

Systemprogrammierung

Grundlagen von Betriebssystemen

Teil C – X.3 Prozesssynchronisation:

Kausalitätsdilemma — Wer „benutzt“ wen,
der Monitor den Semaphore oder umgekehrt?

Wolfgang Schröder-Preikschat

— Selbststudium —



Agenda

Einführung

Semaphor als Monitor

- Sprachkonzept

- Programmierkonvention

- Laufzeitroutinen

- Intermezzo

Semaphor als Systemabstraktion

- Unteilbarer Datentyp

- Teilbarer Datentyp

Gemeinsame Basis

- Warteliste

- Wechselseitiger Ausschluss

Zusammenfassung



Gliederung

Einführung

Semaphor als Monitor

Sprachkonzept

Programmierkonvention

Laufzeitroutinen

Intermezzo

Semaphor als Systemabstraktion

Unteilbarer Datentyp

Teilbarer Datentyp

Gemeinsame Basis

Warteliste

Wechselseitiger Ausschluss

Zusammenfassung



Der Frage nachgehen, ob ein Semaphore am besten als Monitor oder ein Monitor mit Hilfe von Semaphore implementiert werden sollte.

- nach [2, S. 29] sind P und V **unteilbare Operationen** (vgl. [9])
 - die bei gleichzeitigen Prozessen kritische Abschnitte offenbaren:
„[...] critical in the sense that the processes have to be constructed in such a way, that at any moment at most one of [them] is engaged in its critical section.“ [2, S. 11]
 - dies legt eine Implementierung auf der Basis des Monitorkonzepts nahe
- Monitore führen mehr- und einseitige Synchronisation zusammen
 - nämlich **wechselseitiger Ausschluss** von Prozessen beim Aufruf der Monitorprozeduren und **Prozesskommunikation** innen drin [5, 6]
 - dies legt eine Implementierung auf der Basis von Semaphore nahe [7]
- eine Art Henne-Ei-Problem
 - das eine Form von „*bootstrap*-Technik“ [1, S. 1273] zur Lösung braucht



Gliederung

Einführung

Semaphor als Monitor

Sprachkonzept

Programmierkonvention

Laufzeitroutinen

Intermezzo

Semaphor als Systemabstraktion

Unteilbarer Datentyp

Teilbarer Datentyp

Gemeinsame Basis

Warteliste

Wechselseitiger Ausschluss

Zusammenfassung



- fiktive nichtsequentielle Programmiersprache (vgl. Anhang, S. 31)

```
1 monitor<Mesa> Semaphore {
2     int load;                /* semaphore counter */
3     condition zero;         /* monitor's condition variable */
4 protected:
5     void prolaag() {
6         when (load-- <= 0)
7             wait zero;
8     }
9
10    void verhoog() {
11        if (load++ < 0)
12            signal zero;
13    }
14 public:
15     inline void P() { prolaag(); }
16     inline void V() { verhoog(); }
17 };
```

Eine C++ Template-Klasse, deren Parameter die Art des Monitors (Hoare, Hansen, Mesa) und damit die Semantik der Signalisierung (vgl. [8]) definiert. Mit den Schlüsselwörtern `condition` zur Deklaration einer oder mehrerer Bedingungsvariablen, `when` als Fallunterscheidung (`if`) oder Kopfschleife (`while`), je nach Art des Monitors, `wait` zur Synchronisation auf ein mit der Bedingungsvariablen verknüpftes Ereignis und `signal` zur Anzeige eines solchen Ereignisses.

