Betriebssystemtechnik

Betriebsmittelzugriff

19. Juli 2010

Überblick

Motiv: Betriebsmittelvergabe

Koordinierter Zugriff auf wiederverwendbare/konsumierbare Betriebsmittel

Prozesse benötigen Betriebsmittel verschiedener Art und Anzahl, um weiter voranschreiten zu können

- statischer und dynamischer Arbeitsspeicher, persistenter Speicher
- Prozessor (CPU) und ggf. Koprozessor (FPU, GPU)
- Ein-/Ausgabegeräte
- sowie dazu korrespondierende Datenstrukturen der Software

Lernziel

- Notwendigkeit blockierender Synchronisation
- Vorbeugungsmaßnahme zur unkontrollierten Prioritätsumkehr
- blockadefreie Implementierung blockierender Systemfunktionen
- Belangtrennung bei Prozesssteuerung, -einplanung und -einlastung

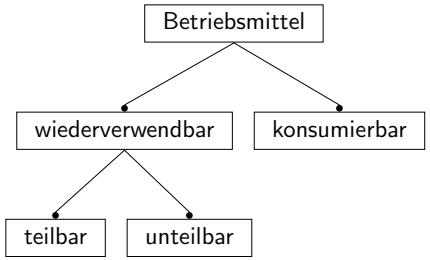
Einordnung

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 [1] bzw. SP [2], BS [3]

Betriebsmittel und Betriebsmittelarten

Wettbewerb um Betriebsmittel (engl. *resource contention*) bezieht sich auf Anzahl und Art eines Betriebsmittels



Betriebsmittelklassen

- Hardware
  - Speicher, Gerät
  - Prozessor(kern)
  - Signal (IRQ)
- Software
  - Puffer, Datei
  - Seite, Prozess
  - Signal, Nachricht

Beachte ↪ Zugriffssteuerung und Betriebsmittelart

einseitige Synchronisation ↦ konsumierbare Betriebsmittel

mehrseitige Synchronisation ↦ wiederverwendbare Betriebsmittel

Autorität von Betriebsmittelvergabe

Ausgewiesene oder (willkürlich) beliebige Instanz?

Betriebsmittel zugeteilt zu bekommen, um als Prozess effektiv und überhaupt Arbeit leisten zu können, ist eine Sache. . .

Zuteilung
- des Prozessors durch den Planer — *scheduler*
- des Busses durch den Schiedsrichter — *arbiter*
- der Kachel durch den Seitenwechsler — *pager*
- ⋮
- eines Datums bei Interprozesskommunikation

. . . ein zugeteiltes Betriebsmittel aber anderen Prozessen eigenmächtig vorzuenthalten, steht auf einem anderen Blatt

- Schutz kritischer Abschnitte durch Aussperrung von Prozessen
- letztlich ein Eingriff in die Autorität zentraler Zuteilungsfunktionen

Betriebsmittelart ⇝ Zugriffsart

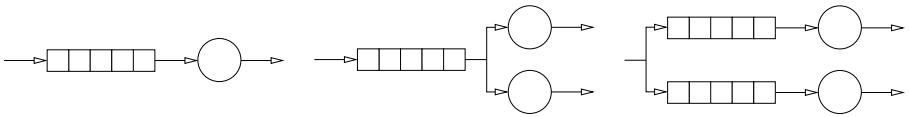
wiederverwendbar ⇌ in der Anzahl (physisch) begrenzt

teilbar
- unkoordinierter Zugriff
- uneingeschränkte Nebenläufigkeit
- ⇒ keine Synchronisation ✓

unteilbar
- koordinierter Zugriff
- eingeschränkte Nebenläufigkeit
- Kontrollflussabhängigkeit
- Wettstreit (*Reader*/*Writer*)
- ⇒ nichtblockierende Synchronisation ✓

konsumierbar ⇌ in der Anzahl (logisch) unbegrenzt
- koordinierter Zugriff
- eingeschränkte Nebenläufigkeit
- Datenflussabhängigkeit
- Kooperation (*Client*/*Server*)
- ⇒ blockierende Synchronisation ✗

Warteschlangen



- eine Warteliste
- ein Bediener
- Uniprozessor

- eine Warteliste
- zwei Bediener
- Multiprozessor
  - symmetrisch
- Verteilung
  - bei Freigabe

- zwei Wartelisten
- zwei Bediener
- Multiprozessor
  - asymmetrisch
- Verteilung
  - bei Ankunft

- eine Hierarchie von Wartelisten *und* Bediener wäre nicht unüblich

Warteschlangen (Forts.)

Bediener
- verarbeitende Komponente eines Wartesystems
  - Prozessor(kern), Gerät, Zusteller, ..., Prozess
- Zuteilung eines Auftrags bedeutet *Einlastung*
  - Mechanismus, abhängig von Bedienerart/-eigenschaften

Warteliste
- buchführende Komponente eines Wartesystems
  - Semaphor, Planer, Lagerhalter, ..., Gerätetreiber
- Aufnahme eines Auftrags geht mit *Einplanung* einher
  - Strategie, abhängig von Bedienerart/-eigenschaften

Beachte ↪ Dilemma
- eine gemeinsame Warteliste erhöht Durchsatz und Wettstreitigkeit
- getrennte Wartelisten senkt Durchsatz und Wettstreitigkeit

⇒ mehrstufige Organisation des Wartesystems:
- untergeordnete Warteliste(n) ▸ Verringerung von Wettstreitigkeit
- übergeordnete Warteliste(n) ▸ Erhöhung von Durchsatz

⇒ Hierarchie verwalteter Betriebsmittel (vgl. auch Kap. 3)

Austausch von Zeitsignalen

Zur Erinnerung: SOS 1 [1] bzw. SP [2]

Semaphor (engl. *semaphore*, [4])
- eine „nicht-negative ganze Zahl“
- für die zwei Elementarperationen definiert sind: P, V

