

Betriebssystemtechnik

Prozesssynchronisation: Wettlauftoleranz

Wolfgang Schröder-Preikschat

Lehrstuhl Informatik 4

18. Juli 2011

Toleranz gegenüber Wettlaufsituationen

Wettlaufsituation bezeichnet einen bestimmten Umstand gleichzeitiger Prozesse eines nichtsequentiellen (verteilten) Programms

- bei dem der zeitliche Ablauf von Elementaroperationen Auswirkung auf das Ergebnis einer Komplexoperation haben kann
- ein **kritischer Wettlauf**, auch (engl.) *race condition* oder *race hazard*
 - neutral/positiv (*condition*) oder negativ (*hazard*) belegt
 - je nachdem, welchen Effekt der Ausgang dieses Wettlaufs bewirkt
- eine Berechnung, die eine ggf. unerwartet **kritische Abhängigkeit** vom relativen Zeitverlauf von Ereignissen zeigt

Toleranz gegenüber dieser Situation meint, nichtsequentielle Programme so zu gestalten, dass gleichzeitige Prozesse jederzeit zugelassen sind

- nichtblockierende Synchronisation mit/ohne Spezialbefehlen
⇒ optimistische Nebenläufigkeitskontrolle

Synchronisationskonzepte: Befehlssatzebene [8]

Alleinstellungsmerkmal dieser Abstraktionsebene sind allgemein die in der **CPU** manifestierten Fähigkeiten eines Rechensystems, hier:

- (a) in Bezug auf die Bereitstellung von Spezialbefehlen und
- (b) hinsichtlich der Semantik dieser Befehle zur Prozessverarbeitung

Techniken zur Synchronisation gleichzeitiger Prozesse können demzufolge nur auf sehr einfache, elementare Konzepte zurückgreifen

zu (a) die Möglichkeit, externe/interne Prozesse aussperren zu können

- Unterbrechungssperre
- Umlaufsperre (*aktives Warten*) ✓

zu (b) die Möglichkeit, kritische Abschnitte so ausformulieren zu können, dass gleichzeitige Prozesse nicht ausgesperrt werden

- nichtblockierende Synchronisation
- unterbrechungstransparente Synchronisation (X)

*Toleranz ist der Verdacht,
dass der andere Recht hat.*



Quelle: Gedenktafel, Berlin, Bundesallee 79

Gliederung

1 Nichtblockierende Synchronisation

- Grundlagen
- Prinzip

2 Fallstudien

- Zählermanipulation
- Listenmanipulation
 - Stapel
 - Schlange

3 Eigenarten

- ABA
- Fehlversuche

4 Zusammenfassung

5 Anhang

Grundsätzliche Idee

Koordinierung sich einander ggf. überlappender Aktivitäten, ohne dabei **gleichzeitige Prozesse**¹ auszuschließen

- toleriert (pseudo-) parallele Programmausführungen
 - parallel** • Multiprozessor, wirkliche Parallelität
 - pseudoparallel** • Uniprozessor, Parallelität durch Unterbrechungen
- die Verfahren greifen auf **nichtprivilegierte Befehle** der ISA zurück
 - CISC** • TAS, FAA, CAS bzw. CMPXCHG
 - RISC** • LL/SC
 - sonst** • gewöhnliche Befehle zum Lesen/Schreiben von Speicherwörtern
- die Befehle funktionieren im Benutzer- wie auch im Systemmodus

Beachte

- kein gegenseitiger Ausschluss $\Rightarrow$ **Verklemmungsvorbeugung**
- die *benutzten* Befehle sind „echte“ Elementaroperationen der ISA

¹Prozesse, deren Ausführung sich zeitlich überschneidet.

Bedingte Wertzuweisung: *Compare and Swap*, CAS

Elementaroperation der Befehlssatzebene:

- unteilbar: Unterbrechungs- und Zugriffssperre (Datenbus)
- implementiert eine **Transaktion** für Uni- und Multiprozessoren
- gebräuchlich als Einzel- (CAS) und Doppelwortvariante (DCAS)
- eingeführt mit IBM System/370 [3, S. 123]

Ausführung mit einem atomaren „*read-modify-write*“-Zyklus:

- 1 Lesen eines Datums aus dem Arbeitsspeicher
- 2 bedingte Modifikation des gelesenen Datums
- 3 bedingtes Zurückschreiben des Datums in den Arbeitsspeicher

Beachte: CAS lässt sich mittels Sperren nachbilden

- Uniprozessoren** • Unterbrechungs- oder Verdrängungssperre
- Multiprozessoren** • Umlaufsperre