- ein Übersetzer generiert daraus das nichtsequentielle Programm
 - à la cfront [13] möglicherweise auch ein C-Programm
 - vielleicht sogar einen Code sehr ähnlich zu den Darstellungen auf S. 7/8



- ein Zähler ergänzt um eine Sperre¹ und eine Ereignisliste

```
1  #include "event.h" und "turnstile.h"
2
3  typedef struct semaphore {
4      int load;          /* counter */
5      turnstile_t lock;  /* something for mutual exclusion */
6      event_t zero;      /* event list for condition variable */
7  } semaphore_t;
```

- zum Datentyp zugehörige „Monitorprozeduren“

```
8  extern void prolaag(semaphore_t *);
9  extern void verhoog(semaphore_t *);
```

- gebräuchliche Akronyme samt Ausformulierungen

```
10 inline void P(semaphore_t *this) { prolaag(this); }
11 inline void V(semaphore_t *this) { verhoog(this); }
```

¹(en.) *turnstile* für (dt.) Drehkreuz im Sinne von Vereinzelungsanlage.



```
1  #include "semaphore.h"
```

- Semaphorzähler um 1 verringern (nl.) *verlaag* ^{Dijkstra} $\rightsquigarrow$ *prolaag*²

```
2  void prolaag(semaphore_t *this) {
3      enter(&this->lock);          /* mutual exclusion begin */
4      if (this->load-- <= 0)        /* counter value shows "stop" */
5          await(&this->zero, &this->lock);    /* wait outside */
6      leave(&this->lock);          /* mutual exclusion end */
7  }
```

- Zunahme (nl.) *verhoog* des Semaphorzählers um 1

```
8  void verhoog(semaphore_t *this) {
9      enter(&this->lock);          /* mutual exclusion begin */
10     if (this->load++ < 0)         /* at least one process waiting */
11         cause(&this->zero);      /* deblock exactly one process */
12     leave(&this->lock);          /* mutual exclusion end */
13 }
```

²Kunstwort, nach Dijkstra aus „**probeer**“ (nl.) für „versuchen“ und *verlaag*.



- ein Zähler ergänzt um eine Warteliste

```
1  #include "waitlist.h" und "turnstile.h"
2
3  typedef struct {
4      int load;          /* number of waiting processes */
5      waitlist_t wait;   /* list of waiting processes */
6  } event_t;
```

- zum Datentyp zugehörige Zugriffsoperationen

```
7  extern void await(event_t *, turnstile_t *);
8  extern void cause(event_t *);
```

- await**
- lässt den Prozess auf ein Ereignis außerhalb des Monitors warten
 - gibt das „Drehkreuz“ frei und stellt sich dort später wieder an
 - erbittet beim Schlafengehen um „erhöhte Aufweckpriorität“
- cause**
- verursacht ein Ereignis und weckt einen schlafenden Prozess auf
 - der aufweckende Prozess behält den Monitor
 - wann der aufwachende Prozess fortsetzt, ist ungewiss



```
1  #include "urges.h", "event.h" und "process.h"
```

- der Prozess blockiert mit hoher Dringlichkeit zur Wiederaufnahme³

```
2  void await(event_t *this, turnstile_t *lock) {
3      allot(&this->wait); /* mark as soon waiting ("sleepy") */
4      this->load += 1;      /* one soon waiting process more */
5      leave(lock);         /* exit surrounding monitor */
6
7      block(UHIGH);        /* potential processor release */
8
9      enter(lock);         /* re-enter surrounding monitor */
10     this->load -= 1;      /* one waiting process less */
11 }
```

- ein verursachtes Ereignis wird nicht vermerkt

```
12 void cause(event_t *this) {
13     if (this->load > 0)    /* at least one process waiting */
14         waken(&this->wait); /* deblock exactly one process */
15 }
```

³allot beugt mögliches „lost wake-up“ zwischen leave und block vor.