P (hol. *prolaag*, „erniedrige“; auch *down*, *wait*)
- hat der Semaphor den Wert 0, wird der laufende Prozess blockiert
- ansonsten wird der Semaphor um 1 dekrementiert

V (hol. *verhoog*, erhöhe; auch *up*, *signal*)
- inkrementiert den Semaphor um 1
- auf den Semaphor ggf. blockierte Prozesse werden deblockiert

Beachte ↪ Abstrakter Datentyp [5]
- zur Signalisierung von Ereignissen zwischen gleichzeitigen Prozessen
  - deren Ausführung sich zeitlich überschneidet

Fallstudie: Semaphor

EWD — Edsger Wybe Dijkstra

```
#include "luxe/inline.h"
#include "luxe/tune/line.h"

typedef struct semaphore {
    line_t line;         /* optional waitlist: must be first member! */
    int load;
} semaphore_t;

extern void ewd_prolaag (semaphore_t *);
extern void ewd_verhoog (semaphore_t *);

INLINE void P (semaphore_t *this) { ewd_prolaag(this); }
INLINE void V (semaphore_t *this) { ewd_verhoog(this); }
```

```
#ifdef __fame_line_zilch
typedef struct {} line_t;
#endif
```

```
#ifdef __fame_line_chain
#include "luxe/chain.h"
typedef chain_t line_t;
#endif
```

```
#ifdef __fame_line_queue
#include "luxe/queue.h"
typedef queue_t line_t;
#endif
```


Fallstudie: Semaphor (Forts.)

nicht-negative ganze Zahl ▸ im logischen Sinn, Optimierungspotential:

$n \geq 0 \Rightarrow n$ = Anzahl erlaubter gleichzeitiger Prozesse

$n < 0 \Rightarrow |n|$ = Anzahl wartender (d.h., blockierter) Prozesse

```
void ewd_prolaag (semaphore_t *this) {
    if (this->load-- <= 0)
        sad_sleep(&this->line);
}
```

```
void ewd_prolaag (semaphore_t *this) {
    while (this->load-- <= 0) {
        sad_sleep(&this->line);
        this->load++;
    }
}
```

```
void ewd_verhoog (semaphore_t *this) {
    if (this->load++ < 0)
        sad_awake(&this->line);
}
```

```
void ewd_verhoog (semaphore_t *this) {
    if (this->load++ < 0)
        sad_flush(&this->line);
}
```

Hoare'sche Signalisierung
- *awake* wählt einen aus

Hansen'sche Signalisierung
- *flush* wählt alle aus

Fallstudie: Semaphor (Forts.)

Elementaroperation ▶ im logischen Sinn atomar, Synchronisierung:

konventionell ⇒ sperrend/blockierend ✓

unkonventionell ⇒ nichtsperrend/-blockierend ✗

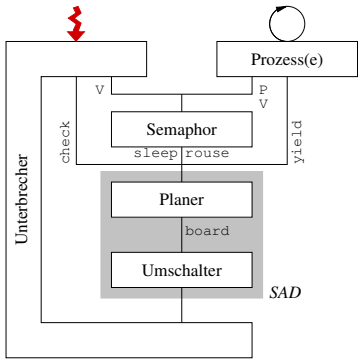
```
void ewd_prolaag (semaphore_t *this) {
    ENTER(ewd);
    ...
    LEAVE(ewd);
}
```

```
void ewd_verhoog (semaphore_t *this) {
    ENTER(ewd);
    ...
    LEAVE(ewd);
}
```

Schutzoptionen

nil	ungültig	
ice	Fortsetzungssperre	✓
irq	Unterbrechungssperre	✓
npx	verdrängungsfreier KA	✗
mux	gegenseitiger Ausschluss	

Infrastruktur zur Ein-/Umplanung: Funktionale Hierarchie



P/*V* ▶ *BM* anfordern/freigeben

sleep ▶ Freigabe von *BM* erwarten
▶ Prozessblockade/-auswahl

rouse ▶ Freigabe von *BM* anzeigen
▶ Modell *awake* oder *flush*

check ▶ Prozessorumplanung prüfen
▶ Zeitgeberunterbrechnung

yield ▶ Prozessen Vortritt gewähren

board ▶ Prozessorumschaltung

Beachte ↪ Leerlaufbetrieb (engl. *idle mode*)

- bei Prozessblockade findet die Auswahl keinen laufbereiten Prozess
- Leerlaufimplementierung durch Befehl oder Programm der Ebene 2
  - z.B. `hlt` (x86) oder aktives Warten auf Bereitlisteneinträge

Infrastruktur zur Ein-/Umplanung (Forts.)

SAD (Abk. für engl. *scheduling and dispatching*)

```
extern void sad_sleep (line_t *);   /* block thread on event */
extern void sad_awake (line_t *);   /* unblock one thread */
extern void sad_flush (line_t *);   /* unblock all threads */
extern void sad_rouse (line_t *);   /* awake or flush threads */

extern void sad_check ();           /* reschedule: asynchronous req. */
extern void sad_yield ();           /* reschedule: synchronous req. */

extern void sad_board (act_t *);    /* dispatch thread of control */
```

```
INLINE void sad_rouse (line_t *this) {
#ifdef __fame_sad_hoare
    sad_awake(this);
#else
    sad_flush(this);
#endif
}
```

Hoare oder Hansen?

- einfache Handhabung?
- robuste Implementierung?

