Betriebssystemtechnik

Ablaufsteuerung

15./22./29. Juni 2009

Überblick

Ablaufsteuerung
- Einleitung
- Wettlaufintoleranz
  - Unterbrechungssperre
  - Fortsetzungssperre
- Wettlauftoleranz
  - Unterbrechungstransparente Synchronisation
  - Nichtblockierende Synchronisation
- Zusammenfassung
- Bibliographie

Motiv: Synchronisation

Koordination der Kooperation und Konkurrenz gleichzeitiger Programmabläufe

ko·or·di'nie·ren beio*rdnen*; in ein Gefüge einbauen; aufeinander abstimmen; nebeneinanderstellen; Termine ∼.

- sich überlappen könnende Aktivitäten *der Reihe nach* ausführen
  - sicherstellen, kritische Abschnitte konsistent zu durchlaufen
- „der Reihe nach" ⇝ die Verzögerung von Prozessen erzwingen
  - die überlappende oder die überlappte Aktivität, je nach Verfahren

Lernziel

- Techniken der Befehlssatzebene zur Synchronisation begreifen
- unteilbare (wettlaufintolerante) kritische Abschnitte eingrenzen
- teilbare (wettlauftolerante) kritische Abschnitte entwickeln

Einordnung

Schicht	Funktion	Konzepte
12	Programmverwaltung	Text, Daten, Überlagerung
11	Dateiverwaltung	Dateisystem; Verzeichnis, Verknüpfung
10	Prozessverwaltung	Aktivitätsträger, Kontext, Stapel
9	Adressraumverwaltung	Arbeitsspeicher, Segment, Seite
8	Informationsaustausch	Paket, Nachricht, Kanal, Portal
7	Geräteprogrammierung	Kern; Signal, Zeichen, Block, Datenstrom
6	Platzanweisung	Hauptspeicher, Fragment, Seitenrahmen
5	Zugriffskontrolle	Subjekt, Objekt, Domäne, Befähigung
4	Betriebsmittelzugriff	Verdrängungs-/Vorgangssperre
3	Auftragseinplanung	Ereignis, Priorität, Zeitscheibe, Energie
2	Ablaufsteuerung	Unterbrechungs-/Fortsetzungssperre, Wettlauftoleranz
1	Kontrollflusswechsel	Koroutine, Unterbrechung, Fortsetzung
0	Stammprozessorabstraktion	Stammsystem
-1	Peripherie	MMU, (A)PIC, DMA, UART, ATA, SCSI, USB, ...
-2	Zentraleinheit	ARM, AVR, PowerPC, SPARC, x86, ...

Rekapitulation: SOS 1 [2] bzw. SP [3], BS [4]

Kritischer Abschnitt (KA) [1, S. 137]

- sich gegenseitig ausschließende Aktivitäten werden nie parallel ausgeführt $\models$ Elementaroperation (ELOP)
  - sie verhalten sich zueinander, als seien sie unteilbar, weil keine Aktivität die andere unterbricht
- Anweisungen, deren Ausführung einen gegenseitigen Ausschluss erfordern (engl. *critical sections*, *critical regions*)

Beachte $\hookrightarrow$ Abstraktionsniveau der ELOP

- setzen von „Synchronisationsklammern“ ist nicht zwingend, um einen kritischen Abschnitt zu schützen
- vielmehr gilt es sicherzustellen, dass die Ausführung eines solchen Abschnitts jederzeit ein konsistentes Ergebnis liefert

Integritätswahrung kritischer Abschnitte

Sicherstellung, dass die Ausführung eines kritischen Abschnitts jederzeit ein konsistentes Ergebnis liefert, ist auf zwei Wegen möglich:

(a) durch zeitweiligen Ausschluss der Möglichkeit einer überlappenden Ausführung der relevanten Programmanweisungen
- Eintrittsanforderungen (problem- bzw. fallspezifisch) abwehren

(b) durch Tolerierung eben dieser Möglichkeit und Vorsehung gewisser Reparatur- bzw. Erholungsmaßnahmen
- überlappend durchführbare, transaktionsartige Verfahren
- mit/ohne spezielle Elementaroperationen der Befehlssatzebene

Beachte

- Überlappung meint je nach Prozessortyp zwei verschiedene Dinge:
  - Uniprozessor $\rightsquigarrow$ Unterbrechung und Wiedereintritt
  - Multiprozessor $\rightsquigarrow$ Parallelverarbeitung
- je nachdem fällt der Umgang mit Überlappung unterschiedlich aus