- im Grunde ist ein Semaphor eine **gemeinsame Verbundvariable**, auf die gleichzeitige Prozesse zugreifen
 - der gemeinsame, gleichzeitige Zugriff bewirkt **gekoppelte Prozesse**
 - die am Semaphor konkurrierenden Prozesse, sind explizit zu koordinieren:
To do this [you] must be able to associate an arbitrary number of event queues with a shared variable and control the transfers of processes to and from them. [5, S. 577]
 - die Prozesskommunikation im Monitor geschieht daher ereignisorientiert
 - dabei konsumiert P ein Ereignis (await), das V produziert (cause)
- **wechselseitiger Ausschluss** durch P/V -Klammerung ist ein Problem
 - der Semaphormonitor müsste sich selbst „benutzen“ [12] $\rightsquigarrow$ **Stillstand** [10]
 - es sei denn, P/V sind nicht von allgemeiner, zählender Art wie hier
- unproblematisch ist somit der **binäre Semaphor** mit enter/leave als P/V -Äquivalent — der dann aber kein Monitor sein kann
 - zur Implementierung eines Monitors werden Systemabstraktionen benötigt
 - wobei diese Abstraktionen Semaphore sind oder ihnen sehr stark ähneln



Gliederung

Einführung

Semaphor als Monitor

Sprachkonzept

Programmierkonvention

Laufzeitroutinen

Intermezzo

Semaphor als Systemabstraktion

Unteilbarer Datentyp

Teilbarer Datentyp

Gemeinsame Basis

Warteliste

Wechselseitiger Ausschluss

Zusammenfassung



- ein Zähler ergänzt um eine Sperre und eine Warteliste

```
1  #include "waitlist.h" und "turnstile.h"
2
3  typedef struct semaphore {
4      int load;           /* counter */
5      turnstile_t lock;   /* something for mutual exclusion */
6      waitlist_t wait;    /* list of waiting processes */
7  } semaphore_t;
```

- zum Datentyp zugehörige Semaphoroperationen

```
8  extern void prolaag(semaphore_t *);
9  extern void verhoog(semaphore_t *);
```

- gebräuchliche Akronyme samt Ausformulierungen

```
10 inline void P(semaphore_t *this) { prolaag(this); }
11 inline void V(semaphore_t *this) { verhoog(this); }
```



```
1  #include <assert.h>
2  #include "urge.h", "process.h" und "semaphore.h"
```

■ klassische Auslegung als kritischer Abschnitt (CS)⁴

```
3  void prolaag(semaphore_t *this) {
4      enter(&this->lock);           /* critical section begin */
5      if (this->load-- <= 0) {        /* counter value shows "stop" */
6          allot(&this->wait);        /* mark as waiting */
7          leave(&this->lock);        /* wait outside CS */
8          block(UHIGH);             /* potential CPU release */
9          assert(this->lock.flag);   /* CS must be locked! */
10     } /* deblocked process could not have been overtaken! */
11     leave(&this->lock);            /* critical section end */
12 }
```

■ der aufgewachte Prozess wird im kritischen Abschnitt fortgesetzt

- siehe V (S. 15), das den kritischen Abschnitt bedingt gesperrt lässt
- hohe Dringlichkeit zur Wiederaufnahme vermeidet lange Sperrzeiten
- Hinweis an den Scheduler, deblockierte Prozesse kurzzeitig anzutreiben

■ ein faires Verfahren bei wettstreitigen Prozessen am selben Semaphor

⁴allot beugt mögliches „lost wake-up“ zwischen leave und block vor.



- klassische Auslegung als kritischer Abschnitt (CS)

```
13 void verhoog(semaphore_t *this) {
14     enter(&this->lock);           /* critical section begin */
15     if (this->load++ < 0)          /* waiting process(es) */
16         waken(&this->wait);       /* deblock exactly one */
17     else                          /* no one waits, release CS! */
18         leave(&this->lock);       /* critical section end */
19 }
```

- der kritische Abschnitt bleibt gesperrt, sollte ein Prozess warten

Vorteil ■ der aufwachende Prozess kann nicht überholt werden

■ vergleichsweise einfaches Protokoll zwischen P und V

Nachteil ■ der kritische Abschnitt bleibt die ganze Zeit über gesperrt

– also auch für Zeiten, während unabhängige Prozesse stattfinden

– Prozesse, die nicht am CS gekoppelt sind, dehnen die Sperrzeit

■ wann der aufwachende Prozess fortsetzt, ist ungewiss

- kann leistungsmindernd sein, ist kritisch für zeitabhängige Prozesse



- ein atomarer Zähler ergänzt um eine Warteliste

```
1  #include "event.h" und "waitlist.h"
2
3  typedef struct semaphore {
4      int load;          /* counter */
5      waitlist_t wait;   /* list of waiting processes */
6  } semaphore_t;
```