Bedingte Wertzuweisung: CAS als Maschinenprogramm

Kritischer Abschnitt

```
int cas(atom_t *ref, word_t exp, word_t val) {
    bool done;

    CS_ENTER(&ref->lock);
    if (done = (ref->data == exp)) ref->data = val;
    CS_LEAVE(&ref->lock);

    return done;
}
```

```
typedef struct {
    word_t data;
    lock_t lock;
} atom_t;
```

- ref** • die Adresse des atomar, bedingt zu ändernden Speicherworts
- exp** • der unter der Adresse *ref* erwartete alte Wert
- val** • der unter der Adresse *ref* zu speichernde neue Wert
 - die Speicherung erfolgt nur, wenn der unter *ref* gespeicherte Wert dem erwarteten Wert *exp* gleicht
 - lag Gleichheit vor, liefert die Funktion *true*, anderenfalls *false*

Bedingte Wertzuweisung: CAS als Spezialbefehl (x86)

```
ZF = (eax == *ref) ? (*ref = val, true) : (eax = *ref, false)
```

```
int cas(word_t *ref, word_t exp, word_t val) {
    unsigned char aux;

    __asm__ __volatile__(
        "lock\n\t"                /* prefix next instruction */
        "cmpxchgl %2,%1\n\t"      /* (ref) == exp ? (ref) = val */
        "sete %0"                 /* extend ZF into aux */
        : "=q" (aux), "=m" (*ref)
        : "r" (val), "m" (*ref), "a" (exp) /* %eax loaded with exp */
        : "memory");

    return aux;
}
```

- lock** • setzt die Bussperre für den nachfolgenden Befehl
 • zwingend für Multi(kern)prozessorsysteme, optional sonst
- cmpxchgl** • *compare & exchange*: $ZF = true \rightarrow$ Speicherung erfolgt
- sete** • definiert Variable done mit dem Wert der ZF-Flagge

Nichtblockierende Synchronisation mit CAS

erledige NBS mit CAS;

wiederhole

ziehe *lokale Kopie* des Inhalts der *Adresse* einer globalen Variablen;
 verwende die Kopie, um einen neuen *lokalen Wert* zu berechnen;
 versuche CAS: an *Adresse*, die *lokale Kopie* mit dem *lokalen Wert*;

solange CAS scheitert;

basta.

- pros**
- Tolerierung beliebiger Überlappungsmuster
 - transparent für die Einplanung: **keine Prioritätsumkehr**
 - **Vorbeugung von Verklemmungen**: Bedingung 1 entkräftet [7]
 - Robustheit: **keine hängenden Sperren** bei Prozessabbrüchen
 - in funktionaler Hinsicht wiederverwendbar und komponierbar
- cons**
- Gefahr von **Aushungerung** (engl. *starvation*)
 - Wiederverwendung sequentieller Altsoftware unmöglich
 - **Entwicklung** nebenläufiger Varianten im Regelfall **nicht trivial**

Gliederung

1 Nichtblockierende Synchronisation

- Grundlagen
- Prinzip

2 Fallstudien

- Zählermanipulation
- Listenmanipulation
 - Stapel
 - Schlange

3 Eigenarten

- ABA
- Fehlversuche

4 Zusammenfassung

5 Anhang

FAA: *fetch and add*

Lesen & Aktualisieren einer Zählervariablen durch ein **wettlauftolerantes Maschinenprogramm**

- mit sperrfreier (engl. *lock-free*, Abk. lf) Fortschrittsgarantie [2]:
 - garantiert systemweiten Durchsatz
 - riskiert jedoch **Aushungerung** gleichzeitiger, einzelner Prozesse

```
int lf_faa(int *this, int rate) {
    int copy;

    do copy = *this;                /* fetch contents */
    while (!CAS(this, copy, copy + rate)); /* add value if unvaried */

    return copy;                    /* return (old) contents */
}
```

```
#define CAS(r,e,v)    cas((word_t*)r, (word_t)e, (word_t)v)
```

FAA: Plausibilitätskontrolle

- copy = *this* • zieht eine lokale Kopie der zu manipulierenden Zahl
- copy + rate* • berechnet den neuen Wert auf Basis dieser Kopie
- CAS* • versucht, den neuen Wert zu binden (engl. *commit*)
- true* $\iff$ kein Zugriffskonflikt, neuer Wert gültig
- false* $\iff$ **Zugriffskonflikt**, neuer Wert verworfen
- do ... while* • terminiert nur, falls der neue Wert gebunden wurde

Beachte $\leftrightarrow$ ABA-Fall (vgl. S. 33)

- die Feststellung des Zugriffskonflikts basiert auf einer Überprüfung des Inhalts einer Speicherstelle, nicht auf der Anwendung ihrer **Adresse**
- nicht jeder **überlappende Schreibzugriff** ist daher wirklich erkennbar
 - z.B., wenn zwischenzeitlich eine Änderung um n und $-n$ erfolgt

