Betriebssystemtechnik

Prozesssteuerung: Betriebsmittelzugriff

Wolfgang Schröder-Preikschat

Lehrstuhl Informatik 4

3. Juli 2012

Gliederung

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
- blockadefreie Implementierung blockierender Systemfunktionen
- Belangtrennung bei Prozesssteuerung, -einplanung und -einlastung

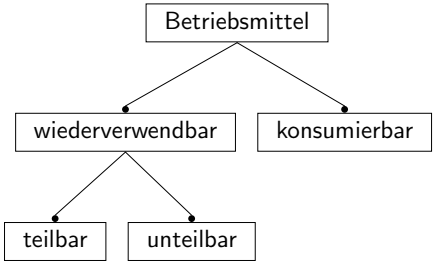
Einordnung

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 [8] bzw. SP [9], BS [7]

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

Betriebsmittel (die zugeteilt wurden) aber anderen Prozessen eigenmächtig und unnötig 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 (*Producer*/*Consumer*)
- ⇒ **blockierende Synchronisation** ✗

Verwaltung zählbarer Betriebsmittel: Semaphor

- *The semaphores are essentially non-negative integers;*
  - *when only used to solve the mutual exclusion problem, the range of their values will even be restricted to "0" and "1".*
- *It is [Scholten] to have demonstrated a considerable field of applicability for semaphores that can take on larger values.*
- *When there is a need for distinction, we shall talk about "binary semaphores" and "general semaphores" respectively.*
- *The definition of the P- and V-operation [...] is insensitive to this distinction.*

(Dijkstra, 1965 [3, S. 28/29])

Prozesse werden bei Nichtverfügbarkeit von Betriebsmitteln schlafen gelegt

- sie warten passiv auf die Bereitstellung angeforderter Betriebsmittel
- oft Wartelisten bildend, bei **Wettbewerb** gleichzeitiger Prozesse

Semaphor: In Anlehnung an [2] und [4, S. 345]

P (hol. *prolaag*, „erniedrige“; auch *down*, *wait*)
- erniedrigt den Wert des Semaphors um 1
- wenn der resultierende Wert negativ ist, wird der Prozess gestoppt und auf eine mit dem Semaphor verbundene Warteliste verbucht

V (hol. *verhoog*, erhöhe; auch *up*, *signal*)
- erhöht den Wert des Semaphors um 1
- wenn der resultierende Wert nichtpositiv ist, wird einer der auf dieser Warteliste verbuchten Prozesse von dieser entfernt

Beachte ↪ „Warteliste“ sagt nichts über die Realisierung aus
- eine Option ist die Implementierung als **dynamische Datenstruktur**
  - einfach/doppelt verkettet, ggf. als Schlange organisiert
- eine andere Option wäre als **Funktion**, zur Auflistung der „Schläfer“
  - an Hand einer Marke, etwa der Speicheradresse des Semaphors

Semaphor: Zweck und Varianten

Semaphore dienen dem Austausch von Zeitsignalen zur Kooperation von Prozessen, zur Signalisierung von Ereignissen

binärer ∼
- mit Wertebereich $[N, 1]$, wobei $N \leq 0$ [1]
- zum wechselseitigen Ausschluss gleichartiger Prozesse
  - KA betretender ↦ P, KA verlassender ↦ V
- V impliziert ein vorangegangenes P vom selben Prozess

allgemeiner ∼
- mit Wertebereich $[N, M]$, wobei $N \leq 0$ [1] und $M > 0$
- zur Bedingungssynchronisation ungleichartiger Proz.
  - Konsument ↦ P, Produzent ↦ V
- zum V muss zuvor kein P erfolgt sein, gleich welch ein Prozess auch P und V ausführt oder ausführen wird

Beachte ↪ Schutz kritischer Abschnitte mittels (bin.) Semaphore
- V ist von dem Prozess auszuführen, der zuvor P ausgeführt hatte
- P definiert (logisch) eine Eigentümerschaft, die für V gelten muss

[1] Negative Werte geben Hinweis auf die Anzahl der Schläfer [4, S. 345].