⇝ SOS 1 [1], SP [2]

Ausführungsstrang

Handlung, Vorgang (engl. *act*)

```
enum act_mood { ACT_NONPREEMPTIVE = 0x01, ACT_RELINQUISH = 0x2 };

typedef struct act {
    bid_t task;             /* scheduling state */
    enum act_mood mood;     /* coordination state */
    void *line;             /* blocked-on waitlist resp. signal */
    void *flux;             /* associated coroutine */
} act_t;

extern void   act_ready (act_t *);   /* ready to run, maybe preempt */
extern act_t *act_order ();          /* next ready to run act */

extern act_t *act_being ();          /* current act (on core) */
extern void   act_board (act_t *);   /* dispatch act (on core) */
```

- Spezialisierung eines Auftrags zur Prozessorvergabe: aktives Objekt
- Entkopplung von strategischen Maßnahmen der Prozesseinplanung

Einplanungseinheit

Angebot, Offerte (engl. *bid*) von bereitgestellten Aufträgen

```
enum bid_trim { BID_AVAILABLE, BID_READY, BID_RUNNING, BID_BLOCKED };

typedef struct bid {
    link_t link;          /* optional linkage: must come first! */
    enum bid_trim trim;   /* task state */
    int rank;             /* figure of merit */
} bid_t;

extern void   bid_ready (lot_t *, bid_t *);   /* add bid */
extern bid_t *bid_elect (lot_t *);            /* next bid, if any */
extern bid_t *bid_merit (bid_t *, bid_t *);   /* higher ranked bid */

extern lot_t *bid_hoard ();                   /* pool of bids */
```

- Abstraktion von Art und Erscheinung eines Auftrags: aktiv $\rightleftharpoons$ passiv
- Abstraktion von der Ausprägung einer Bedienstation: CPU/Gerät
- Kapselung rein strategischer Maßnahmen zur Auftragseinplanung

Prozesssteuerung: Bausteine

„schmerzlich“ (engl. *sad*)

Handlung (engl. *act*)

Angebot (engl. *bid*)

Lieferung (engl. *lot*)

Prozesssteuerung: Abstraktionen und Zuständigkeiten

SAD
- Ermöglichung wettlauftoleranter blockierender Funktionen
- Zustandsmaschine zur Steuerung von Ausführungssträngen

ACT
- Einplanung und Einlastung von Ausführungssträngen
- Auftragsspezialisierung für die Bedienstation „CPU“

BID
- Buchführung von Aufträgen an beliebige Bedienstationen
- Umsetzung der jeweiligen Einplanungstrategie

LOT
- Repräsentation und Verwaltung der Auftragsliste
- Abbildung auf statische oder dynamische Datenstrukturen

Beachte $\hookrightarrow$ Variantenvielfalt

SAD
- Arten der Signalisierung blockierter Ausführungsstränge

ACT
- verschiedene Gewichtsklassen von Ausführungssträngen

BID
- unterschiedliche Verfahren zur Auftragseinplanung

LOT
- fallspezifische Auslegung der Auftragsliste(n)

Verdrängungsfreie kritische Abschnitte

NPCS (Abk. für engl. *non-preemptive critical section*)
- Einplanung von Fäden läuft (nahezu) wie gewöhnlich durch
  - ausgelöste Fäden kommen auf die Bereitstellungsliste
  - der laufende Faden bekommt nicht den Prozessor entzogen
- nur die Einlastung von Fäden wird zeitweilig ausgesetzt
  - Verdrängungsereignisse werden zeitversetzt behandelt
  - d.h., fremd- wie auch selbstverursachte Verdrängungen[1]
- zeitweilige Monopolisierung des Prozessors durch einen Faden

Beachte $\hookrightarrow$ Determiniertheit & Verklemmung
- macht die Freigabe unteilbarer Betriebsmittel vorherseh-/sagbar
- macht die Nachforderung unteilbarer Betriebsmittel unteilbar

[1] Eine Verdrängung ist selbstverursacht, wenn der laufende Faden einen Faden höherer Priorität als er selbst bereitstellt. Sie ist fremdverursacht, wenn ein anderer (ggf. externer) Prozess den laufenden Faden dazu zwingt, den Prozessor abzugeben.

Analogie zur Fortsetzungssperre

SAD — minimale Erweiterung

```
extern void sad_avert ();          /* enter non-preemptive section */
extern void sad_admit ();          /* leave non-preemptive section */
extern void sad_waive ();          /* catch up preemption, if any */
```

Verdrängungsfreies *P*

```
void ewd_prolaag (semaphore_t *this) {
    sad_avert();
    if (this->load-- <= 0)
        sad_sleep(&this->line);
    sad_admit();
}
```

Verdrängungsfreies *V*

```
void ewd_verhoog (semaphore_t *this) {
    sad_avert();
    if (this->load++ < 0)
        sad_awake(&this->line);
    sad_admit();
}
```

Beachte ↪ Verdrängungsschutz ≠ Fortsetzungsschutz

- Verdrängungsschutz verzögert lediglich gleichzeitige Prozesse
- Fortsetzungen können nach wie vor zur Ausführung gelangen

⇒ ein *V* ausgelöst vom Unterbrecher kann *P* und *V* überlappen !!!

Verdrängungssteuerung

Verdrängung abwehren

```
void sad_avert () {
    act_t *self = act_being();              /* current thread */

    self->mood |= ACT_NONPREEMPTIVE;        /* processor sharing off */
}
```

Verdrängung zulassen

```
void sad_admit () {
    act_t *self = act_being();              /* current thread */

    self->mood &= ~ACT_NONPREEMPTIVE;       /* processor sharing on */
    if (self->mood & ACT_RELINQUISH)        /* preemption pending? */
        sad_waive();                        /* yes, catch up... */
}
```

- zwischenzeitig eintreffende Prozesse landen auf der Bereitliste
- *waive* lässt zurückgestellte Prozesse ggf. zu (vgl. TIP/ICE *clear*)

Verdrängungssteuerung (Forts.)