FAA: Abbildung auf einen Spezialbefehl

Fortschrittsgarantie ist dabei **allein durch die Hardware** gegeben, nicht mehr durch einen als Maschinenprogramm implementierten Algorithmus

```
#define FAA(r,v) faa((int *)r, (int)v)
```

```
int wf_faa(int *this, int rate) {
    return FAA(this, rate);
}
```

```
gcc -O6 -S
```

```
wf_faa:
    movl    4(%esp), %edx
    movl    8(%esp), %eax
    lock
    xaddl    %eax, (%edx)
    ret
```

```
INLINE int faa(int *ref, int val) {
    int aux = val;

    asm volatile (
        "lock\n\t"
        "xaddl %0,%1"
        : "=g" (aux), "=g" (*ref)
        : "0" (aux), "1" (*ref));

    return aux;
}
```

- wartefrei (engl. *wait-free*, Abk. wf)) [2]:
 - garantiert systemweiten Durchsatz und ist frei von Aushungerung

Gliederung

- 1 Nichtblockierende Synchronisation
 - Grundlagen
 - Prinzip
- 2 Fallstudien
 - Zählermanipulation
 - Listenmanipulation
 - Stapel
 - Schlange
- 3 Eigenarten
 - ABA
 - Fehlversuche
- 4 Zusammenfassung
- 5 Anhang

Stapel: *last in, first out* (LIFO)

Verwendung z.B. für **stapelorientierte Fadeneinplanung** [1], die Fäden nach LCFS (Abk. für (engl.) *last come, first served*) verarbeitet

dos — Abk. für (engl.) *devoid of synchronization*

```
void dos_push(chain_t *head, chain_t *item) {
    item->link = head->link;
    head->link = item;
}
```

```
chain_t *dos_pull(chain_t *head) {
    chain_t *node

    if ((node = head->link) != 0)
        head->link = node->link;

    return node;
}
```

push $\models$ Bereitstellung eines Fadens

pull $\models$ Auswahl eines Fadens

```
void dos_zero(chain_t *head) {
    head->link = 0;
}
```

Listenelement

```
typedef struct chain chain_t;

struct chain {
    chain_t *link;
};
```

Stapel: Nichtblockierend synchronisiert

Element durch ein **wettlauftolerantes Maschinenprogramm** am Kopf der Liste einfügen (*push*) und entnehmen (*pull*)

- mit sperrfreier (engl. *lock-free*) Fortschrittsgarantie [2]

```
void lf_push(chain_t *head, chain_t *item) {
    do item->link = head->link;          /* is elected head */
    while (!CAS(&head->link, item->link, item)); /* try push item */
}
```

```
chain_t *lf_pull(chain_t *head) {
    chain_t *node;

    do if ((node = head->link) == 0) break; /* access head */
    while (!CAS(&head->link, node, node->link)); /* try pull node */

    return node;
}
```

Stapel: Plausibilitätskontrolle

- kritischer Datenbestand ist *link_{head}*, der Listenkopf

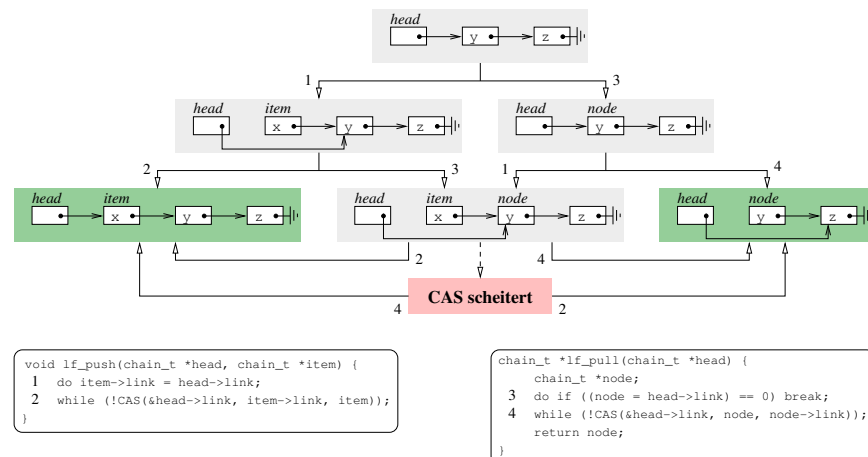
- push* # das durch *item* adressierte Element ist noch nicht in der Liste und es wird auch nicht mehrfach in diese Liste eingetragen
- der Listenkopf ist in *link_{item}* als Kopie vermerkt
 - neuer Listenkopf wäre *item*, dessen Bindung CAS versucht
 - do ... while* terminiert, falls *item* als Kopf gebunden wurde
- pull*
- der Listenkopf ist in *node* als (lokale) Kopie vermerkt
 - neuer Listenkopf wäre *node_{link}*, dessen Bindung CAS versucht
 - do ... while* terminiert, falls *node_{link}* als Kopf gebunden wurde

Beachte

- auch wenn mehrere Fäden auf denselben Kopfzeiger gleichzeitig zugreifen, wird CAS für nur einen Faden die Manipulation zulassen
- je nach Nutzung von *push* und *pull* droht das „ABA-Problem“

Stapel: Plausibilitätskontrolle (Forts.)