Semaphor: Schnittstelle

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

typedef struct semaphore {
    line_t line;         /* optional waitlist: must be first member! */
    int load;            /* binary (EWD) or nonbinary (CSS) counter */
} semaphore_t;

extern void ewd_prolaag(semaphore_t *);  /* EWD - Edsgar Wybe Dijkstra */
extern void ewd_verhoog(semaphore_t *);  /*       binary semaphore     */
extern void css_prolaag(semaphore_t *);  /* CSS - Carel S. Scholten    */
extern void css_verhoog(semaphore_t *);  /*       nonbinary semaphore  */

INLINE void P(semaphore_t *this) { css_prolaag(this); }
INLINE void V(semaphore_t *this) { css_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
```


Gliederung

Kritischer Abschnitt (KA) als sequentielles Programm

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

```
void ewd_verhoog(semaphore_t *this) {
    next = sad_waken(&this->line);
    if (next)
        sad_ready(next);
    else
        this->load = CS_IDLE;
}
```

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

```
void ewd_verhoog(semaphore_t *this) {
    this->load = CS_IDLE;
    next = sad_waken(&this->line);
    if (next)
        sad_ready(next);
}
```

Beachte ↪ Wettlaufsituationen

- mehrere Prozesse können denselben KA gleichzeitig durchlaufen
- Prozesse werden daran gehindert, einen freien KA zu betreten

⇒ P und V bilden jeweils selbst einen KA, der geeignet zu schützen ist

P und V als Elementaroperation (ELOP)

(a) **unteilbares Maschinenprogramm**

- gleichzeitige Prozesse der Maschinenprogrammebene verhindern ✓

```
void P(semaphore_t *this) {
    ENTER(opt);
    ...
    LEAVE(opt);
}
```

```
void V(semaphore_t *this) {
    ENTER(opt);
    ...
    LEAVE(opt);
}
```

... ewd_{prolaag,verhoog} (S. 13)

Schutzoptionen: Werte von opt

nil	ungültig
irq	Unterbrechungssperre
ice	Fortsetzungssperre
npx	Verdrängungssperre
mux	Umlaufsperre

(b) **unteilbarer Maschinenbefehl**

- gleichzeitige Prozesse der Maschinenprogrammebene zulassen ✗
  - dazu jedoch gleichzeitige Prozesse der Befehlssatzebene verhindern

P und V (binär) als nichtsequentielles Programm

Überlappungsmuster: seien P_b und V_b die Elementaroperationen eines binären Semaphors, der nur zum Schutz kritischer Abschnitte dient

1. P_b überlappt P_b
   - genau nur einer von gleichzeitigen Prozessen, die P_b betreten haben, darf P_b verlassen; die anderen müssen in P_b blockieren
2. P_b überlappt V_b
   - V_b darf gleichzeitige Prozesse, die P_b betreten haben und ausführen, nicht unnötig lange verzögern
3. V_b überlappt P_b
   - ein P_b durchlaufender Prozess darf das durch V_b eines gleichzeitigen Prozesses erzeugte Wecksignal nicht verpassen
4. V_b überlappt V_b
   - V_b darf einen durch P_b blockierten Prozess nur einmal aus seinem Schlafzustand erlösen

Beachte ↪ Semantik zum Schutz kritischer Abschnitte

- V_b wird von dem Prozess ausgeführt, der zuvor P_b ausgeführt hat

P und V (binär) als nichtsequentielles Programm: Die 1.

test and set (vgl. Kap. VI-1, S. 15/x86) & *reset atomic flag*

```
INLINE int  TAS(int *flag) { return tas(flag); }  /* CS_BUSY?! */
INLINE void RAF(int *flag) { *flag = 0; }         /* CS_IDLE ! */
```

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

```
void ewd_verhoog(semaphore_t *this) {
    next = sad_waken(&this->line);
    if (next)
        sad_ready(next);
    else
        RAF(&this->load);
}
```

1 ✓ 4 ✓ atomares *waken*

Beachte ↪ V ist ein sequentielles Programm

- aufwecken und ggf. bereitstellen des nächsten wartenden Prozesses in V geschieht, während der Semaphor noch aktiv ist
- der kritische Abschnitt (KA), dessem Schutz der Semaphor dient, wird durch V (a) stark verlängert und (b) nur bedingt verlassen

P und V (binär) als nichtsequentielles Programm: Die 2.

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

```
void ewd_verhoog(semaphore_t *this) {
    RAF(&this->load);
    next = sad_waken(&this->line);
    if (next)
        sad_ready(next);
}
```

❶ ✓ ❷ ✓ ❹ ✓

Beachte ↪ Wettlaufsituation

- sofort nach Freigabe des KA durch V, kann ein überlappender Prozess P erfolgreich durchlaufen und den KA betreten
- holt V nebenläufig dazu einen Prozess aus dem Schlafzustand, können sich zwei Prozesse gleichzeitig im KA befinden
  - der aufgeweckte Prozess kehrt aus *sleep* zurück, der ungehindert weiter voranschreitet, P verlässt und den KA (ebenfalls) betritt
  - obwohl ggf. die Wartebedingung für ihn und andere wieder gilt

⇒ P muss wiederholt versuchen, den Semaphor zu belegen

P und V (binär) als nichtsequentielles Programm: Die 3.