- zum Datentyp zugehörige Semaphoroperationen

```
7  extern void prolaag(semaphore_t *);
8  extern void verhoog(semaphore_t *);
```

- gebräuchliche Akronyme samt Ausformulierungen

```
9  inline void P(semaphore_t *this) { prolaag(this); }
10 inline void V(semaphore_t *this) { verhoog(this); }
```



```
1  #include "elop.h", "urge.h", "process.h" und "semaphore.h"
```

- unkonventionelle Auslegung als parallel ausführbarer Abschnitt⁵

```
2  void prolaag(semaphore_t *this) {  
3      allot(&this->wait);          /* mark process as "sleepy" */  
4      if (FAA(&this->load, -1) > 0) /* resource available */  
5          purge(&this->wait);      /* awake, keep going */  
6      else                          /* counter showed "stop" */  
7          block(UHIGH);           /* potential bedtime */  
8  }
```

- FAA serialisiert die entscheidende Aktion im P zur Prozesssteuerung
 - nämlich die Ressourcenanforderung zu verbuchen (Z. 4) und
 - den Prozess entweder voranschreiten oder blockieren zu lassen
- scheitert die Ressourcenanforderung, wartet der Prozess auf ein V
 - er blockiert mit hoher Dringlichkeit zur Wiederaufnahme (Z 7)
 - ein Hinweis an den Planer, deblockierte Prozesse kurzzeitig anzutreiben

⁵allot beugt mögliches „lost wake-up“ zwischen FAA (Z. 4) und block vor.



- unkonventionelle Auslegung als parallel ausführbarer Abschnitt

```
9 void verhoog(semaphore_t *this) {  
10     if (FAA(&this->load, +1) < 0)    /* waiting process(es) */  
11         waken(&this->wait);          /* deblock exactly one */  
12 }
```

- FAA serialisiert die entscheidende Aktion im V zur Prozesssteuerung
 - nämlich die Ressourcenbereitstellung zu verbuchen (Z. 10) und
 - gegebenenfalls einen darauf wartenden Prozess zu deblockieren (Z. 11)
- das P/V -Protokoll ist fair, bei geringer Interferenz mit dem Planer
 - ein durch V deblockierter Prozess kann in P nicht verhungern
 - gekoppelte Prozesse werden gemäß Planerstrategie abgearbeitet
 - dies gilt für die deblockierten (V) und die neu eintreffenden (P) Prozesse
 - wartende Prozesse (P) werden nach Ankunftszeit abgearbeitet
- Signale gehen nicht verloren, da V mit FAA immer und korrekt zählt

P und V müssen keine unteilbaren Operationen sein, um bei gleichzeitigen, am selben Semaphor gekoppelten Prozesse noch korrekt zu funktionieren!



Gliederung

Einführung

Semaphor als Monitor

Sprachkonzept

Programmierkonvention

Laufzeitroutinen

Intermezzo

Semaphor als Systemabstraktion

Unteilbarer Datentyp

Teilbarer Datentyp

Gemeinsame Basis

Warteliste

Wechselseitiger Ausschluss

Zusammenfassung



- eine bedingte Warteschlange⁶

```
1  #include "urges.h", "process.h" und bedingt "queue.h"
2
3  typedef struct waitlist {
4      conditional queue_t list;  /* waiting processes */
5  } waitlist_t;
```

- zum Datentyp zugehörige Zugriffsoperationen

```
6  extern void  allot(waitlist_t *);
7  extern void  purge(waitlist_t *);
8  extern void  *waken(waitlist_t *);
```

allot ■ sich selbst auf die Warteliste setzen, Prozess disponieren

purge ■ die Warteliste von einem selbst (laufender Prozess) bereinigen

waken ■ einen auf der Warteliste stehenden Prozess aufwecken

- gebräuchliche Funktionssymbolik samt Ausformulierungen

```
9  inline void sleep(waitlist_t *wait) { allot(wait); block(UN0); }
10 inline void wakeup(waitlist_t *wait) { (void)waken(wait); }
```