Integritätswahrung kritischer Abschnitte (Forts.)

zu (a) Eintrittsanforderungen abwehren ist *kein* gegenseitiger, sondern ein einseitiger Ausschluss
- von (hard- oder softwarebedingten) Unterbrechungen
- von (hardwarebedingten) Buszugriffen[1]

zu (b) Prinzip: nichtblockierende Synchronisation

1. lokale Kopie der zu aktualisierenden Variablen anlegen
2. lokale Kopie aktualisieren
3. Variable mit aktualisierter Kopie abgleichen $\rightsquigarrow$ Transaktion
4. scheitert 3., die Programmsequenz ab 1. erneut versuchen

Beachte

zu (a) privilegierte Befehle $\rightsquigarrow$ privilegierter Arbeitsmodus der CPU

zu (b) nichtprivilegierte Befehle $\rightsquigarrow$ beliebiger Arbeitsmodus

[1] Der Schiedsrichter (engl. *arbiter*) wird angewiesen, Zugriffe anderer Prozessoren auf den gemeinsamen Daten-/Adressbus nicht durchzulassen.

Abwehr von Eintrittsanforderungen

Elementaroperationen schließen überlappte Ausführungen eines kritischen Abschnitts ggf. durch denselben oder einen anderen Prozessor aus

selber Prozessor $\rightsquigarrow$ Unterbrechungssperre, Fortsetzungssperre

anderer Prozessor $\rightsquigarrow$ Zugriffssperre (auf den gemeinsamen Bus)

Beachte $\hookrightarrow$ Holzhammermethode

- setzen einer Sperre, obwohl die Eintrittsanforderung einen anderen kritischen Bereich betreffen kann
  - wenn KA und Anforderungsquelle (AQ) uneindeutig zugeordnet sind
    - genauer: der *Typ* des KA bzw. die *Klasse gemeinsamer Variablen*
    - d.h. jene Variablen, deren gemeinsame Aktualisierung kritisch ist
  - der Regelfall ist $N : M$ zwischen KA und AQ, mit $N \gg M$
- mögliche Nebenläufigkeit wird unnötig eingeschränkt und so letztlich auch ein wahrscheinlicher Verlust an Leistung hervorgerufen

Abwehr von Eintrittsanforderungen (Forts.)

```
orq_avert();
...             /* critical section */
orq_admit();
```

ORQ ∼ IRQ: Abk. für (engl.) *overlap request*

pros
- die einen KA bildenden Anweisungsfolgen bleiben unverändert
  - ist der Forderung nach Wiederverwendbarkeit zudienlich
- sequentielles Programmierparadigma bleibt bestehen/erhalten

cons
- die Zulassung von Eintrittsanforderungen verzögert sich
  - um die ggf. nur (schwer) abschätzbare WCET[2] des KA
- das Potential an Nebenläufigkeit wird nur suboptimal genutzt

Beachte
- einen KA durch „ORQ-Abwehr“ zu schützen, ist trivial
- dazu ist der KA aber überhaupt erst zu finden, was nicht trivial ist

[2] Ausführungszeit des schlimmsten Falls (engl. *worst-case execution time*).

Abwehr von Eintrittsanforderungen (Forts.)

Programmabläufe koordinieren sich nach dem Grundsatz: „wer zuerst kommt, mahlt zuerst“ (engl. *first come, first served*; FCFS)

- eine mit der Strategie zur Prozesseinplanung zumeist in Konflikt stehende „Anspruchshaltung“ zur Nutzung des Prozessors
  - Prozesseinplanung bestimmt die Reihenfolge der Prozessorvergabe
    - an welchen Prozess der Prozessor vergeben wird,
    - ob und ggf. wann der Prozess den Prozessor abgibt oder
    - ob dem Prozess der Prozessor entzogen werden kann, ist Strategie
  - den Prozessor nicht abzugeben, sollte dieser Strategie entsprechen
- wurden Eintrittsanforderungen abgewehrt, d.h., ist ein KA aktiv, beansprucht der laufende Prozess, den Prozessor nicht zu vergeben

Beachte
- die Entscheidungen zur Prozessorvergabe sind nicht mehr verbindlich
- Prioritätsverletzung bzw. Prioritätsumkehr kann die Folge sein