```
void ewd_prolaag(semaphore_t *this) {
    while (TAS(&this->load))
        sad_sleep(&this->line);
}
```

```
void ewd_verhoog(semaphore_t *this) {
    RAF(&this->load);
    next = sad_waken(&this->line);
    if (next)
        sad_ready(next);
}
```

❶ ✓ ❷ ✓ ❹ ✓

Beachte ↪ Wettlaufsituation

- ein in P vor betreten seines Schlafzustands verdrängter Prozess, kann seinen Weckruf verpassen: (engl.) *lost wake-up*
  - wenn der verdrängende Prozess V auf denselben Semaphor anwendet
- Überprüfen des Semaphorwertes und schlafen legen des prüfenden Prozesses zusammen bilden selbst einen kritischen Abschnitt
  - der (P durchlaufend) prüfende Prozess stellt fest, dass der Semaphor aktiv ist und wird daher *sleep* (nebenläufig zu V) aufrufen
  - obwohl ggf. die Wartebedingung für ihn und andere nicht mehr gilt

⇒ der P gerade durchlaufende Prozess muss durch V aufspürbar sein

P und V (binär) als nichtsequentielles Programm: Die 4.

Prozesszustandsüberwachung (*process state inspection*, PSI) zwischen Semaphor- und Einplanungsebene einführen:

```
void ewd_prolaag(semaphore_t *this) {
    psi_snoop(&this->line);
    while (TAS(&this->load))
        psi_doubt(&this->line);
    psi_clean(&this->line);
}
```

```
void ewd_verhoog(semaphore_t *this) {
    RAF(&this->load);
    next = psi_trace(&this->line);
    if (next)
        sad_ready(next);
}
```

❶ ✓ ❷ ✓ ❸ ✓ ❹ ✓

snoop • den laufenden Prozess im Halbschlaf befindlich erfassen können

trace • einen schläfrigen Prozess erfassen und ggf. als wach feststellen

doubt • den laufenden Prozess schlafen legen oder schläfrig fortsetzen

clean • den ggf. noch bestehenden Halbschlaf des Prozesses beenden

Beachte ↪ Wettlaufsituation

- laufende/bereitgestellte Prozesse sind nicht erneut bereitzustellen

Prozesszustandsüberwachung: Randbedingungen

Wiederbereitstellung eines Prozesses, der (a) noch läuft oder (b) als lauffähig bereitgestellt wurde, ist **betriebsartenabhängig** zu behandeln:

AMP (*asymmetric multiprocessing*) und PUP (*preemptive uniprocessing*)

- ein in P durch V überlappter Prozess steht auf der Bereitliste
  - $V_b \| P_b$ ⇒ nur **bedingte Bereitstellung** durch *ready*
  - $V_b \| P_{b_{\text{nach TAS}+}}$ ⇒ **Verdrängungssteuerung** durch *doubt*
- die Maßnahmen durch PSI und im Planer greifen lassen

SMP (*symmetric multiprocessing*)

- ein durch V bereitgestellter Prozess ist sofort abrufbar
- dieser darf nicht gleichzeitig auf einem anderen Kern laufen:
  1. schläfrigen Prozess in *snoop* an den eigenen Kern heften
  2. in *doubt* den Prozess dann wie im Leerlauffall behandeln
     ⇒ sich selbst von der Bereitliste nehmen

Beachte ↪ Pseudoparallele Ausführung ist kniffliger als echtparallele

- *snoop*, *doubt* und *trace* müssen einander helfen: **Zustandsmaschine**

Prozesszustandsüberwachung: Funktionale Eigenschaften

Vorbeugung der Wettlaufsituation bei der **Prozesswiederbereitstellung** durch einen verdrängbar arbeitenden Planer:

snoop
- weist den laufenden Prozess als „gefasst“ (*providing*) aus

doubt
- prüft, ob der laufende Prozess immer noch gefasst ist
  - ja ⇒ bis zum Blockieren **durchlaufen** (*run to completion*)
  - nein ⇒ *trace* ist erfolgt, nicht blockieren, **zurückkehren**
- das die Prüfung stattgefunden hat, wird *trace* kommuniziert
  - CAS ⇒ gefassten Prozess „blockierend“ (*blocking*) setzen

trace
- erfasst den nächsten Prozess auf der Warteliste, falls verfügbar
- dass der Prozess weiterlaufen kann, wird *doubt* kommuniziert
  - CAS ⇒ gefassten Prozess „aufgespürt“ (*traced*) setzen

clean
- räumt auf, der Prozess betritt „zufrieden“ (*pleased*) den KA

Beachte ↪ Prozessdurchlauf bis zum Blockieren
- eventuelle präemptive Umplanung des Prozessors ist aufzuschieben

Prozesszustandsüberwachung: Implementierungsvariante