⁶Inkludiert durch bedingte Kompilation.



```
1  #include "being.h"
```

- den noch laufenden Prozess als Aufweckkandidaten disponieren

```
2  void allot(waitlist_t *this) {
3      process_t *self = being(ONESELF);
4      label(self, this);          /* define waiting token */
5      conditional enqueue(&this->list, &self->line);
6  }
```

- dem laufenden Prozess seinen Aufweckkandidatenstatus nehmen⁷

```
7  void purge(waitlist_t *this) {
8      process_t *self = being(ONESELF);
9      label(self, 0);             /* clear waiting token */
10     conditional discard(&this->list, &self->line);
11 }
```

- den nächsten Prozess von der Warteliste nehmen, ihn aufwecken

```
12 void *waken(waitlist_t *this) {
13     process_t *next = check(this);          /* remove list entry */
14     if (next) ready(next);                  /* put process on ready list */
15     return next;                            /* may be that a process was readied */
16 }
```

⁷Die Bereinigung der bedingt vorhandenen Queue kann entfallen, wenn beim Aufwecken als gelöscht markierte Einträge übersprungen werden (check, S. 22).



```
17 conditional process_t *check(waitlist_t *this) {
18     process_t *next;
19
20     variant "queue entries sorted by arrival time (FCFS)"
21     do {
22         chain_t *item = dequeue(&this->list);
23         if (!item)
24             return 0;
25         next = forge(item);
26     } while (!match(next, this));
27     return next;
28
29     variant "queue entries sorted in ascending order by number"
30     next = chief();
31     for (int tops = stock(); next && (tops > 0); tops--) {
32         if (match(next, this))
33             return next;
34         next = other(next);
35     }
36     return 0;
37
38 }
```

`match(p, w) ■ (p->wait == w) && CAS(&p->wait, w, 0) ? p : 0`



- ein „Drehkreuz“ als Merker (*flag*) ergänzt um eine Warteliste

```
1 #include <stdbool.h> und "waitlist.h"
2
3 typedef struct turnstile {
4     bool flag;           /* lock state: locked when true */
5     waitlist_t wait;     /* list of waiting processes */
6 } turnstile_t;
```

- zum Datentyp zugehörige Zugriffsoperationen

```
7 extern void enter(turnstile_t *);
8 extern void leave(turnstile_t *);
```

- enter** ■ ein Prozess betritt das „Drehkreuz“
- gleichzeitig eintreffende Prozesse passieren nur nacheinander⁸
- leave** ■ der Prozess betritt das „Drehkreuz“
- der nächste, gegebenenfalls wartende, Prozess kann passieren

⁸Ganz im Sinne einer Vereinzelungsanlage.



```
1  #include "elop.h", "urge.h", "process.h" und "turnstile.h"
```

- an dem „Drehkreuz“ dürfen sich gleichzeitige Prozesse vordrängen⁹

```
2  void enter(turnstile_t *this) {
3      do {
4          allot(&this->wait);          /* mark process as "sleepy" */
5          if (!TAS(&this->flag)) {      /* lock acquired */
6              purge(&this->wait);      /* awake, keep going */
7              return;                  /* enter critical section */
8          }
9          block(UHIGH);                 /* potential bedtime */
10         } while (TAS(&this->flag)); /* has been overtaken, retry */
11 }
```

- unbedingte Freigabe riskiert Überholung, aber kein „lost update“

```
12 void leave(turnstile_t *this) {
13     this->flag = false;                /* release critical section */
14     waken(&this->wait);                 /* rouse one sleeper, if any */
15 }
```

⁹allot beugt mögliches „lost wake-up“ zwischen TAS und block vor, while toleriert mögliche Überholungen seit Freigabe des kritischen Abschnitts (Z. 13).



