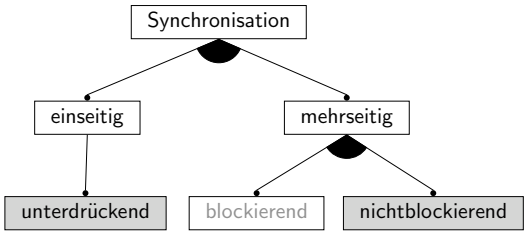
- jeder bei Ablauf eines Fadens auszuführende Schritt trägt dazu bei, dass das nichtsequentielle Programm insgesamt voranschreitet
- garantiert systemweiten Durchsatz, erlaubt jedoch Aushungerung

wartefrei (engl. *wait-free*), umfasst Sperrfreiheit

- die Anzahl der zur Beendigung einer Operation bei Fadenabläufen auszuführenden Schritte ist begrenzt
- garantiert systemweiten Durchsatz und ist frei von Aushungerung

Merkmaldiagramm

Synchronisation mit Elementaroperationen der Ebene 2

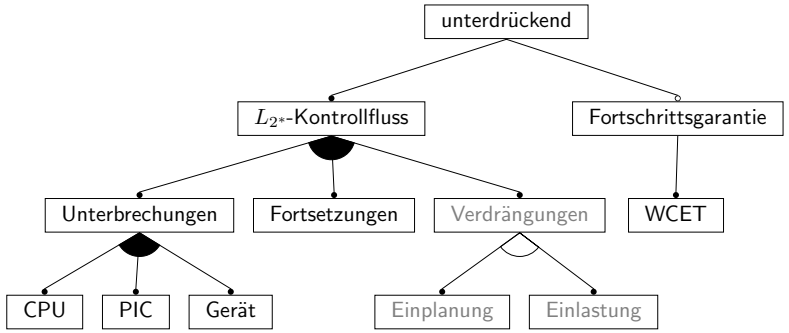


Beachte ↪ blockierende Synchronisation

- bezogen auf das Abstraktionsniveau der Befehlssatzebene heißt das:
  - (a) Schlossvariable (engl. *lock variable*) und
  - (b) aktives Warten (engl. *busy waiting*, *spin locking*)
- Blockadefreiheit ist damit grundsätzlich ausgeschlossen
  - weshalb Verfahren dazu auch nicht weiter untersucht wurden...

Merkmaldiagramm (Forts.)

Einseitiger Ausschluss der Ausführung eines Ebene 2-Programms

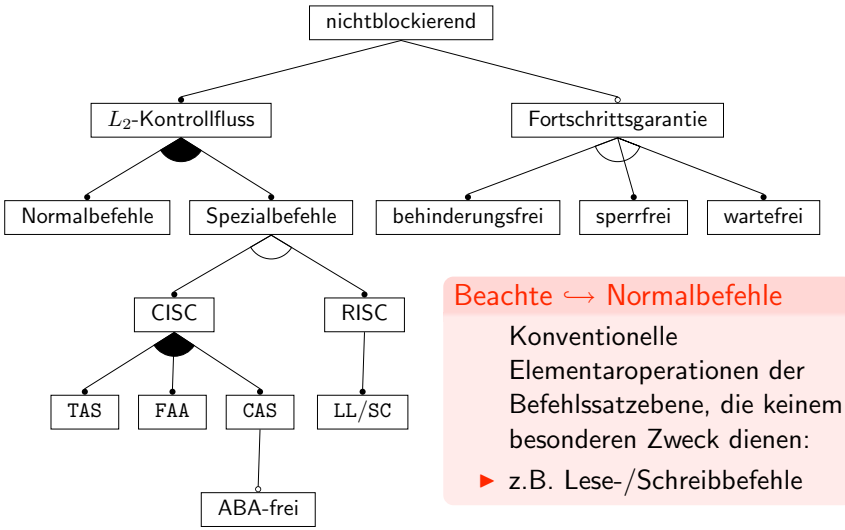


Beachte ↪ Verdrängungen

- diese „benutzen“ Abstraktionen zur Prozessverwaltung (z.B. Fäden)
- die auf Ebene der Ablaufsteuerung jedoch nicht bekannt sind
  - es sei denn, die Befehlssatzebene würde ein Prozesskonzept bieten

Merkmaldiagramm (Forts.)

Einseitiger Ausschluss der Ausführung eines Ebene 2-Befehls



Beachte ↪ Normalbefehle

Konventionelle Elementaroperationen der Befehlssatzebene, die keinem besonderen Zweck dienen:

- z.B. Lese-/Schreibbefehle

Resümee

Wettlaufintoleranz $\hookleftarrow$ Abwehr von Ereignisanforderungen

- einseitiger Ausschluss der Ausführung eines Ebene$_2$-Programms
- Unterbrechungssperre, total/partiell
- Fortsetzungssperre, Fortsetzungen/Einfädelungen

Wettlauftoleranz $\hookleftarrow$ nichtblockierende Synchronisation

- Spezialfall: Unterbrechungstransparente Synchronisation
- einseitiger Ausschluss der Ausführung eines Ebene$_2$-Befehls
- Einzelwort „*compare and swap*“
- Problem „ABA“, Etikettierung von Zeiger, Generationszähler

Fortschrittsgarantien $\hookleftarrow$ WCET

- Weiterleitung zurückgestellter Arbeitsaufträge
- Reparatur inkonsistent gewordener Datenstrukturen
- Behinderungs-, Sperr- und Wartefreiheit

Literaturverzeichnis

[1] Ralf Guido Herrtwich and Günter Hommel.
Kooperation und Konkurrenz — Nebenläufige, verteilte und echtzeitabhängige Programmsysteme.
Springer-Verlag, 1989.

[2] Wolfgang Schröder-Preikschat.
Softwaresysteme 1.
http://www4.informatik.uni-erlangen.de/Lehre/SOS1, 2004.

[3] Wolfgang Schröder-Preikschat.
Systemprogrammierung.
http://www4.informatik.uni-erlangen.de/Lehre/SP, 2008.

[4] Daniel Lohmann.
Betriebssysteme.
http://www4.informatik.uni-erlangen.de/Lehre/BS, 2007.

Literaturverzeichnis (Forts.)

[5] David D. Clark.
The structuring of systems using upcalls.
ACM SIGOPS Operating Systems Review, 19(5):171–180, 1985.

[6] Friedrich Schön, Wolfgang Schröder-Preikschat, Olaf Spinczyk, and Ute Spinczyk.
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)*, pages 270–277. IEEE Computer Society Press, March 2000.

[7] Theodore P. Baker.
Stack-based scheduling of realtime processes.
Real-Time Systems, 3(1):67–99, 1991.

Literaturverzeichnis (Forts.)

[8] LEO GmbH.
LEO Deutsch-Englisches Wörterbuch.
`http://dict.leo.org`.

[9] Barbara H. Liskov and Stephen N. Zilles.
Programming with abstract data types.
ACM SIGPLAN Notices, 9(4):50–59, April 1974.

[10] Susan Owicki and Leslie Lamport.
Proving liveness properties of concurrent programs.
ACM Transactions on Programming Languages and Systems (TOPLAS), 4(3):455–495, July 1982.