```
void psi_snoop(line_t *line) {
    thread_t *self = sad_being();
    self->mood = ACT_PROVIDING;
    sad_stash(self, line);
}
```

```
void psi_doubt(line_t *line) {                                  psi_trace(line) !?
    thread_t *self = sad_being();                               ↙
    if (CAS(&self->mood, ACT_PROVIDING, ACT_BLOCKING)) sad_delay(self);
    else psi_snoop(line);
}
```

```
thread_t *psi_trace(line_t *line) {
    thread_t *next = sad_seize(line);
    return (next && !CAS(&next->mood, ACT_PROVIDING, ACT_TRACED)) ? next : 0;
}
```

```
void psi_clean(line_t *line) {
    thread_t *self = sad_being();
    if (self->mood == ACT_PROVIDING) sad_unban(self, line);
    self->mood = ACT_PLEASED;
}
```


Flattern (engl. *thrashing*) von Zwischenspeicherzeilen

CAS zerstört Lokalitäten im Zwischenspeicher, auch wenn die Operation nicht gelingt und eigentlich nicht hätte angewendet werden müssen
- hier: *doubt*/*trace* ⇔ *mood* ≠ *PROVIDING*
- sonst: z. B. *ready* ⇔ *mood* ≠ {*BLOCKING*, *BLOCKED*}
- Folge: Laufzeitbeinflussung anderer Prozesse, Leistungsschwächung

⇒ das CAS nur **bei voraussichtlichem Erfolg** zur Wirkung bringen

Bedingtes CAS: „*test and compare and swap*“

```
#define TACAS(r,e,v)  tacas((word_t *)r, (word_t)e, (word_t)v)
```

```
#include "luxe/inline.h"
#include "luxe/machine/inline/cas.c"

INLINE int tacas(word_t *ref, word_t exp, word_t val) {
    return (*ref == exp) && cas(ref, exp, val);
}
```

Beachte ↪ TAS wirkt ebenso ⇝ „*test and test and set*“: TATAS

⇒ auch das TAS nur bei voraussichtlichem Erfolg zur Wirkung bringen

Querschneidender Belang der Prozesseinplanung

Einplanung von Prozessen und **Koordinierung** gleichzeitiger Prozesse, die P und V nebenläufig passieren sollen, haben Einfluss aufeinander
- grundsätzlich gibt Einplanung die Prozesse (lokal/global) vor
- damit wird auch die Möglichkeit gleichzeitiger Prozesse geschaffen
- Koordinierung solcher Prozesse bedeutet aber **Reihenfolgenbildung**
- die mit dem jeweiligen Einplanungsverfahren verträglich sein sollte

Beachte ↪ Gerechtigkeit
- ein in P vor betreten seines Schlafzustands verdrängter Prozess, kann vorrangig in die semaphoreigene Warteschlange eingereiht werden
  - wenn der verdrängende Prozess P auf denselben Semaphor anwendet
- ein in V ausgewählter wartender Prozess kann nachrangig zu anderen aus seiner Warteschlange den KA betreten
  - wenn der verdrängende Prozess P und V vor dem verdrängten schafft

- mit RR [9] bezweckte Gerechtigkeit ist so nicht (mehr) durchsetzbar

Semaphor (binär) vs. Mutex

Dijkstra entwickelte den binären Semaphor (*mutex* [4, S. 345]), Scholten schlug die Verallgemeinerung vor [3, S. 28]

- ein binärer Semaphor kann als zählender Semaphor gesehen werden
  - mit Wertebereich $[N, 1]$, wobei $N \leq 0$ gleich Anzahl der „Schläfer“
  - zum Schutz kritischer Abschnitte wird er aber nie über 1 hinauswachsen
- für **kritische Abschnitte** sollte dieser nur bedingt zählend arbeiten:
  1. V muss **Wertebereichüberschreitung** verhindern: $load \leq 1$
  2. de- und inkrementieren ist **kostspieliger** als setzen und löschen
  3. die zählende Variante zeigt weitergehende **Überlappungsmuster**
  4. V darf nur der ausführen, der P ausführte: **Eigentümerschaft** prüfen **!!**
- beide Semaphore sind **Exemplare verschiedener Datentypen**

Heute ist zum Konzept „Semaphor“ irrtümlicherweise nur die allgemeine Bedeutung verbreitet, implementiert als zählende Variante

- hier muss V der Prozess tun können, der nicht das zugehörige P tat

Beachte ↪ Eigentümerschaft bzw. bedingtes Wirken von V

- Mutex ist ein binärer Semaphor, der V nur zugehörig zum P zulässt

P und V (binär) mit Überprüfung der Eigentümerschaft

mutex nach WSP (Abk. für Wolfgang Schröder-Preikschat)