Verdrängung ersuchen

```
void sad_check () {
    act_t *self = act_being();              /* current thread */

    if (!(self->mood & ACT_NONPREEMPTIVE))  /* preemption enabled? */
        sad_waive();                        /* yes, reschedule */
    else self->mood |= ACT_RELINQUISH;      /* no, catch up later */
}
```

- verdrängbar zu sein oder nicht, wird als Prozessattribut aufgefasst
- Prozesse werden signalisiert, den Prozessor „freiwillig“ abzugeben

Beachte ↪ Bereitstellung von Prozessen

- seien *this* der bereitgestellte und *self* der aktuelle Prozess
- Verdrängung von *self* $\iff$ $PRIO(this) > PRIO(self)$
- bei Zurückstellung verbleibt *this* schlichtweg auf der Bereitliste

Einsatzbereich

NPCS schützt kritische Abschnitte, die mehr als ein wiederverwendbares unteilbares Betriebsmittel anfordern

- exklusive Belegung nur der CPU allein steht nicht im Vordergrund
- unterbrechungsbedingte Überlappungen des KA sind weiter möglich
- ebenso echte Parallelität: gesonderte Schutzverfahren sind gefordert

Eignung zeigt sich vielmehr in der Vorbeugung von (a) unkontrollierter Prioritätsumkehr und (b) Verklemmungen

(a)
- Betriebsmittel im KA blockiert von Prozess niedriger Priorität
- Zurückstellung unabhängiger Prozesse mittlerer Priorität

⇒ Wartezeitbegrenzung für abhängige Prozesse hoher Priorität [6]

(b)
- Nachforderung von Betriebsmitteln ist unteilbar

⇒ Entkräftung einer der vier Verklemmungsbedingungen [1, 2]

Prozess ∼ Gerichteter Ablauf eines Geschehens

Koordinierung nichtsequentieller Vorgänge auf der Ebene von Prozessen mittels gegenseitigen Ausschluss kennt grundsätzlich zwei Ansätze

(a) Elementaroperationen der Befehlssatzebene (Ebene 2)
- aktives Warten ⇝ blockieren ohne Prozessorabgabe
- typisch für Schloss- oder Sperrvariable (engl. *spinlock*)

(b) Elementaroperationen der Betriebssystemebene (Ebene 3)
- passives Warten ⇝ blockieren mit Prozessorabgabe
- typisch für Semaphor, Barriere

Beachte ↪ Prozesskonzept
- gegenseitiger Ausschluss nimmt gegenseitig verzahnte Prozesse an
- für jeden Prozess gibt es logisch/physisch einen eigenen Prozessor
  - Zeitmultiplexbetrieb schafft logische Prozessoren (Virtualisierung)
  - echter Parallelbetrieb kommt mit physischen Prozessor(kern)en

⇒ gegenseitiger Ausschluss „benutzt“ prozesseigene Prozessoren

Wettlauftoleranter Semaphor

Semaphor, dessen Implementierung gleichzeitige Prozesse zulässt aber dennoch Charakter einer Elementaroperation besitzt

Neuralgische Punkte

```
void ewd_prolaag (semaphore_t *this) {
1   if (this->load-- <= 0)
2       sad_sleep(&this->line);
}

void ewd_verhoog (semaphore_t *this) {
3   if (this->load++ < 0)
4       sad_awake(&this->line);
}
```

Kritische Operationen

1	▶ Zähler erniedrigen
1–2	▶ Bedingung prüfen ▶ Prüfergebnis nutzen
3	▶ Zähler erhöhen
3–4	▶ Bedingung prüfen ▶ Prüfergebnis nutzen

Beachte ↪ Teilbarkeit

1–2	▶ Prozess blockiert, obwohl die Bedingung dazu nicht mehr gilt
3–4	▶ Versuch der Deblockierung, obwohl kein Prozess blockiert ist

Wettlaufsituationen

1–2	▶ F_1 hat 1 passiert und die Blockierungsbedingung festgestellt ▶ zwischen 1 und 2 wird F_1 jedoch unbestimmt lang verzögert ▶ F_2 hat 3 passiert und stellt die Deblockierungsbedingung fest ▶ *awake* erfasst F_1 nicht, da F_1 noch vor dem *sleep* steht ▶ F_1 setzt seine Ausführung fort und „hängt“ sich in *sleep* auf ⇒ vor 1 müsste bekannt sein, worauf F_1 ggf. in *sleep* wartet
3–4	▶ angenommen, es gilt: $load = -1$, d.h., Faden F_x sei blockiert ▶ F_1 liest und merkt sich $load < 0$ in 3: Deblockierungsabsicht ▶ zwischen 3 und 4 wird F_1 jedoch unbestimmt lang verzögert ▶ F_2 passiert 3 und 4: $load = -1 \mapsto load = 0$, F_x deblockiert ▶ F_1 setzt seine Ausführung fort und passiert ebenfalls 4 ⇒ schlimmstenfalls Mehraufwand für nichts und wieder nichts

Beachte ↪ Wettlaufsituation 1–2: „*lost wake-up*“-Problem
- Auflösung von *sleep*: 1. Blockierungsabsicht und 2. Blockierung

Nebenläufigkeit nicht einschränkender Semaphor

SAD — minimale Erweiterung (Forts.)

```
extern void sad_agree (line_t *);     /* allow for blocked-on event */
extern void sad_forgo ();             /* block on event agreed upon */
extern int  sad_annul ();             /* clear blocked-on event */
```

Wettlauftolerantes P/V

```
void ewd_prolaag (semaphore_t *this) {
    sad_agree(&this->line);
    if (ami_lower(&this->load) <= 0)
        sad_forgo();
    else if (sad_annul())
        sad_awake(&this->line);
}

void ewd_verhoog (semaphore_t *this) {
    if (ami_raise(&this->load) < 0)
        sad_awake(&this->line);
}
```