Abwehr von Unterbrechungen

Möglichkeiten der totalen oder partiellen Unterbrechungssteuerung — Beispiel Intel Hardware:

total
- in der Senke der Unterbrechungsanforderung (UA): CPU
  - Unterbrechungsschalter im FLAGS Register, Bit 9 (IF)
  - 0 ↪ UA nicht annehmen: einstellen mit `cli`, `sti` oder `popf`

partiell
- im Mittler der UA: Gerät/PIC, 8259A
  - Unterbrechungsschalter im IMR, Bits 0–6
  - 1 ↪ UA nicht weiterleiten: einstellen mit `outb` (OCW1)
- in der Quelle der UA: Gerät/UART, 8250 bzw. 16550
  - Unterbrechungsschalter im IER, Bits 0–3
  - 0 ↪ UA nicht aussenden: einstellen mit `outb`

Beachte ↪ verschachtelte kritische Abschnitte
- (1) abwehren, (2) KA, (3) wiederzulassen einer UA reicht nicht aus
- stattdessen: (1) sichern, (2) abwehren, (3) KA, (4) wiederherstellen

Abwehr von Unterbrechungen (Forts.)

Intel Familie, Auswahl

	totale Abwehr		partielle Abwehr	
Funktion	Senke (CPU)		Mittler (PIC)	Quelle (UART)

IRQ_AVERT	`cli`	`pushf`
		`cli`
			`inb IMR, %al`
			`pushl %eax`
			`movl` *mask*`, %eax`
			`outb %al, IMR`
				`inb IER, %al`
				`pushl %eax`
				`movl` *mask*`, %eax`
				`outb %al, IER`

IRQ_ADMIT	`sti`	`popf`	`popl %eax` `outb %al, IMR`	`popl %eax` `outb %al, IER`
		verschachtelte kritische Abschnitte		

sichern
- FLAGS Register, IMR oder IER

abwehren
- nicht annehmen, weiterleiten oder aussenden

zulassen
- wieder annehmen, weiterleiten oder aussenden

Abwehr von Fortsetzungen

Fortsetzung (engl. *continuation*), auch: Einfädelung (engl. *merge*)

- Nachspann*anhang* einer Unterbrechungsbehandlung
  - ist grundsätzlich unterbrechbar: $IPL = 0$
- läuft synchron mit anderen Betriebssystemaktivitäten
  - im Gegensatz zum Nachspann: asynchron zum BS
- Auslösung (durch Nachspann) erfordert Koordinierung

Beachte ↪ Fortsetzung ⊂ Epilog [4]

- Epilog = Nachspann + Fortsetzung
  - der Nachspann ist „nach unten“ orientiert, fokussiert auf (s)ein Gerät
    - Betriebssystemfunktionen dürfen nicht aufgerufen werden
  - die Fortsetzung ist „nach oben“ orientiert, fokussiert auf das BS
    - dient (einzig) dem Zweck, Betriebssystemfunktionen aufzurufen
  - sie wird durch eine Art Hochruf (engl. *upcall*, [5]) aktiviert
- die Fortsetzung (eines Nachspanns) ist ein Epilog ohne AST

Abwehr, Zulassung und Weitergabe von Fortsetzungen

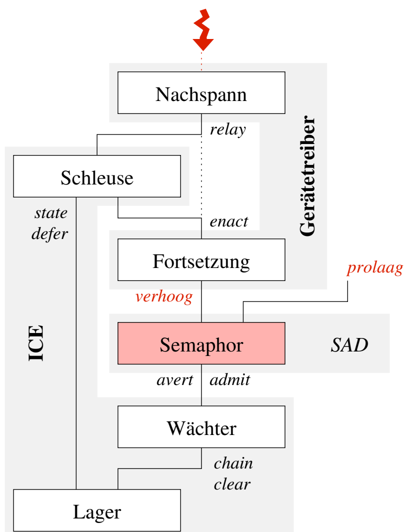
ICE (Abk. engl. *interrupt continuation executive*)

```
extern void ice_avert (ice_t *);            /* defer continuations */
extern void ice_treva (volatile ice_t *);   /* allow continuations */
extern void ice_admit (ice_t *);            /* unleash continuations */
extern void ice_clear (ice_t *);            /* process continuations */
extern void ice_defer (ice_t *, job_t *);   /* defer continuation */
extern void ice_relay (ice_t *, job_t *);   /* enact/defer cont. */

extern char   ice_state (ice_t *);          /* guard state */
extern job_t *ice_chain (volatile ice_t *); /* guard load */
extern ice_t *ice_guard ();                 /* guard singleton */
```

```
typedef struct ice {
    char busy;       /* state */
    queue_t load;    /* jobs */
} ice_t;
```

```
typedef remit_t job_t;

extern void job_enact (job_t *);
```