Datenstrukturentwicklung je nach Überlappungsfall



Gliederung

- 1 Nichtblockierende Synchronisation
 - Grundlagen
 - Prinzip
- 2 Fallstudien
 - Zählermanipulation
 - Listenmanipulation
 - Stapel
 - Schlange
- 3 Eigenarten
 - ABA
 - Fehlversuche
- 4 Zusammenfassung
- 5 Anhang

Schlange: *first in, first out* (FIFO)

Verwendung z.B. für eine **ankunftszeitorientierte Fadeneinplanung**, die Fäden nach FCFS (Abk. für (engl.) *first come, first served*) verarbeitet

```
chain_t *dos_fetch(queue_t *this) {
    chain_t *node;
    if ((node = this->head.link) && !(this->head.link = node->link))
        this->tail = &this->head;
    return node;
}
```

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

```
void dos_reset(queue_t *this) {
    this->head.link = 0;
    this->tail = &this->head;
}
```

Schlängenkopf

```
typedef struct queue {
    chain_t head;
    chain_t *tail;
} queue_t;
```

aback $\models$ Fadenbereitstlg.
fetch $\models$ Fadenauswahl

Schlange: Szenario 1, *aback* || *aback*

```
void lf_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 $ref(link_{item})$
- CAS gelingt $\iff$ *tail* hält den alten Wert $\Rightarrow tail = ref(link_{item})$
- *do ... while* terminiert $\iff$ CAS gelingt

Schlange: Szenario 2, *fetch* || *fetch*

```
chain_t *lf_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:
 - $link_{head} = node \wedge tail = ref(link_{node}) \Rightarrow$ einziges Element
- im Falle des einzigen Elements gilt nach weiteren Verlauf:
 - $link_{head} = link_{node} = 0 \Rightarrow tail = ref(link_{head})$

Schlange: Szenario 3, *fetch* || (*fetch*, *aback*)

```
chain_t *lf_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: $tail = ref(link_{item})$
 - dann müsste jedoch auch gelten: $link_{head} = item$
- $\Rightarrow$ **Widerspruch**: *fetch* toleriert überlappendes *aback* nicht

Schlange: 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 $\leftrightarrow$ 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 $ref(link_{head})$, obwohl zwischenzeitlich ggf. ein weiteres (zweites) Element an die Liste angehängt wurde
 - muss das Zurücksetzen des Fußzeigers ($tail$) in Abhängigkeit vom Ausführungsverlauf von $aback$ vornehmen

Schlange: $aback$ und $fetch$ müssen einander helfen

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:

$$link_{that} = \begin{cases} that, & that \text{ ist gültiges Verkettungsglied} \\ 0, & \text{Verkettungsglied } that \text{ wurde entfernt} \\ \text{sonst,} & \text{Listenelement } that \text{ mit Nachfolger} \end{cases}$$

- 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

Schlange: 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: $tail = ref(link_{item}) \wedge link_{head} = 0$
 - $\Rightarrow aback$: $link_{head}$ ist auf $item$ zu korrigieren

Schlange: Wettlauftolerantes $aback$

```
void lf_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 $\leftrightarrow$ Problem ABA: $item$

- zwischenzeitiges wieder einfügen von Elementen, die zuvor in der Schlange standen, kann CAS in $fetch$ gelingen lassen
- ggf. muss $aback$ weitere Hilfestellung leisten, um $fetch$ von diesem Problem zu befreien

Schlange: Wettlauftolerantes *fetch*

```
chain_t *lf_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 $\hookrightarrow$ Problem ABA: $next = link_{node}$ (gepaart)

- zwischenzeitiges wieder einfügen von *node* kann CAS gelingen lassen, wobei $link_{node}$ aber mittlerweile einen anderen Wert haben könnte

Schlange: Plausibilitätskontrolle

Neuralgische Punkte

```
void lf_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 *lf_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,
                     (next = node->link) == node ? 0 : next));