Elementaroperationen

lower	▶ dekrementieren
raise	▶ inkrementieren

▶ *fetch and add*, FAA

Neuralgische Punkte
1. nach *agree*
2. vor *forgo*
3. vor *annul*

Wettlaufsteuerung: Neuralgische Punkte

Problem (1. Verzögerung nach *agree*)

- *für Ausführungsstrang F_1 gilt: line $\neq 0 \wedge$ trim $=$ BID_RUNNING*
- *F_1 ist in Blockadeabsicht, kann durch F_2 deblockiert werden*

$\Rightarrow$ *ein noch laufender Faden kann auf die Bereitliste kommen*

Beachte $\hookrightarrow$ Analogie zum Leerlaufbetrieb

- zum Blockierungszeitpunkt von Faden F_x sei die Bereitliste leer
- für F_x gilt: *trim $=$ BID_BLOCKED $\wedge$ flux* $\mapsto$ Standlauf (engl. *in idle*)
- d.h., F_x geht ins aktive und ggf. auch passive Warten über
  - aktiv
    - solange leer laufen, bis die Bereitliste wieder gefüllt wurde
    - Bedingung: Aktion „von außen“, die einen Faden auslöst !!!
  - passiv
    - zwischen zwei Abfragen der Bereitliste den Prozessor anhalten
    - kritischer Abschnitt $\rightsquigarrow$ Unterbrechungssperre

$\Rightarrow$ F_x selbst könnte auf die Bereitliste gelangen, obwohl er läuft

Wettlaufsteuerung: Neuralgische Punkte (Forts.)

Problem (2. Verzögerung vor *forgo*)

- *für Ausführungsstrang F_1 gilt zusätzlich zu 1.: load ≤ 0*
- *F_1 wird blockieren, steht dann ggf. aber bereits auf der Bereitliste*

$\Rightarrow$ *ein sich blockierender Faden wird sein Aufwecksignal nicht verlieren*

Beachte $\hookrightarrow$ Verdrängung bzw. Multiprozessorbetrieb

(a)
- angenommen, F_1 wird verdrängt und F_2 erhält den Prozessor
- verdrängte Fäden kommen auf die Bereitliste, so dann auch F_1

$\Rightarrow$ ein *awake* durch F_2 könnte F_1 abermals auf diese Liste setzen

(b)
- angenommen, F_2 eines anderen Prozessors durchläuft *awake*
- als Folge davon kann F_1 (laufend) auf die Bereitliste kommen

$\Rightarrow$ Verdrängung könnte dann F_1 abermals auf diese Liste setzen

Wettlaufsteuerung: Neuralgische Punkte (Forts.)

Problem (3. Verzögerung vor *annul*)

- *für Ausführungsstrang F_1 gilt zusätzlich zu 1.: load > 0*
- *F_1 hat ein Zeitsignal konsumiert, wird ggf. einen KA betreten*

$\Rightarrow$ *ein weiteres Aufweckereignis kann diesen Faden erneut signalisieren*

Beachte $\hookrightarrow$ Zählender Semaphor

- angenommen, F_2 zeigt durch *awake* ein neues Aufweckereignis an[a]
- anstatt einen blockierten Faden bereitzustellen, wird F_1 signalisiert

$\Rightarrow$ ein Zeitsignal geht verloren, sollte F_1 erneut signalisiert werden

[a] Ein binärer Semaphor dürfte von F_2 so nicht benutzt werden. Das würde nämlich bedeuten, dass F_2 einen von ihm nicht betretenen KA freigibt. Als Abhilfe ist bekannt, den Besitzer des binären Semaphors in P zu verbuchen und in V zu prüfen: *Mutex*.

Wettlaufsteuerung: Lösungsansätze

Problem 1
- ist ähnlich zum Problem des Leerlaufbetriebs zu lösen
- nimmt sich F_1 selbst von der Bereitliste: weiterlaufen
- wird F_1 vom anderen Prozessor erfasst: überspringen

Problem 2
- verhindern, dass F_1 mehrfach auf die Bereitliste kommt
  - damit wäre Problem 1 automatisch gleich mit gelöst
- Verdrängungssperre zwischen *agree* und *forgo*/*annul*
- in *annul* ggf. der Verdrängungsaufforderung nachkommen

Problem 3
- verhindern, dass F_1 eine Zeitsignalanzeige verpasst
  - damit einem möglichen „*lost wake-up*“ vorbeugen
- durch *awake* bereitgestellte Fäden sind „aufgeweckt“
- in *annul* dieses Attribut prüfen und ggf. *awake* aufrufen

Beachte $\hookrightarrow$ Wettlaufsituationen

- *annul* muss die Blockadeabsicht des laufenden Fadens löschen
  - erst dannach sind die ihm ggf. zugestellten Attribute zu überprüfen

Tücke im Detail

Anmerkung (Blockierungsabsicht eines Faden)

1. *der Faden muss an einer eindeutigen Wartebedingung gebunden sein*
   - *idealerweise durch einen (logischen/physikalischen) Adresswert*
   - *z.B. den einer Ereignisvariablen, Warteliste oder eines Semaphors*
2. *der Faden muss (logisch/physisch) auf einer Warteliste verbucht sein*

Beachte ↪ Implementierung der Warteliste

statisch
- alle Einträge der Fadentabelle derselben Wartebedingung
- implizite Verbuchung bei Umsetzung von Bedingung 1
- ⇒ Adresse atomar in *agree* setzen, in *annul*/*awake* löschen

dynamisch
- Wartestapel oder -schlange, einfach/doppelt verkettet
- explizite Verbuchung durch Fadenverkettung (*act*)
- ⇒ Faden atomar in *agree* ein-, in *annul*/*awake* austragen