Fortsetzung der Ereigniszustellung: Entwurfsskizze



Nachspann	▶ SLIH ▶ vgl. Kapitel 5
Schleuse	▶ serialisiert Fortsetzungen
Wächter	▶ kontrolliert krit. Abschnitt
Lager	▶ speichert Fortsetzungen

Beachte ↪ Semaphor

- blockadefrei synchronisiert
- ein überlappendes V kommt verzögert zur Ausführung

Serialisierte kritische Abschnitte: Semaphor

```
void ewd_prolaag (semaphore_t *) {
    ice_avert(ice_guard());
    /* ... */
    ice_admit(ice_guard());
}
```

```
void ewd_verhoog (semaphore_t *) {
    ice_avert(ice_guard());
    /* ... */
    ice_admit(ice_guard());
}
```

guard	▶ ist die Verwaltungsstruktur zur Steuerung von Fortsetzungen
avert	▶ signalisiert, dass Fortsetzungen zurückzustellen sind
admit	▶ signalisiert, dass Forsetzungen nicht zurückzustellen sind ▶ verarbeitet zurückgestellte Fortsetzungen, sofern erforderlich

Beachte

- zwischen *avert* und *admit* werden Fortsetzungen zurückgestellt
- die Funktionen arbeiten blockade- aber nicht verzögerungsfrei

Fortsetzung eines Nachspanns: Treiber $\Longleftrightarrow$ Faden

```
typedef struct driver {
    job_t task;          /* SLIH continuation */
    semaphore_t port;    /* driver thread signalling socket */
} driver_t;

void job_signal (driver_t *this) { ewd_verhoog(&this->port); }

driver_t job_driver = {{{0}, &job_signal}, {0}};

void a_slih () { ice_relay(ice_guard(), (job_t*)&job_driver); }
void a_hils () { ewd_prolaag(&job_driver.port); }
```

- *a_hils* ▶ Routine eines Fadens: konsumiert ein Gerätesignal
- *a_slih* ▶ Nachspann des Treibers: löst eine Fortsetzung aus
- *job_driver* ▶ Deskriptor einer Fortsetzung
- *job_signal* ▶ Fortsetzung des Nachspanns: produziert ein Gerätesignal

Serialisierung kritischer Abschnitte

```
void ice_avert (ice_t *this) {
    this->busy = 1;
}
```

```
void ice_admit (ice_t *this) {
    ice_treva(this);
    if (ice_chain(this))
        ice_clear(this);
}
```

```
void ice_treva (volatile ice_t *this) {
    this->busy = 0;
}

job_t *ice_chain (volatile ice_t *this) {
    return (job_t*)this->load.head.link;
}
```

- *avert* ▶ serialisieren
- *admit* ▶ aufarbeiten
- *treva* ▶ ¬ *avert*
- *chain* ▶ etwas zu tun?

Beachte ↪ *admit*

- Fortsetzungen können sich überholen: kein striktes FCFS

Serialisierung kritischer Abschnitte (Forts.)

Fortsetzungen weiterleiten, d.h., bedingt aktivieren

```
void ice_relay (ice_t *this, job_t *task) {
    if (!ice_state(this)) job_enact(task);     /* idle, call it */
    else ice_defer(this, task);                /* busy, store it */
}

char ice_state (ice_t *this) {
    return this->busy;                         /* 0 = idle, 1 = busy */
}

void ice_defer (ice_t *this, job_t *task) {
    fad_aback(&this->load, &task->next);       /* append continuation */
}
```

- *relay* ▶ durchschleusen einer Fortsetzung: aktivieren/zurückstellen
- *state* ▶ Aktivierungszustand des serialisierten kritischen Abschnitts
- *defer* ▶ zurückstellen (speichern) einer Fortsetzung

Serialisierung kritischer Abschnitte (Forts.)

Fortsetzungen freistellen und aktivieren

```
void ice_clear (ice_t *this) {
    job_t *task;
    while ((task = (job_t*)fad_fetch(&this->load)))
        job_enact(task);
}

void job_enact (job_t *task) {
    (task->work)(task);        /* activate continuation object */
}
```

- *clear* ▶ Warteschlange zurückgesteller Fortsetzungen abarbeiten
- *enact* ▶ Fortsetzungs*methode* auf Fortsetzungs*objekt* applizieren

Beachte ↪ *clear*

- die Freistellung von Fortsetzungen erfolgt außerhalb des KA
- die Aktivierungsreihenfolge von Fortsetzungen ist unbestimmt

Serialisierung kritischer Abschnitte (Forts.)