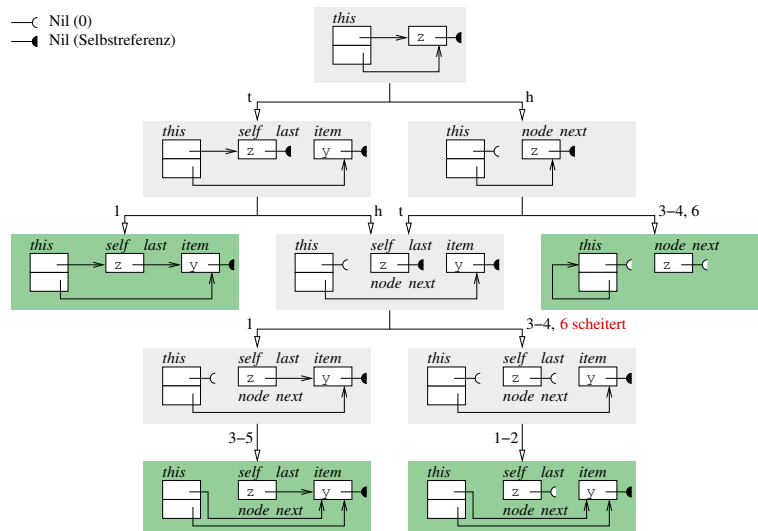
        3 if (next == node) {
        4   if (!CAS(&node->link, next, 0))
        5     this->head.link = node->link;
        6   else CAS(&this->tail, &node->link, &this->head);
        }

        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

Schlange: Plausibilitätskontrolle (Forts.)

Datenstrukturentwicklung je nach Überlappungsfall



Gliederung

- 1 Nichtblockierende Synchronisation
 - Grundlagen
 - Prinzip
- 2 Fallstudien
 - Zählermanipulation
 - Listenmanipulation
 - Stapel
 - Schlange
- 3 Eigenarten
 - ABA
 - Fehlversuche
- 4 Zusammenfassung
- 5 Anhang

Problem ABA

Phänomen der nichtblockierenden Synchronisation auf Basis eines CAS, d.h., einer ELOP, die inhaltsbasiert arbeitet [3, S. 125]²

- angenommen zwei Fäden, F_1 und F_2 , stehen im Wettstreit um eine gemeinsame Variable V
 - F_1 liest den Wert A von V , speichert diesen als Kopie, wird dann allerdings vor dem CAS_V für unbestimmte Zeit verzögert
 - F_2 durchläuft dieselbe Sequenz, schafft jedoch mittels CAS_V den Wert B an V zuzuweisen
 - anschließend wird (in einem weiteren Durchlauf dieser Sequenz) wieder der ursprüngliche Wert A an V zugewiesen
 - F_1 setzt seine Ausführung mit CAS_V fort, erkennt, dass V den Wert A seiner Kopie speichert und überschreibt V
- im Ergebnis kann dieses Überlappungsmuster dazu führen, dass F_1 mittels CAS_V einen falschen Wert nach V transferiert

²Bei *Adressreservierung* wie z.B. mit LL/SC besteht dieses Problem nicht.

Wartestapel mit Wettlaufsituation

Ausgangszustand der (LCFS) Liste: $head \rightarrow A \rightarrow B \rightarrow C$, $head$ ist ref_{CAS} :

	$\mathcal{M}$	Op.	CAS-Parameter			Liste
			*ref	exp	val	
1.	F_1	<i>pull</i>	A	A	B	unverändert
2.	F_2	<i>pull</i>	A	A	B	$ref \rightarrow B \rightarrow C$
3.	F_2	<i>pull</i>	B	B	C	$ref \rightarrow C$
4.	F_2	<i>push</i>	C	C	A	$ref \rightarrow A \rightarrow C$
5.	F_1	<i>pull</i>	A	A	B	$ref \rightarrow B \rightarrow \odot$ $A \rightarrow C$ verloren

- F_1 wird im *pull* vor CAS unterbrochen, behält lokalen Zustand bei
- 4. F_2 führt die Operationen komplett aus, aktualisiert die Liste
- F_1 beendet *pull* mit dem zum Zeitpunkt 1. gültigen lokalen Zustand

Kritische Variable mittels „Zeitstempel“ absichern

Abhilfe besteht darin, den umstrittenen Zeiger (nämlich *item* bzw. *node*) um einen problemspezifischen **Generationszähler** zu erweitern

- Etikettieren**
- Zeiger mit einem Anhänger (engl. *tag*) versehen
 - Ausrichtung (engl. *alignment*) ausnutzen, z.B.:

$$\text{sizeof}(\text{chain_t}) \rightsquigarrow 4 = 2^2 \Rightarrow n = 2$$

$$\Rightarrow \text{chain_t} * \text{ ist Vielfaches von } 4$$

$$\Rightarrow \text{chain_t} * \text{Bits}[0:1] \text{ immer } 0$$

DCAS

- Platzhalter für n -Bit Marke/Zähler in jedem Zeiger
- Abk. für (engl.) *double compare and swap*
- Marke/Zähler als elementaren Datentyp auslegen
 - unsigned int* hat Wertebereich von z.B. $[0, 2^{32} - 1]$
- zwei Maschinenworte (Zeiger, Marke/Zähler) ändern
- push* bzw. *pull* verändern sodann den Zeigerwert um eine Generation

Generationszähler „considered harmful“?