- bedingte Freigabe vermeidet Überholung des aufwachenden Prozesses

```
1 void leave(turnstile_t *this) {  
2     if (!waken(&this->wait))  
3         this->flag = false;  
4 }
```

Was aber, wenn genau zwischen Z. 2 und Z. 3 ein weiterer Prozess ein P auf demselben Semaphor ($this$) versucht?

- ist die Sperrflagge gesetzt ($TAS = true$), muss ein Prozess blockieren

```
5 void enter(turnstile_t *this) {  
6     allot(&this->wait);  
7     if (TAS(&this->flag))  
8         block(UHIGH);  
9     else  
10         purge(&this->wait);  
11 }
```

Ein P ausführender Prozess wird durch V frühestens nach Z. 6 als „schläfrig“ wahrgenommen. Was aber, wenn dieser Prozess Z. 6 erst erreicht, nachdem der V ausführende Prozess zwar Z. 2, aber noch nicht Z. 3 passiert hat?

- ein Prozess in V kann die Aktualisierung der Warteliste versäumen
 - bei unbedingter Freigabe in V , blockiert der nächste Prozesse in P nicht
 - das aber muss der gegebenenfalls in V aufgeweckte Prozess erkennen
 - er darf P nach Rückkehr aus `block` (Z. 8) nicht unbedingt verlassen
 - die Lösung auf S. 24 trägt dem Rechnung — lässt dann aber einen soeben aufwachenden Prozess eventuell immer wieder blockieren $\rightsquigarrow$ „starvation“



Gliederung

Einführung

Semaphor als Monitor

Sprachkonzept

Programmierkonvention

Laufzeitroutinen

Intermezzo

Semaphor als Systemabstraktion

Unteilbarer Datentyp

Teilbarer Datentyp

Gemeinsame Basis

Warteliste

Wechselseitiger Ausschluss

Zusammenfassung



Resümee

- einen Semaphor als Monitor zu implementieren, ist bedingt adäquat
 - nämlich nur falls **programmiersprachliche Unterstützung** besteht oder
 - wenn der Mehraufwand der **Programmierkonvention** tolerierbar ist
- ein Monitor-**Laufzeitsystem** auf Semaphore zu basieren, ist adäquat
 - wechselseitiger Ausschluss ermöglicht der **binäre Semaphor** und
 - Prozesskommunikation leistet ein **allgemeiner Semaphor**
 - wobei jedoch die **Ereignissignale** nicht gespeichert werden dürfen!
- den Monitor selbst auf Semaphore zu basieren, ist missverständlich
 - da zu „Monitor“ eben Programmiersprache und Kompilierer gehören
 - auch wenn [7] explizit Semaphore nennt, so wird damit auf Eigenschaften das Laufzeitsystems eines Hoare'schen Monitors Bezug genommen
 - Monitore nach Semaphore zu übersetzen macht Sinn, umgekehrt nicht!

Zu guter Letzt...

Wie ein Monitor-Laufzeitsystem ab block und ready „nach unten“ zur Befehlssatzebene weiter gestaltet sein kann, zeigt [3, Anhang].^a

^ahttps://www4.cs.fau.de/Publications/2015/drescher_15_cstr.pdf



Literaturverzeichnis I

- [1] BUCHHOLZ, W. :
The System Design of the IBM Type 701 Computer.
In: *Proceedings of the IRE* 41 (1953), Nr. 10, S. 1262–1275.
<http://dx.doi.org/10.1109/JRPROC.1953.274300>. –
DOI 10.1109/JRPROC.1953.274300

- [2] 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)

- [3] DRESCHER, G. ; SCHRÖDER-PREIKSCHAT, W. :
An Experiment in Wait-Free Synchronisation of Priority-Controlled Simultaneous Processes: Guarded Sections / Friedrich-Alexander-Universität Erlangen-Nürnberg, Department of Computer Science.
Erlangen, Germany, Jan. 2015 (CS-2015-01). –
Technical Reports