Plausibilitätskontrolle

relay stellt Fortsetzungen zurück $\iff$ *state* = *BUSY*

- Programmausführung zwischen *avert* und *treva* unterbrochen

clear läuft mit *state* = *IDLE* $\Rightarrow$ keine Fortsetzung geht verloren

enact aktiviert Fortsetzungen, die ggf. serialisierte KA ausführen

admit lässt Überlappungen von Fortsetzungen zu

(a) keine Überlappung $\Rightarrow$ *clear* baut Fortsetzungsliste ab

(b) Überlappung $\Rightarrow$ indirekte Rekursion ist möglich

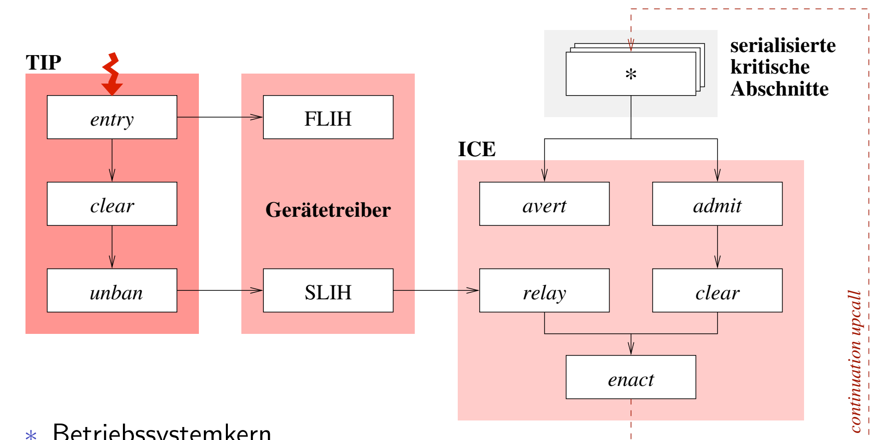
- sofern *enact* einen KA aktiviert, der mit *admit* endet
- Fall (a) ist sodann Terminationsbedingung

Beachte ↪ Rekursionstiefe, hängt ab von: (ein Fall für TAS, falls kritisch)

- Frequenz, Häufigkeit und Art von Unterbrechungen
- Funktion der Vor- und Nachspänne der Unterbrechungen
- Funktion der Fortsetzungen

Serialisierung kritischer Abschnitte (Forts.)

Gesamtzusammenhang: Aktivierungsfolge zentraler Funktionen



∗ Betriebssystemkern

(a) im normalen (synchronen) Fadenkontrollfluss aktivierte Funktionen

(b) asynchron ausgelöste, „einsynchronisierte“ Funktionen, z.B:

- *verhoog*, *preempt* – in normalen Ablauf einzufädelnde Aufrufe

Fortsetzungsliste — konstruktiv gesicherte Datenstruktur

Beachte ↪ Operationsprinzip AST (vgl. Kap. 5)

- Nachspannfreistellung (TIP, *clear*) überlappt sich selbst nicht
  - Parallelverarbeitung (Zustellergruppe) einmal außen vor gelassen
- Folge: Fortsetzungsweiterleitung (ICE, *clear*) überlappt sich nie
  - auch Verdrängungsereignisse werden sich hinten anstellen:

    Verdrängung $\iff$ Vorspann $\mapsto$ Nachspann $\mapsto$ Fortsetzung

  - durch Konstruktion bedingte Wechselwirkung (engl. *interaction*)
    - beugt Wettlaufsituationen in Bezug auf die Fortsetzungsliste vor
    - macht explizite Synchronisation der Zurgiffsoperationen überflüssig

FAD (Abk. für engl. *fundamental algorithms and data structures*)

```
extern void fad_reset (queue_t *);              /* clear queue */
extern void fad_aback (queue_t *, chain_t *);   /* append chain item */
extern chain_t *fad_fetch (queue_t *);          /* remove chain item */
```


Fortsetzungsliste: FAD

```
typedef struct chain chain_t;

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

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

```
typedef struct queue queue_t;

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

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

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

Unterbrechungstransparente Synchronisation [6]

Koordinierung unterbrechungsbedingter Aktivitäten, ohne asynchrone Programmunterbrechungen abzuwehren

- die Systemsoftware ist frei von Unterbrechungssperren[3]
- Synchronisation verwendet nur nichtprivilegierte Befehle der ISA
  - mit gewissen Atomizitätseigenschaften in Bezug auf Unterbrechungen
  - wie z.B. atomares lesen/schreiben von Speicherworten ($n > 1$ Bytes)
  - aber auch TAS, FAA oder CAS, d.h., atomare Komplexbefehle
- nur die Hardware selbst sperrt Unterbrechungen (zeitweilig) aus
  - gleichwohl bestimmt Software, wie lange 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
- die Konzentration auf das Wesentliche fördern: Fortsetzungsliste

[3] Erneute Abwehr von Unterbrechungen nach Unterbrechungsfreigabe, setzt keine Unterbrechungssperre im eigentlichen Sinn (vgl. Kapitel 5).