Abhilfe (engl. *workaround*): „umgehen unlösbarer Fehler“ [4]

- die Effektivität des Lösungsansatzes steht und fällt mit dem für den Generationszähler definierten **endlichen Wertebereich**
 - dessen Auslegung letztlich vom jeweiligen Anwendungsfall abhängt
- Überlappungsmuster** gleichzeitiger Prozesse haben Einfluss auf den für den Generationszähler zur Verfügung zu stellenden Wertebereich
 - bestimmt durch Zusammenspiel *und* Anzahl der wettstreitigen Fäden
 - ein Bit kann reichen, ebenso, wie ein *unsigned int* zu klein sein kann
- diese, dem jeweiligen Anwendungsfall zu entnehmenden Muster zu entdecken, ist zumeist schwer und nicht selten unmöglich

Vorbeugung (engl. *prevention*) muss zuerst kommen — sofern machbar:

- beliebige Überlappungsmuster konstruktiv (Entwurf) ausschließen
- oder auf **Adressreservierungsverfahren** der Hardware zurückgreifen
 - unterstützt nicht jede Hardware, ist nur typisch für RISC
 - z.B. ELOP-Paar *load linked, store conditional* (LL/SC) verwenden

Abhilfe zum ABA-Problem: Etikettierung

Abstrakter Datentyp *chain_p*: Spezialisierung von *chain_t**

```
typedef chain_t* chain_p;
```

```
extern chain_p aba_wheel(chain_p); /* rotate pointer tag bit(s) */
extern chain_t *aba_index(chain_p); /* return pointer value */
```

Etikett anheften

```
chain_p aba_wheel(chain_p item) {
    return (chain_p)((unsigned)item ^ 1);
}
```

Etikett entfernen

```
chain_t *aba_index(chain_p item) {
    return (chain_t *)((unsigned)item & ~1);
}
```

Verwendung

- wheel*
 - markieren
 - Zeiger färben
- index*
 - bereinigen
 - Zeiger liefern

Lebensdauer des Generationszählers bzw. der Etiketten

Abhilfe gegen das Problem der Mehrdeutigkeit (hier: ABA) von Zeigern ist **nicht transparent** für die Programme, die die Zeiger verwenden

- dies trifft insbesondere auch zu auf die Etikettierung von Zeigern
 - damit bleiben lediglich *Einzelwort*-CAS weiterhin möglich
 - Transparenz durch Beibehaltung der Zeigergröße ist nicht das Ziel
- voll ausgeprägte Generationszähler sind offensichtlich intransparent
 - Zeiger und Generationszähler müssen eine Einheit bilden
 - beide zusammen verdoppeln die Zeigergröße $\leadsto$ *Doppelwort*-CAS

Zeiger samt Generationszähler/Etikett sind Exemplar eines Typs

- problemspezifische Auslegung, je nach Wertebereich des Zählers
 - Einzelwort* *chain_t**, falls einfache Etikettierung genügt
 - Doppelwort* *chain_t** und *unsigned int*, sonst
- Repräsentation als **abstrakter Datentyp** [5] $\Rightarrow$ Anpassung notwendig

Stapel: Etikettierter Zeigertyp *chain_p*

```
void lf_push(chain_t *this, chain_p item) {
    chain_p turn = aba_wheel(item); /* new pointer generation */

    do aba_index(item)->link = this->link;
    while (!CAS(&this->link, aba_index(item)->link, turn));
}

chain_p lf_pull(chain_t *this) {
    chain_p node;

    do if (aba_index((node = this->link)) == 0) break;
    while (!CAS(&this->link, node, aba_index(node)->link));

    return node;
}
```

Beachte $\hookrightarrow$ abstrakter Datentyp *chain_p* $\mapsto$ *chain_t**

- Exemplare dieses Typs dürfen nur mittels *index* benutzt werden

Wiederholungsversuche — Aktives Warten?