```
#include "luxe/semaphore.h"

typedef struct mutex {
    semaphore_t sema;    /* binary semaphore used for mutual exclusion */
    thread_t *flux;      /* actual thread owning that semaphore */
} mutex_t;

extern void wsp_acquire(mutex_t *);  /* activate and own semaphore */
extern void wsp_release(mutex_t *);  /* conditionally deactivate semaphore */
```

```
void wsp_acquire(mutex_t *this) {
    ewd_prolaag(&this->sema);
    this->flux = sad_being();
}
```

being Zeiger auf den Kontrollblock des aktuellen Fadens

```
void wsp_release(mutex_t *this) {
    if (this->flux != sad_being())
        sad_panic(EPERM, this);
    else {
        this->flux = 0;
        ewd_verhoog(&this->sema);
    }
}
```

⇒ die Eigentümerschaft muss innerhalb des KA gesetzt/-prüft werden

P und V (binär) mit Überprüfung der Eigentümerschaft (Forts.)

Beachte ↪ Bedingter Prozessabbruch bei Freigabe des KA

- schwerwiegender Programmierfehler in der Benutzung von P/V !!!
  - V wird ohne das dazu korrespondierende, vorangehende P aufgerufen
  - heißt: ein Faden verlässt einen KA, den er vorher nicht betreten hatte
- einfaches ignorieren dieses Falls ist als **Systemfehler** einzustufen !!!

Panik schieben (engl. *panic*) ist die durchaus angemessene erste Reaktion, vor allem innerhalb eines Betriebssytemkerns

- das System selbst kann nur recht beschränkte Maßnahmen ergreifen:
  - Prozess abbrechen
  - Prozess einem Fehleraufspürer (engl. *debugger*) zustellen
  - Prozess eine Ausnahmesituation verursachen (engl. *raise*) lassen
- eine **synchrone Programmunterbrechung** ist mehr als zweckmäßig
  - ausgelöst durch den Prozessor der Ebene 3: das Betriebssystem [9]
  - behandelt im Programm der Ebene 2 oder Ebene 3, ggf. arbeitsteilig
    - d.h., im Betriebssystem oder/und im Maschinenprogramm

Gliederung

Abhängigkeit(en) zwischen Erzeuger und Verbraucher

Erzeuger
- produziert Daten in möglicherweise beliebiger Anzahl
- ist dabei **logisch unabhängig** vom Verbraucher
- braucht nicht auf den Verbraucher zu warten

⇒ V, zählend

Verbraucher
- konsumiert die vom Erzeuger produzierten Daten
- ist dabei **logisch abhängig** vom Erzeuger
- muss gegebenenfalls auf den Erzeuger warten

⇒ P, zählend

beide
- Datenhaltung braucht wiederverwendbare Betriebsmittel
  - Ursprungs- oder Zwischenspeicher, begrenzt
  - ⇒ P & V, zählend: **limitierender Faktor**
- impliziert Lese-/Schreibbefehle auf den Datenspeicher
  - ⇒ P & V, binär bzw. *Mutex* oder Schlossvariable
  - ⇒ aber ebenso: **nichtblockierende Synchronisation**

Erzeuger-Verbraucher-Problem

Datenpuffer mit Pufferbegrenzung (engl. *bounded buffer*, vgl. auch [9])

```
#define NDATA ...

typedef char data_t;

typedef struct {
    data_t data[NDATA];
    unsigned nput;
    unsigned nget;
    semaphore_t free;
    semaphore_t full;
} buffer_t;
```

```
void bb_raw(buffer_t *bb) {
    bb->nput = 0;
    bb->nget = 0;
    bb->free.load = NDATA;
    bb->full.load = 0;
}
```

```
void bb_put(buffer_t *bb, data_t item) {
    P(&bb->free);
    bb->data[FAI(&bb->nput)] = item;
    V(&bb->full);
}
```

```
data_t bb_get(buffer_t *bb) {
    data_t item;

    P(&bb->full);
    item = bb->data[FAI(&bb->nget)];
    V(&bb->free);

    return item;
}
```

```
#define FAI(ref)    FAA(ref, 1) % NDATA
```

FAA vgl. Kap. VI-2, S. 21

P und V (zählend) als nichtsequentielles Programm: Die 1.