Fallstudie: Warteschlangensynchronisation

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) {
    fad_reset(this);
}
```

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

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

ITS ⟺ FAD

reset Wiederverwendung
- *chain_t*
- *queue_t*

aback Ersetzung

fetch Spezialisierung

Fallstudie: Warteschlangensynchronisation (Forts.)

Element wettlauftolerant 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*

Fallstudie: Warteschlangensynchronisation (Forts.)

Element wettlauftolerant entfernen (*fad_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

Fallstudie: Warteschlangensynchronisation (Forts.)

Plausibilitätskontrolle

aback
- überlappt sich selbst immer nur stapelweise, wenn überhaupt
- Wettlaufsituation $\iff$ *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
- Wettlaufsituation $\iff$ *head* = 0, jedoch *tail* $\neq$ *&head*
  - Normalfall *LINK(item)* = 0 $\Rightarrow$ *aback* überlappte nicht
  - Konfliktfall *LINK(item)* $\neq$ 0 $\Rightarrow$ *item* $\rightsquigarrow$ *lost & found*
- umtragen der Einträge aus „*lost & found*"-Liste ist atomar

Beachte $\hookrightarrow$ Eignung für TIP und ICE

- Nachspannliste: zutreffend, jedoch nicht bei Parallelverarbeitung
- Fortsetzungsliste: unzutreffend: AST serialisiert Nachspänne !!!

Rückblick: Nachspann-/Fortsetzungliste

Querschneidende Belange durch Verdrängung und Parallelverarbeitung

Nachspannliste
- Aufbau auf Anforderung durch einen Vorspann
  - Vorspänne überlappen sich nur stapelweise
  - dito: Fließbandmodell der Parallelverarbeitung
- Abbau im Rahmen der Behandlung eines AST
  - Freistellung der Nachspänne ist unteilbar und überlappt deren Zurückstellung nicht
  - gilt bei Parallelverarbeitung nicht

Fortsetzungsliste
- Aufbau auf Anforderung durch einen Nachspann
  - Nachspänne überlappen sich nie: AST serialisiert
  - gilt bei Parallelverarbeitung nur bedingt
- Abbau beim Verlassen eines serialisierten KA
  - Freistellung der Fortsetzungen ist bedingt teilbar und überlappt deren Zurückstellung nicht
  - gilt bei Parallelverarbeitung nicht

Beachte $\hookrightarrow$ Parallelverarbeitung (Multiprozessoren)

- lässt Annahmen zum Überlappungsmuster nicht aufrechterhalten

Nichtblockierende Synchronisation

Koordinierung sich einander ggf. überlappender Aktivitäten, ohne dabei gleichzeitige Prozesse[4] 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
- d.h., sie funktionieren im Benutzer- wie auch auch im Systemmodus

Beachte

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

[4] Prozesse, deren Ausführung sich zeitlich überschneidet.

CAS: Prinzip

compare and swap
- Elementaroperation der Befehlssatzebene[5]
- unteilbar: Unterbrechungs- *und* Zugriffssperre (Datenbus)
- implementiert eine „Transaktion" für Uni- und Multiprozessoren

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

    orq_avert();
    if (aux = (*ref == exp)) *ref = val;
    orq_admit();

    return aux;
}
```

Operationsergebnis

- *true*: gleich, geschrieben, gelungen
- *false*: ungleich, gelesen, gescheitert

Generalisierte Schnittstelle

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

[5] Lässt sich durch passende Auslegung von *avert* und *admit* ggf. nachbilden.