Scheitern der etwa durch CAS³ abzuschließenden **Transaktion** zieht die Wiederholung des kompletten Vorbereitungsvorgangs nach sich

- je höher der Grad an Wettstreitigkeit unter gleichzeitigen Prozessen, umso höher die Wahrscheinlichkeit, dass CAS scheitert
- ein zur Umlaufsperrung sehr ähnliches Problem ergibt sich, das jedoch durch **sensitive Techniken** gleicher Art lösbar ist (vgl. Kap. VII-1)
 - Häufigkeit von „*read-modify-write*“-Zyklen pro Durchlauf minimieren
 - prozessspezifisches Zurücktreten vom erneuten Transaktionsversuch
 - variable Wartezeiten, um Konflikte bei Wiederholungen zu vermeiden

Unterschied zur Umlaufsperrung

- gleichzeitige Prozesse müssen nicht untätig darauf warten, dass ein kritischer Abschnitt frei ist, d.h., das TAS gelingt
- sie verlassen die Schleife jedoch erst, wenn das CAS gelingt, kommen in der Schleife allerdings mit ihren (lokalen) Berechnungen voran

³Für LL/SC-artige Elementaroperationen gilt dies ebenfalls.

- 1 Nichtblockierende Synchronisation
 - Grundlagen
 - Prinzip
- 2 Fallstudien
 - Zählermanipulation
 - Listenmanipulation
 - Stapel
 - Schlange
- 3 Eigenarten
 - ABA
 - Fehlversuche
- 4 Zusammenfassung
- 5 Anhang

- [1] BAKER, T. P.:
Stack-Based Scheduling of Realtime Processes.
In: *Real-Time Systems* 3 (1991), März, Nr. 1, S. 67–99
- [2] HERLIHY, M. :
Wait-Free Synchronization.
In: *ACM Transactions on Programming Languages and Systems* 11 (1991), Jan., Nr. 1, S. 124–149
- [3] IBM CORPORATION (Hrsg.):
IBM System/370 Principles of Operation.
Fourth.
White Plains, NY, USA: IBM Corporation, Sept. 1 1975. –
GA22-7000-4, File No. S/370-01
- [4] LEO GMBH:
LEO Deutsch-Englisches Wörterbuch.
Sauerlach, Deutschland : <http://dict.leo.org>,
- [5] LISKOV, B. J. H. ; ZILLES, S. N.:
Programming with Abstract Data Types.
In: LEAVENWORTH, B. (Hrsg.): *Proceedings of the ACM SIGPLAN Symposium on Very High Level Languages* Bd. 9.
New York, NY, USA : ACM, Apr. 1974 (ACM SIGPLAN Notices 4), S. 50–59
- [6] SCHÖN, F. ; SCHRÖDER-PREIKSCHAT, W. ; SPINCZYK, O. ; SPINCZYK, U. :
On Interrupt-Transparent Synchronization in an Embedded Object-Oriented Operating System.
In: LEE, I. (Hrsg.) ; KAISER, J. (Hrsg.) ; KIKUNO, T. (Hrsg.) ; SELIG, B. (Hrsg.): *Proceedings of the 3rd IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC '00)*.
Washington, DC, USA : IEEE Computer Society, 2000, S. 270–277

- **Wettlauftoleranz** stellt sich als Merkmal kritischer Abschnitte dar
 - eine durch die Art des Schutzes bedingte **nichtfunktionale Eigenschaft**
 - gleichzeitige Prozesse nichtsequentieller Programme nie aussperren
- klassischer Ansatz dazu ist die **nichtblockierende Synchronisation**
 - Spezialbefehl: Einzelwort „*compare and swap*“ (CAS)
 - Zähler- und Listenmanipulation (Stapel, Schlange) als Fallstudien
 - Sonderfall: unterbrechungstransparente Synchronisation (s. Anhang)
 - kommt ohne Spezialbefehl des (realen) Prozessors aus
 - bedingt atomare Lese-/Schreiboperationen auf dem Arbeitsspeicher
- die Verfahren bringen **Eigenarten** mit sich und bergen Gefahren
 - bei CAS-basierten Algorithmen droht das ABA-Problem
 - Abhilfe schafft die Etikettierung der kritischen Variablen $\leadsto$ CAS
 - wo dies nicht möglich ist, können Generationszähler abhelfen $\leadsto$ DCAS
 - aktives Warten bei Fehlversuchen ähnlich zu Umlaufsperrern behandeln
- grundsätzlich ist ein **anderes Denken** in der Algorithmenentwicklung

- [7] SCHRÖDER-PREIKSCHAT, W. ; KLEINÖDER, J. :
Systemprogrammierung.
http://www4.informatik.uni-erlangen.de/Lehre/WS08/V_SP, 2008 ff.
- [8] TANENBAUM, A. S.:
Multilevel Machines.
In: *Structured Computer Organization*.
Prentice-Hall, Inc., 1979. –
ISBN 0-130-95990-1, Kapitel 7, S. 344–386