```
void css_prolaag(semaphore_t *this) {
    psi_snoop(&this->line);
    while (FAA(&this->load, -1) <= 0)
        psi_doubt(&this->line);
    psi_clean(&this->line);
}
```

```
void css_verhoog(semaphore_t *this) {
    thread_t *next;

    if (FAA(&this->load, +1) < 0) {
        next = psi_trace(&this->line);
        if (next)
            sad_ready(next);
    }
}
```

- „zu schön, um wahr zu sein“

Beachte ↪ Unterschied zum binären Semaphor P_b/V_b

Seien P_c und V_c die Elementaroperationen eines zählenden Semaphors:
- vor *clean*: der P_c verlassende Prozess ist in V_c noch aufspürbar (*seize*)
  - statt eines in P_c blockierten Prozesses, wird er (ggf. erneut) erfasst
    - ⇒ ein durch V_c angezeigtes Zeitsignal könnte verloren gehen !!!
- falls $load > 0$: überlappte Ausführung von *clean* in P_c ist möglich
  - in P_b wird *clean* (logisch) im Kontext des geschützten KA ausgeführt
    - ⇒ sicherstellen, dass *clean* frei von Wettlaufsituationen ist

P und V (zählend) als nichtsequentielles Programm: Die 2.

Sicherstellung, dass ein Zeitsignal nicht verloren geht, wenn P_c in der Schlussphase von V_c überlappt wird — **Lösungsidee**:

snoop
- wie gehabt

doubt
- wie gehabt

trace
- markiert den erfassten Prozess als „erwischt“ (*caught*)
- die Marke muss Bestand haben bis zum Verlassen von P_c

clean
- prüft, ob der P_c verlassende Prozess erwischt wurde: CAS
  - ja ⇒ den nächsten wartenden Prozess aufspüren: *trace*
  - nein ⇒ keine weitere Maßnahme
- ein Aufruf von *trace* erfasst den zuvor verpassten Prozess
- der anschließend (bedingt) bereitzustellen ist ⇝ *ready*

Übung
- die Implementierung wird als „Trockenübung“ empfohlen...

Gliederung

Disputation: Binärer Semaphor $\overset{?}{\Longleftrightarrow}$ Mutex

Konzepte wie Schlossvariable, binärer Semaphor und Mutex dienen alle dem Zweck, **gleichzeitige Prozesse wechselseitig auszuschließen**

funktional folgen ihre Operationen demselben Verwendungsmuster
- sie „klammern“ kritische Abschnitte: *enter* KA *leave*
- wobei: $\xi_{leave_{t+1}} = \xi_{enter_t}$, für Prozess ξ zur Zeit t

nicht-funktional wirken sie jedoch verschieden
- vor allem: aktives oder passives Warten in *enter*
- **keine erzwingt** $\xi_{leave_{t+1}} = \xi_{enter_t}$ *per definitionem*

Beachte ↪ Fachliteratur und Lehrbücher
- uneinheitliche/unpräzise Definition der Konzepte, wenn überhaupt
- allgemeine „Folklore“:

  Die **Semantik** *eines Mutex ist, dass leave nur derjenige Prozess macht, der zuvor das enter gemacht hat.*
- dass diese Semantik durchgesetzt wird, steht auf einem anderen Blatt

Disputation: Binärer Semaphor $\overset{?}{\Longleftrightarrow}$ Mutex (Forts.)

Durchsetzung der zu *Mutex* überlieferten Semantik gibt Anlass über den Unterschied zum binären Semaphor nachzudenken:

vor Laufzeit
- Hansen [5, S. 93/94] erklärt, dass dies unmöglich ist

zur Laufzeit
- folgt direkt aus „Prozess = Programm in Ausführung“: *leave* prüft, welcher Prozess das *enter* machte (S. 26)

Beachte ↪ Theorie vs. Praxis
- Praxis zeigt, dass die *Mutex*-**Folklore** nicht zwingend umgesetzt ist
- Praxis zeigt, dass — für diese Fälle — kein Unterschied besteht
- Praxis zeigt, dass der *Mutex*-**Begriff** „des Kaisers neue Kleider“ sind

Was noch zu sagen wäre ↪ Dijkstra's *mutex* [4, S. 345], 1968