Blockadeabsicht vereinbaren/zurücknehmen

```
void sad_agree (line_t *line) {
    act_t *self = act_being();

    self->mood = (ACT_NONPREEMPTIVE | ACT_UNSHIFTABLE);
    act_stick(line, self);              /* add to waitlist */
}
```

```
int sad_annul () {
    act_t *self = act_being();

    act_purge(self->line, self);        /* remove from waitlist */
    self->mood &= ~(ACT_NONPREEMPTIVE | ACT_UNSHIFTABLE);

    if (self->mood & ACT_RELINQUISH)    /* interim preemption signal? */
        sad_waive();                    /* yes, be responsive... */

    return self->mood & ACT_AROUSED;    /* interim wake-up signal? */
}
```


Blockade auflösen

```
void sad_awake (line_t *line) {
    act_t *next;

    if ((next = act_unbag(line))) {     /* next from waitlist? */
        next->mood |= ACT_AROUSED;      /* yes, note wake-up call */
        act_ready(next);                /* set ready to run */
    }
}
```

Beachte ↪ Multiprozessorbetrieb

- ein Faden wird zwischen *agree* und *forgo*/*annul* nicht verdrängt
- kein anderer desselben Prozessors kann ihn der Warteliste entnehmen
- wohl aber einer eines anderen Prozessors ⇝ Problem 1
  - der entnommene, ggf. noch laufende Faden kommt auf die Bereitliste
  - er kann von dieser durch einem anderen Prozessor entnommen werden
  - darf aber nicht eingelastet werden, sollte er wirklich noch laufen...

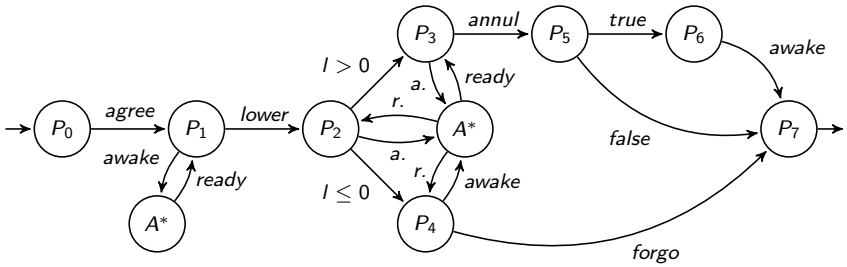
Blockadeabsicht verfolgen: Prozessor ggf. umschalten

```
void sad_forgo () {
    act_t *next, *self = act_being();

    while ((next = act_order())) {          /* next from ready list */
        int tied = next->mood & ACT_UNSHIFTABLE;
        next->mood &= ~(ACT_NONPREEMPTIVE | ACT_UNSHIFTABLE);
        if (next == self) return;           /* running thread? */
        if (!tied) break;                   /* core-bound? */
        next->mood |= ACT_RELINQUISH;       /* yes, send signal */
    }
    assert(next);                           /* there will be a next! */

    self->mood &= ~ACT_RELINQUISH;          /* will release core */
    act_board(next);                        /* dispatch next thread */
}
```

Beachte ↪ Multiprozessorbetrieb

- unbewegliche Fäden werden nicht von ihrem Prozessor „gezogen“
  - sie werden aufgefordert, die Kontrolle über ihren Prozessor abzugeben

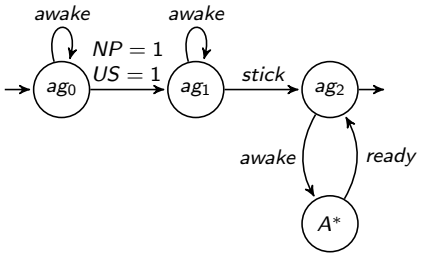
Plausibilitätskontrolle: P



- P_0 P aufgerufen
- P_1 unverdrängbar, unbeweglich, aufweckbar
- P_2 Zeitsignal angefordert: *load* erniedrigt
- P_3 Zeitsignal verfügbar: nicht blockieren
- P_4 Zeitsignal nicht verfügbar: blockieren
- P_5 nicht aufweckbar, verdrängbar, beweglich
- P_6 „*lost wake-up*“ festgestellt
- P_7 Zeitsignal konsumiert: P verlassen

A^* überlappendes *awake* (vgl. S. 7-39): kritisch, kann „*lost wake-up*“-Problem verursachen
- für P_0 und ab P_5 unkritisch, zu beachten dazwischen
- der Zweck von P_1 ist, A^* kontrollierbar zu machen

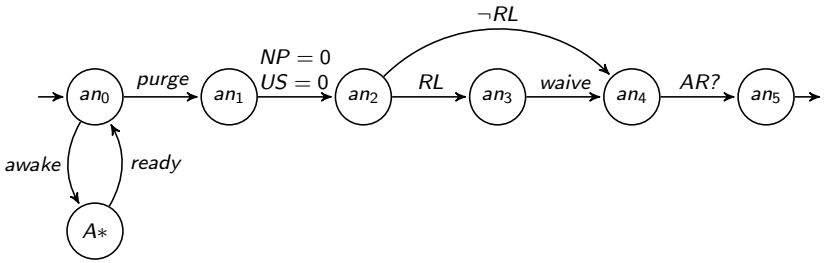
Plausibilitätskontrolle: $P_0 \mapsto P_1$, *agree*



- ag_0 agree aufgerufen
- ag_1 Faden unverdrängbar und unbeweglich
- ag_2 Faden auf Warteliste, *awake* verlassen

A^* überlappendes *awake* (vgl. S 7-36)
- stellt laufenden Faden ggf. bereit
- nur durch einen anderen Prozessor
- zieht Faden jedoch nicht hinüber

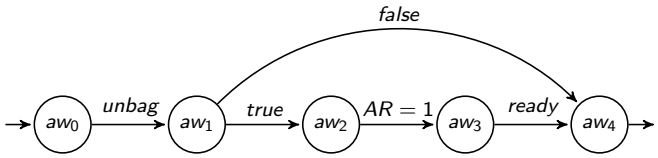
Plausibilitätskontrolle: $P_3 \mapsto P_5$, *annul*