CAS: Nachbildung für x86

```
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;
}
```

Beachte ↪ *lock*

- optional — für Uniprozessorsysteme: Befehle der ISA sind ununterbrechbar
- zwingend — für Multiprozessorsysteme: setzt Zugriffssperre (Datenbus)

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 [3]
- Robustheit: keine hängenden Sperren bei Programmabbrüchen
- in funktionaler Hinsicht wiederverwendbar und komponierbar

cons
- Gefahr von Verhungerung (engl. *starvation*)
- Wiederverwendung sequentieller Altsoftware unmöglich
- Entwicklung nebenläufiger Varianten im Regelfall nicht trivial

Fallstudie: Zählermanipulation

fetch and add — atomare Manipulation eines Speicherzelleninhalts
- Nachbildung von XADD mittels CAS: unteilbares Zählen
  - einer mit 80486 eingeführten ELOP für die x86-Familie
- geeignet zur (blockadefreien) Implementierung von Semaphore

NBS (Abk. für engl. *non-blocking synchronization*)

```
extern int nbs_count (int *, int);          /* fetch and add number */
```

```
int nbs_count (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 */
}
```


Fallstudie: Zählermanipulation (Forts.)

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* ⟺ kein Zugriffskonflikt, neuer Wert gültig
  - *false* ⟺ Zugriffskonflikt, neuer Wert verworfen
- *do ... while* — terminiert nur, falls der neue Wert gebunden wurde

Beachte ↪ ABA-Fall (vgl. S. 6-40)

- die Festellung des Zugriffskonflikts basiert auf eine 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

Fallstudie: Wartestapelmanipulation

LCFS (Abk. für engl. *last come, first served*)

- mögliche Grundlage für stapelorientierte Fadenverarbeitung [7]

  Fadeneinplanung $\mapsto$ Fadenkontrollblock *push*
  Fadeneinlastung $\mapsto$ Fadenkontrollblock *pull*
- atomare Manipulation einer einfach verketteten (LIFO) Liste

NBS — minimale Erweiterung

```
extern void nbs_flush (chain_t *);               /* clear chain */
extern void nbs_ahead (chain_t *, chain_t *);    /* push chain item */
extern chain_t *nbs_strip (chain_t *);           /* pull chain item */
```

```
void nbs_flush (chain_t *this) {
    this->link = 0;
}
```


Fallstudie: Wartestapelmanipulation (Forts.)

Aufnahme in die Liste nach LCFS

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

Entnahme aus der Liste nach LCFS

```
chain_t *nbs_strip (chain_t *this) {
    chain_t *node;

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

    return node;
}
```


Fallstudie: Wartestapelmanipulation (Forts.)

Plausibilitätskontrolle

ahead

- das durch *item* adressierte Element ist noch nicht in der Liste und es wird auch nicht gleichzeitig in diese Liste eingetragen
- kritischer Datenbestand ist *LINK(this)*, der Listenkopf
- dieser wird in *LINK(item)* als Kopie angelegt
- neuer Listenkopf ist *item*, dessen Bindung CAS versucht
- *do ... while* terminiert, falls *item* als Kopf gebunden wurde

strip

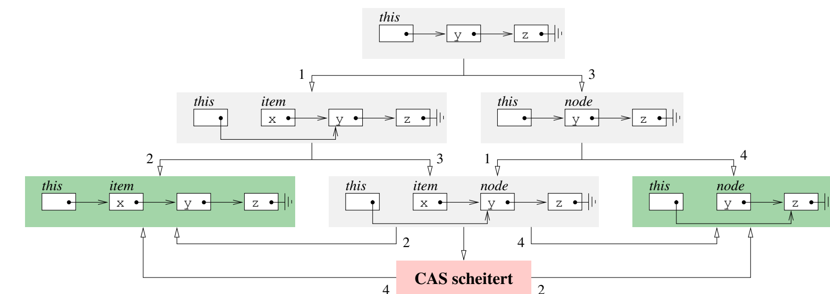
- kritischer Datenbestand ist *LINK(this)*, der Listenkopf
- dieser wird in *node* als (lokale) Kopie angelegt
- neuer Kopf ist *SUCC(node)*, dessen Bindung CAS versucht
- *do ... while* terminiert, falls *SUCC(node)* gebunden wurde

Beachte

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

Fallstudie: Wartestapelmanipulation (Forts.)

Plausibilitätskontrolle: Datenstrukturentwicklung je nach Überlappungsfall




```
void nbs_ahead (chain_t *this, chain_t *item) {
 1  do item->link = this->link;
 2  while (!CAS(&this->link, item->link, item));
}
```

```
chain_t *nbs_strip (chain_t *this) {
    chain_t *node;
 3  do if ((node = this->link) == 0) break;
 4  while (!CAS(&this->link, node, node->link));
    return node;
}
```

Problem ABA

Phänomen der nichtblockierenden Synchronisation auf Basis eines CAS, d.h., einer ELOP, die inhaltsbasiert arbeitet[6]

- 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

[6] Bei *Adressreservierung* wie z.B. mit `LL/SC` besteht dieses Problem nicht.