```
begin semaphore mutex; mutex := 1;   ← binärer Semaphor!!!
  parbegin
    begin L1: P(mutex); critial section 1; V(mutex); remainder of cycle 1; go to L1 end;
    begin L2: P(mutex); critial section 2; V(mutex); remainder of cycle 2; go to L2 end
  parend
end
```


Mutex — „*The Emperor's Old Clothes*“? (in Analogie zu [6])

Ein Symbol, dass (vor!) 1968 zur Bezeichnung einer Koordinierungsvariablen eingeführt wurde, hat sich in den zurückliegenden Jahren anscheinend als eigener Begriff verselbstständigt, ohne dass dieser jedoch klar zum Begriff des binären Semaphors abgegrenzt wurde, etwa durch eindeutige Formulierung der Gemeinsamkeiten und Unterschiede.

Resümee

Betriebsmittelart ⟷ Zugriffsart

- wiederverwendbar (begrenzt), konsumierbar (unbegrenzt)
- teilbar, unteilbar

Mehrseitige Synchronisation ⟷ binärer Semaphor

- Motivation: wechselseitiger Ausschluss (*mutual exclusion*)
- Semantik („P/V-Paarung“), *Mutex* als Spezialisierung

Einseitige Synchronisation ⟷ allgemeiner Semaphor

- Motivation: Erzeuger-Verbraucher-Problem (*bounded buffer*)
- Verallgemeinerung der Implementierung des binären Semaphors

Anhang:

Warteschlangen ⟷ Hierarchie

- Uni-/Multiprozessor, ein/mehr Bediener, {a,}symmetrisch
- Verteilung bei Freigabe/Ankunft, Arbeitsentzug

Literaturverzeichnis

[1] Blumofe, R. D. ; Leiserson, C. E.:
Scheduling Multithreaded Computations by Work Stealing.
In: *Journal of the ACM* 46 (1999), Nr. 5, S. 720–748

[2] Dijkstra, E. W.:
Over seinpalen / Technische Universiteit Eindhoven.
Eindhoven, The Netherlands, 1964 ca. (EWD-74). –
Manuskript. –
(dt.) Über Signalmasten

[3] Dijkstra, E. W.:
Cooperating Sequential Processes / Technische Universiteit Eindhoven.
Eindhoven, The Netherlands, 1965 (EWD-123). –
Forschungsbericht. –
(Reprinted in *Great Papers in Computer Science*, P. Laplante, ed., IEEE Press, New York, NY, 1996)

[4] Dijkstra, E. W.:
The Structure of the THE-Multiprogramming System.
In: *Communications of the ACM* 11 (1968), Mai, Nr. 5, S. 341–346

Literaturverzeichnis (Forts.)

[5] Hansen, P. B.:
Betriebssysteme.
Carl Hanser Verlag, 1977. –
ISBN 3–446–12105–6

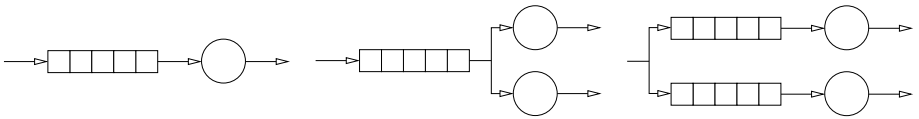
[6] Hoare, C. A. R.:
The Emperor's Old Clothes.
In: *Communications of the ACM* 24 (1981), Nr. 2, S. 75–83

[7] Lohmann, D. ; Kleinöder, J. :
Betriebssysteme.
http://www4.informatik.uni-erlangen.de/Lehre/WS07/V_BS, 2007

[8] Schröder-Preikschat, W. ; Kleinöder, J. :
Softwaresysteme 1.
http://www4.informatik.uni-erlangen.de/Lehre/SS04/V_SOS1, 2004–2007

[9] Schröder-Preikschat, W. ; Kleinöder, J. :
Systemprogrammierung.
http://www4.informatik.uni-erlangen.de/Lehre/WS08/V_SP, 2008 *ff.*

Warteschlangen laufbereiter/blockierter Prozesse



- 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

Arbeitsbezug durch Verteilung der CPU- oder E/A-Stöße

Bediener
- verarbeitende Komponente eines Wartesystems
  - Prozessor(kern), Gerät, Zusteller, . . . , Prozess
- Zuteilung eines Auftrags („Stoß“) 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 ⇒ **Hierarchie verwalteter Betriebsmittel**
- eine gemeinsame Warteliste erhöht Durchsatz und Wettstreitigkeit
- getrennte Wartelisten senkt Durchsatz und Wettstreitigkeit
- Kompromiss ist die **mehrstufige Organisation** des Wartesystems:
  - bedienerlokale Wartelisten zur Verringerung von Wettstreitigkeit
  - bedienerglobale Warteliste zur Erhöhung von Durchsatz

Arbeitsentzug (engl. *work stealing* [1])

Prozessoren versuchen dem **Leerlauf** (engl. *idle state*) vorzubeugen, indem sie Aufträge („CPU-Stöße“) von anderen Prozessoren stehlen
- keine Auftragsverteilung an prozessorlokale Warteschlangen
- stattdessen Auswahl einer gefüllten „fremden“ Warteschlange
  - typischwerweise nach einem **Zufallsauswahlverfahren**

Diebe (engl. *stealer*) entnehmen Aufträge vom Fuß (engl. *tail*) der jeweils ausgewählten Warteschlange eines anderen Prozessors
- aber nur, wenn die Warteschlange ihres eigenen Prozessors leer ist
- sie agieren „minimal invasiv“, um **Wettstreitrisiko** zu senken

Warteschlangen werden potentiell von gleichzeitigen Prozessen aktualisiert, die rollenspezifische **Zugriffsmuster** zeigen ⇝ **Koordinierungsaspekte**

Besitzer
- sind Leser *und* Schreiber ihrer eigenen Warteschlange

Diebe
- sind Leser fremder Warteschlangen und Besitzer sonst

Verwaltung blockierter Prozesse