- an_0 *annul* aufgerufen
- an_1 Faden von Warteliste gestrichen
- an_2 Faden verdrängbar und beweglich
- an_3 CPU aufgeben (engl. *relinquish*, *RL*)
- an_4 *true* $\iff$ „*lost wake-up*“, *false* sonst
- an_5 *annul* verlasssen

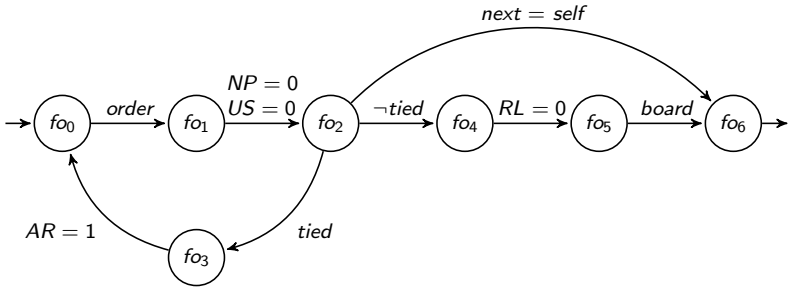
A^* überlappendes *awake* (vgl. S. 7-36):
- kann nur vor an_1 noch zum „*lost wake-up*“-Problem führen
- nämlich solange der laufende Faden nicht von der Wartliste gelöscht ist

Plausibilitätskontrolle: A^*, *awake*



- aw_0 *awake* aufgerufen
  - es musste gelten *load* < 0, d.h., wenigstens ein Faden müsste auf *line* warten
- aw_1 *unbag* ausgeführt, ggf. einen Faden von der Warteliste gestrichen
  - *awake* eines anderen Prozessor(kern)s hat die Warteliste ggf. gelöscht $\rightsquigarrow$ *false*
- aw_2 exakt einen Faden aufgeweckt, sein Wartezustand aufgehoben
  - der Faden steht nicht mehr auf der Warteliste
- aw_3 Faden als „aufgeweckt“ (engl. *aroused*, *AR*) ausgezeichnet: bereitstellen
  - ein Fadenattribut, um in *annul* ein „*lost wake-up*“ erkennen zu können
- aw_4 *awake* verlassen

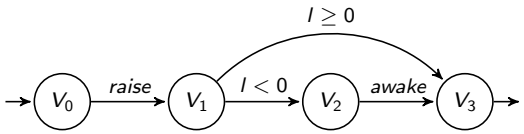
Plausibilitätskontrolle: $P_4 \mapsto P_7$, *forgo*



- fo_0 *forgo* aufgerufen
- fo_1 Faden von Bereitliste gestrichen
- fo_2 Faden verdrängbar und beweglich
- fo_3 fremder Faden, anderen auswählen
- fo_4 Faden einlastbar auf Prozessor
- fo_5 Faden gibt Prozessor(kern) auf
- fo_6 *forgo* verlasssen

fo_2 Standlauffaden $\iff$ *next* = *self*, fremder Faden $\iff$ unbeweglich

Plausibilitätskontrolle: V



- V_0 V aufgerufen
- V_1 ein Zeitsignal wurde produziert: *load* wurde um einen Zähler erhöht
- V_2 wenigstens ein Faden müsste bereit sein, ein Zeitsignal zu konsumieren
- V_3 V verlassen

Ausführungsstrang (Forts.)

ACT — minimale Erweiterung

```
enum act_mood { /* ... */ ACT_AROUSED = 0x4, ACT_UNSHIFTABLE = 0x8 };

extern void   act_stick (line_t *, act_t *);   /* put on waitlist */
extern void   act_purge (line_t *, act_t *);   /* kill from waitlist */
extern act_t *act_unbag (line_t *);            /* next from waitlist */
```

- die Schnittstelle verbirgt die Implementierung einer Warteliste
  - Tabelle
    - statische Datenstruktur: Prozess- bzw. Fadentabelle
    - ggf. kombiniert mit Verkettungen (Streuspeicher, *hash table*)
  - Kette
    - dynamische Datenstruktur: fallspezifische verkettete Liste
    - Stapel (LIFO), Schlange (FIFO), Baum
- Attribute dienen der Wettlaufsteuerung gleichzeitiger Prozesse
  - zur Erfassung aufgeweckter und unbeweglicher Ausführungsstränge
  - zur Koordinierung der Aktivitäten zur Einplanung/-lastung von Fäden

Datenstrukturen für Warte- und Bereitliste

Liste (von it. *lista*: Leiste, Papierstreifen) bezeichnet ein bestimmtes Verzeichnis oder generell eine Form von Verzeichnisstruktur [7]

- die Implementierung von Prozesslisten ist fallspezifisch auszulegen
  - immer nur eine dynamische Datenstruktur anzunehmen, wäre falsch
  - eine statische Datenstruktur käme ebenso gut in Frage
- Tabellen, verkettete Strukturen, Kombinationen davon, ...

Beachte ↪ Zielkonflikt bzw. Kompromiss

Tabelle	+	Eintrag aufnehmen/löschen: einfache Schreiboperation
	+	Eintrag atomar auswählen: ein einfaches CAS
	−	nächsten Eintrag suchen: „komplexe“ Programmschleife
Kette	−	Eintrag aufnehmen/löschen: „komplexe“ CAS-Konstruktion
	−	Eintrag atomar auswählen: „komplexe“ CAS-Konstruktion
	+	nächsten Eintrag suchen: einfach Verkettung verfolgen

Datenstrukturen für Warte- und Bereitliste (Forts.)