- [4] GEHANI, N. H. ; ROOME, W. D.:
Concurrent C++: Concurrent Programming with Class(es).
In: *Softw. Pract. Exper.* 18 (1988), dec, Nr. 12, S. 1157–1177



Literaturverzeichnis II

- [5] HANSEN, P. B.:
Structured Multiprogramming.
In: *Communications of the ACM* 15 (1972), Jul., Nr. 7, S. 574–578
- [6] HANSEN, P. B.:
The Programming Language Concurrent Pascal.
In: *IEEE Transactions on Software Engineering* SE-1 (1975), Jun., Nr. 2, S. 199–207
- [7] HOARE, C. A. R.:
Monitors: An Operating System Structuring Concept.
In: *Communications of the ACM* 17 (1974), Okt., Nr. 10, S. 549–557
- [8] KLEINÖDER, J. ; SCHRÖDER-PREIKSCHAT, W. :
Monitor.
In: [11], Kapitel 10.2
- [9] KLEINÖDER, J. ; SCHRÖDER-PREIKSCHAT, W. :
Semaphore und Sperren.
In: [11], Kapitel 10.3
- [10] KLEINÖDER, J. ; SCHRÖDER-PREIKSCHAT, W. :
Stillstand.
In: [11], Kapitel 11



Literaturverzeichnis III

- [11] KLEINÖDER, J. ; SCHRÖDER-PREIKSCHAT, W. ; LEHRSTUHL INFORMATIK 4 (Hrsg.):
Systemprogrammierung.
FAU Erlangen-Nürnberg, 2015 (Vorlesungsfolien)

- [12] PARNAS, D. L.:
Some Hypothesis About the “Uses” Hierarchy for Operating Systems / TH
Darmstadt, Fachbereich Informatik.
1976 (BSI 76/1). –
Forschungsbericht

- [13] STROUSTRUP, B. :
Evolving a Language in and for the Real World: C++ 1991-2006.
*In: Proceedings of the Third ACM SIGPLAN Conference on History of Programming
Languages.*
Association for Computing Machinery (HOPL III). –
ISBN 9781595937667, 4:1–4:59



```
1  type
2      semaphore = monitor(seed: integer);
3      var
4          load: integer;
5          zero: resource(NPROC);
6
7      procedure entry prolaag;
8      begin
9          load := load - 1;
10         if load < 0 then zero.acquire;
11     end;
12
13     procedure entry verhoog;
14     begin
15         load := load + 1;
16         if load <= 0 then zero.release;
17     end;
18
19 begin
20     load := seed;
21 end
```

Schlüsselwort **entry** deklariert eine **Monitorprozedur**, deren Ausführung dem wechselseitigen Ausschluss unterliegt. Für die **Prozesskommunikation** ist die resource-Abstraktion (S. 32) zuständig, die eine Warteliste von Prozessen implementiert.



- die Prozesswarteschlange (queue) enthält maximal einen Eintrag
 - „gemeine Warteschlangen“ sind Felder von Einzelprozesswarteschlangen

```
1  type
2      resource = monitor(size: integer);
3      var
4          list: array [0..size-1] of queue;
5          head, tail, wait: integer;
6          free: boolean;
7      .
8      .
29  begin
30      head := 0; tail := 0; wait := 0;
31      free := true;
32  end
```

- darauf basierend eine ankunftszeitbasierte Prozesssteuerung (FCFS)

<pre>7 procedure entry acquire; 8 var self: integer; 9 begin 10 if free then free := false else 11 begin 12 self = tail; 13 tail := (tail + 1) mod size; 14 wait := wait + 1; 15 delay(list[self]); 16 end; 17 end;</pre>	<pre>18 procedure entry release; 19 var next: integer; 20 begin 21 if wait = 0 then free := true else 22 begin 23 next = head; 24 head := (head + 1) mod size; 25 wait := wait - 1; 26 continue(list[next]); 27 end; 28 end;</pre>
--	---

- delay blockiert den aktiven Prozess, continue setzt einen anderen fort