```
INLINE void sad_stash(thread_t *this, line_t *line) {
#ifdef __fame_line_zilch
    this->wait = line;
#else
    nbs_aback(&line->wait, &this->wait);
#endif
}
```

```
INLINE void sad_unban(thread_t *this, line_t *line) {
#ifdef __fame_line_zilch
    this->wait = 0;
#else
    nbs_purge(&line->wait, &this->wait);
#endif
}
```

```
INLINE thread_t *sad_seize(line_t *line) {
#ifdef __fame_line_zilch
    thread_t *next;

    for (next = sad_table(); next < sad_table() + NTHREADS; next++)
        if (next->wait & TACAS(&next->wait, line, 0))
            return next;

    return 0;
#else
    return (thread_t *)nbs_fetch(&line->wait);
#endif
}
```

NBS — Abk. für (engl.) *nonblocking synchronization*

aback gegebenes Element „nach hinten“ auf die gegebene Warteschlange platzieren

fetch nächstes Element von vorne aus der gegebenen Warteschlange „rückholen“

purge gegebenes Element aus der gegebenen Warteschlange „löschen“

Verwaltung bereitgestellter Prozesse

```
void sad_ready(thread_t *this) {
    thread_t *self = sad_being();

    if (sad_merit(self, this) == self) {
        if (sad_valid(this))
            sad_queue(sad_stock(), this);
    } else {
        if (sad_valid(self)) {
            sad_queue(sad_stock(), self);
            sad_board(this);
        }
    }
}
```

```
void sad_delay(thread_t *this) {
    thread_t *next;

    this->mood = ACT_BLOCKED;
    while (!(next = sad_elect(sad_stock())))
        sad_coast(sad_stock());

    if (next != this)
        sad_board(next);
}
```

```
INLINE void sad_coast(stock_t *this) {
    cpu_coast(sad_token(this), 0, apm_cstate());
}
```

```
INLINE thread_t *sad_merit(thread_t *one, thread_t *two) {
#ifdef __fame_sad_preemption
    return (two->rank > one->rank) ? two : one;
else
    return one;
#endif
}
```

```
INLINE void *sad_token(stock_t *this) {
#ifdef __fame_sad_queue
    return &this->ready.head.link; /* queue */
else
    return &this->ready.last;      /* table */
#endif
}
```

```
INLINE int sad_valid(thread_t * this) {
    return TACAS(&this->mood, ACT_BLOCKING, ACT_READY) || TACAS(&this->mood, ACT_BLOCKED, ACT_READY);
}
```

Leerlaufsteuerung: x86

```
INLINE void cpu_coast(void *ref, int exp, int esm) {
    cpu_watch(ref);           /* let CPU monitor given memory address */
    if (*ref == exp)          /* does CPU idle condition still hold? */
        cpu_await(ref, esm);  /* yes, enter CPU idle mode with given C-state */
}
```

```
INLINE void cpu_watch(void *ref) {
    asm volatile (
        "xorl %%ecx, %%ecx\n\t"
        "xorl %%edx, %%edx\n\t"
        "monitor"
        :
        : "a" (ref)
        : "eax", "ecx", "edx");
}
```

```
INLINE void cpu_await(void *ref, int esm) {
    asm volatile (
        "mwait"
        :
        : "a" (ref), "c" (esm)
        : "eax", "ecx");
}
```

Beachte ↪ Analogie zur Implementierung von P/V

1. anzeigen, welche Adresse der Prozessor(kern) überwachen soll ∼ *snoop*
2. die mit der Adresse verknüpfte Wartebedingung auswerten ∼ TAS/CAS
3. den Prozessor(kern) bedingt schlafen legen ∼ *doubt*