Problem ABA (Forts.)

Wartestapelmanipulation

Wartestapelzustand: $LINK(this) \rightarrow A \rightarrow B \rightarrow C$

	F_1	F_2	F_2	F_2	F_1
	st...	*strip*	*strip*	*ahead*	*...rip*
$LINK(this)$	A	A	B	C	A
$node_{strip}/item_{ahead}$	A	A	B	A	A
$LINK(node)$	B	B	C	–	B
$\text{CAS}_{LINK(this)}$	...	$A \rightsquigarrow B$	$B \rightsquigarrow C$	$C \rightsquigarrow A$	$A \rightsquigarrow B$

Beachte die Zerstückelung der Liste: $LINK(this) \rightarrow B \rightarrow ? \wedge A \rightarrow C$

- eine Teilliste (A, Kopf) ist aus dem Wartestapel verschwunden
- eine andere Teilliste (B, Kopf) gelangte zurück in den Wartestapel
  - nachdem F_2 Eintrag B mittels *strip* regulär entnommen hatte
  - jedoch ohne dass B danach mittels *ahead* wieder aufgenommen wurde

Problem ABA (Forts.)

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.:

$$sizeof(chain_t) \rightsquigarrow 4 = 2^2 \Rightarrow n = 2$$
$$\Rightarrow chain_t* \text{ ist Vielfaches von } 4$$
$$\Rightarrow chain_t*_{Bits[0:1]} \text{ immer } 0$$

- Platzhalter für n-Bit Marke/Zähler in jedem Zeiger

DCAS
- 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

Problem ABA (Forts.)

Abhilfe (engl. *workaround*)
- bei Software, umgehen unlösbarer Fehler [8]

Beachte $\hookrightarrow$ Generationszähler

- eine Lösung für CAS-artige Verfahren ist nicht wirklich gegeben
- die Wahrscheinlichkeit, dass das Problem auftritt, wird verringert
- Überlappungsmuster haben Einfluss auf den Wertebereich
  - bestimmt durch Zusammenspiel *und* Anzahl der wettstreitigen Fäden
  - ein Bit kann reichen, ebenso, wie ein *unsigned int* zu klein sein kann

Vorbeugung (engl. *prevention*) ist der richtige Ansatz — sofern möglich
- beliebige Überlappungsmuster konstruktiv (Entwurf) ausschließen
- auf Adressreservierungsverfahren der Hardware zurückgreifen
  - unterstützt nicht jede Hardware, ist nur typisch für RISC
  - z.B. CAS mit *load linked*, *store conditional* (LL/SC) implementieren

Fallstudie: Wartestapelmanipulation (Forts.)

Vorbeugung bzw. Abhilfe zum ABA-Problem für die Nachspannliste

Abläufe zur Propagierung von asynchronen Systemsprüngen aufgreifen:

Zurückstellung
- baut die Nachspannliste nur auf, niemals ab
- ruft nur *ahead* auf, niemals *strip*

Freistellung
- baut die Nachspannliste nur ab, niemals auf
- ruft nur *strip* auf, niemals *ahead*

Beachte $\hookrightarrow$ Rechnerbetriebsart

Uniprozessor
- Freistellung schreitet sequentiell voran
- *strip* überlappt nie $\Rightarrow$ kein ABA-Problem

Multiprozessor
- Freistellung schreitet bedingt parallel voran
- *strip* überlappt nur im Falle Zustellergruppe: *ahead*
  - Mehrfachauslösung eines Nachspanns ist kritisch
  - Freiliste $\leadsto$ eindeutiger *tip_t* ∗ $\Rightarrow$ kein ABA-Problem
  - nicht so jedoch bei Ereigniszähler/Bitschalter
  - ABA-Problem $\iff f_{\text{Zurückstellung}(x)} < f_{\text{Freistellung}(x)}$

Fallstudie: Wartestapelmanipulation (Forts.)

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
- in *ahead*
- markieren
- Zeiger färben

index
- in beiden
- bereinigen
- Zeiger liefern

Fallstudie: Wartestapelmanipulation (Forts.)

Lebensdauer des Generationszählers bzw. der Etiketten

Abhilfe gegen des 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 Instanz 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 [9] $\Rightarrow$ Anpassung von NBS

Fallstudie: Wartestapelmanipulation (Forts.)

Etikettierter Zeigertyp *chain_t* ∗

```
void nbs_ahead (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 *nbs_strip (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* ∗

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