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...

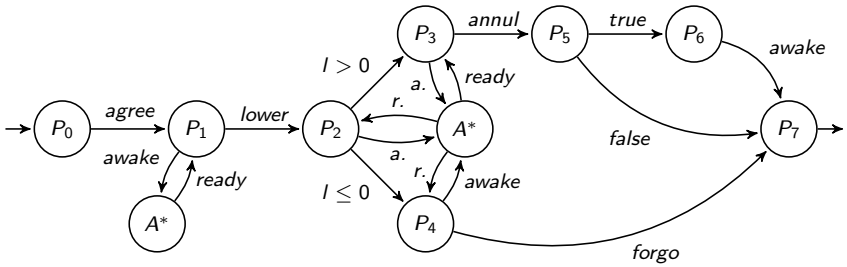
Blockadeabsicht verfolgen: Prozessor ggf. umschalten

```
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

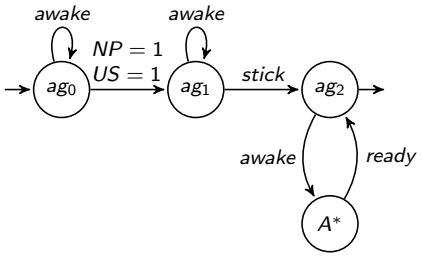
Plausibilitätskontrolle: P



- 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

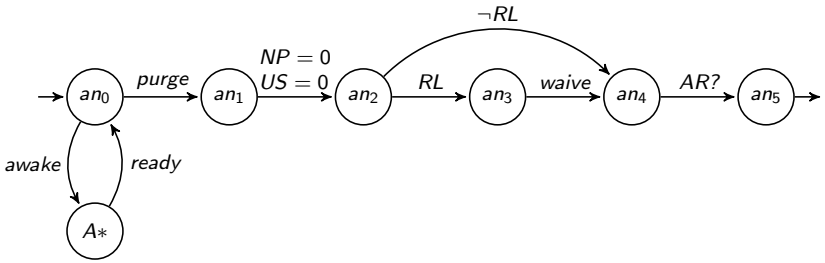
Plausibilitätskontrolle: $P_0 \mapsto P_1$, *agree*



- 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

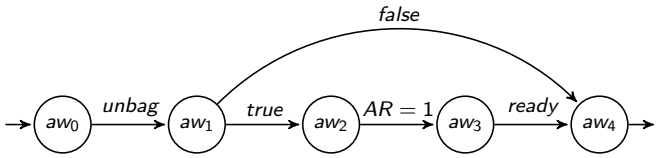
Plausibilitätskontrolle: $P_3 \mapsto P_5$, *annul*



- 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

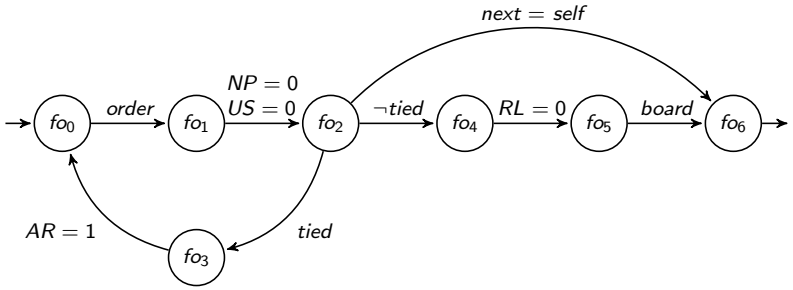
A^* überlappendes *awake* (vgl. S. 7-36):
- kann nur vor an_1 noch zum „*lost wake-up*“-Problem führen
- nämlich solange der laufende Faden nicht von der Wartliste gelöscht ist

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

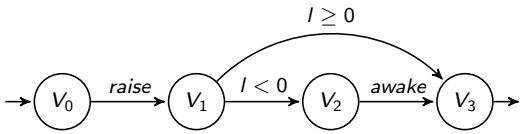
Plausibilitätskontrolle: $P_4 \mapsto P_7$, *forgo*



- 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.