Gliederung

1 Nichtblockierende Synchronisation

- Grundlagen
- Prinzip

2 Fallstudien

- Zählermanipulation
- Listenmanipulation
 - Stapel
 - Schlange

3 Eigenarten

- ABA
- Fehlversuche

4 Zusammenfassung

5 Anhang

Unterbrechungstransparente Synchronisation [6]

Koordinierung von unterbrechungsbedingten Aktivitäten, ohne **asynchrone Programmunterbrechungen** abzuwehren

- die Systemsoftware ist *gänzlich* frei von Unterbrechungssperren⁴
- Synchronisation verwendet nur nichtprivilegierte Befehle der ISA
 - mit gewissen **Atomizitätseigenschaften** bezüglich Unterbrechungen
 - wie z.B. **atomares lesen/schreiben von Speicherworten** ($n > 1$ Bytes)
- nur die Hardware selbst sperrt Unterbrechungen (zeitweilig) aus
 - gleichwohl bestimmt Software, wie *häufig* diese Sperren aktiv sind !!!

Beachte ↔ Hilfestellung durch andere Konzepte

- um Unterbrechungssynchronisation zur *Ausnahme* werden zu lassen
 - z.B. Fortsetzungssperren, um KA konventionell schützen zu können
- Konzentration auf das Wesentliche: **Nachspannliste** (Kap. II-3, S. 14)

⁴Erneute Abwehr von Unterbrechungen nach Unterbrechungsfreigabe, setzt keine Unterbrechungssperre im eigentlichen Sinn (vgl. Kap. II-3, S. 13).

Schlange: Nachspannliste zur AST-Propagierung

its — Abk. für (engl.) *interrupt transparent synchronization*

```
extern void    its_reset(queue_t *);          /* clear queue */
extern void    its_aback(queue_t *, chain_t *); /* append chain item */
extern chain_t its_fetch(queue_t *);          /* remove chain item */
```

```
void its_reset(queue_t *this) {
    dos_reset(this);
}
```

```
void its_aback(queue_t *, chain_t *) {
    ...
}
```

```
chain_t *its_fetch(queue_t *this) {
    chain_t *item = dos_fetch(this);
    ...
    return item;
}
```

its ↔ **dos**

reset Wiederverwendung

- *chain_t*
- *queue_t*

aback Ersetzung

fetch Spezialisierung

Schlange: Wettlauftolerantes *aback* (Forts.)

Element einfügen

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

    item->link = 0;          /* make item last chain element */

    last = this->tail;        /* remember item insertion point */
    this->tail = item;        /* advance tail pointer, optimistically */

    while (last->link)        /* overlapping aback: find actual tail */
        last = last->link;

    last->link = item;        /* append item */
}
```

Atomizität • lesen/schreiben von Zeigerwerten ist ELOP

Überlappungsmuster • *aback* überlappt *aback* oder *fetch*

Schlange: Wettlauftolerantes *fetch* (Forts.)

Element entfernen

(dos.fetch expandiert)

```

chain_t *its_fetch(queue_t *this) {
    chain_t *item;

    if ((item = this->head.link) && !(this->head.link = item->link)) {
        this->tail = &this->head;        /* point of problem! */
        if (item->link) {                /* race condition detected! */
            chain_t *help, *lost = item->link;
            do {                        /* requeue lost elements */
                help = lost->link;
                its_aback(this, lost);
            } while ((lost = help));
        }
    }
    return item;
}

```

Überlappungsmuster • *fetch* wird nur von *aback* überlappt

Schlange: Plausibilitätskontrolle (Forts.)

aback

- überlappt sich selbst immer nur **stapelweise**, wenn überhaupt
- Wettauflausituation $\iff last = tail$, jedoch $tail \neq item$
 - Normalfall** $link_{last} = 0 \Rightarrow$ kein *aback*-Wiedereintritt
 - Konfliktfall** $link_{last} \neq 0 \Rightarrow last$ falsch, korrigieren
- Zuweisung an *link* (einfügen von *item*) $\iff last \neq tail$

fetch

- überlappt sich **nie selbst**, auch nicht durch Verdrängung
- Wettauflausituation $\iff head = 0$, jedoch $tail \neq ref(head)$
 - Normalfall** $link_{item} = 0 \Rightarrow$ *aback* überlappte nicht
 - Konfliktfall** $link_{item} \neq 0 \Rightarrow item \leadsto lost \& found$
- umtragen der Einträge aus „*lost & found*“-Liste ist atomar

Beachte $\leftrightarrow$ Parallelverarbeitung

Nachspannliste • die **Überlappungsmuster** können beliebig ausfallen