Beispiel: Fallspezifische Auslegung der Warteliste

```
void act_stick (line_t *line, act_t *this) {
    this->line = line;    /* bind to event */
#ifdef __fame_line_chain
    nbs_ahead(line, &this->task.link);
#endif
#ifdef __fame_line_queue
    nbs_aback(line, &this->task.link);
#endif
}
```

```
void act_purge (line_t *line, act_t *this) {
#ifdef __fame_line_chain
    nbs_erase(line, &this->task.link);
#endif
#ifdef __fame_line_queue
    nbs_purge(line, &this->task.link);
#endif
    this->line = 0;       /* unbind from event */
}
```

```
act_t *act_unbag (line_t *line) {
    act_t *next;
#ifdef __fame_line_zilch
    next = lot_clear(line);
#else
#ifdef __fame_line_chain
    next = nbs_strip(line);
#endif
#ifdef __fame_line_queue
    next = nbs_fetch(line);
#endif
    next->line = 0;
#endif
    return next;
}
```

zilch ▶ Tabelle (FCFS)

chain ▶ Stapel (LCFS)

queue ▶ Schlange (FCFS)

Beachte ↪ Variantenbildung: Eingang zur `#ifdef`-Hölle [8]

- Instanzenbildung verschiedener Aspekte durch Entmischung

Datenstrukturen für Warte- und Bereitliste (Forts.)

Beispiel: Wettlauftolerante Auswahl eines zu deblockierenden Fadens

```
extern lot_t lot_store[];                /* act table */

lot_t *lot_clear (line_t *line) {
    lot_t *next;

    for (next = &lot_store[0]; next < &lot_store[N_ACT]; next++)
        if (CAS(&next->line, line, 0))   /* act blocked-on line? */
            return next;                 /* yes, condition cleared */

    return 0;                            /* nothing found, fail... */
}
```

pros
- sehr einfache Lösung im Vergleich zu Verkettungsstrukturen
- prioritätsorientierte Fadenauswahl — und -bereitstellung
- ansteigende Indexwerte ↦ absteigende Priorität

cons
- skaliert schlecht mit zunehmender Tabellenlänge/Fadenanzahl

Querschneidende Belange

Qual der Wahl — welche nichtfunktionale Eigenschaften das System zur Steuerung von Betriebsmittelzugriffen mitbringen soll

Determiniertheit
- wartefreie Lösungsvarianten präsentieren
- Tabelle ☺, Kette ☹

Skalierbarkeit
- dynamische Datenstrukturen verwenden
- Kette ☺, Tabelle ☹

Performanz
- auf bedingungsspezifische Wartelisten setzen
- Kette ☺, Tabelle ☹

Lokalität
- minimal-invasiv auf Zwischenspeicher einwirken
- Tabelle ☺, Kette ☹

Beachte ↪ Aspektgewahre Systemsoftware [9]

- ob Tabelle oder Kette: die Entscheidung für das eine oder andere betrifft viele Stellen in der Software
- darauf müssen Entwurf *und* Implementierung Rücksicht nehmen

Nachtrag...

AMI (Abk. für engl. *atomic machine instruction*)

```
extern int ami_lower (int *);    /* decrement, deliver new value */
extern int ami_raise (int *);    /* increment, deliver new value */
```

```
int ami_lower (int *this) {
    return faa(this, -1);
}

int ami_raise (int *this) {
    return faa(this, 1);
}
```

```
#include "luxe/fame/smp.h"

#ifdef __fame_smp
#define LOCK_PREFIX "lock\n\t"
#else
#define LOCK_PREFIX
#endif
```

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

    __asm__ __volatile__(
        LOCK_PREFIX
        "xadd %0,%1"
        : "=q" (aux), "=m" (*ref)
        : "q" (aux), "m" (*ref)
        : "memory");

    return aux;
}
```

Resümee

Betriebsmittelart ⟷ Zugriffsart
- wiederverwendbar (begrenzt), konsumierbar (unbegrenzt)
- teilbar, unteilbar

Warteschlangen ⟷ Hierarchie
- Uni-/Multiprozessor, ein/mehr Bediener
- symmetrisch/asymmetrisch, Verteilung bei Freigabe/Ankunft

Zeitsignale ⟷ abstrakter Datentyp
- logisch: nicht-negative ganze Zahl, Elementaroperation
- semaphoreigene/planerverwaltete Warteliste

Verdrängungssperre ⟷ NPCS
- Abgrenzung zur Fortsetzungssperre
- Verdrängungssteuerung, Haupteinsatzbereich

Vorgangssperre ⟷ wettlauftoleranter Semaphor
- Wartelistenkonzept, neuralgische Punkte
- Zustandsmaschine, Zustandsautomaten

Literaturverzeichnis

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

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

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

Literaturverzeichnis (Forts.)

[4] Edsger Wybe Dijkstra.
Cooperating sequential processes.
Technical report, Technische Universiteit Eindhoven, Eindhoven, The Netherlands, 1965.
(Reprinted in *Great Papers in Computer Science*, P. Laplante, ed., IEEE Press, New York, NY, 1996).

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

[6] Wolfgang Schröder-Preikschat.
Echtzeitsysteme.
http://www4.informatik.uni-erlangen.de/Lehre/EZS, 2005.

Literaturverzeichnis (Forts.)

[7] Wikipedia Foundation Inc.
Wikipedia, Die freie Enzyklopädie.
http://de.wikipedia.org.

[8] Daniel Lohmann, Wanja Hofer, Wolfgang Schröder-Preikschat, Jochen Streicher, and Olaf Spinczyk.
CiAO: An aspect-oriented operating-system family for resource-constrained embedded systems.
In *Proceedings of the 2009 USENIX Technical Conference*, pages 215–228, Berkeley, CA, USA, June 2009. USENIX Association.

[9] Daniel Lohmann.
Aspect Awareness in the Development of Configurable System Software.
PhD thesis, Friedrich-Alexander University Erlangen-Nuremberg, 2009.