— BS //—

Semaphor

Überblick

Semaphor — „Zeichenträger“, Signalmast

sem|a·phore **1** any apparatus for signaling, as by an arrangement of lights, flags, and mechanical arms on railroads **2** a system for signaling by the use of two flags, one held in each hand: the letters of the alphabet are represented by the various positions of the arms **3** any system of signaling by semaphore [8]

Betriebssystemunterstützung zur Prozesssynchronisation

- wenn Wartesituationen durch **Prozessblockaden** realisiert werden sollen, [5]:

  Semaphore Eine *gemeinsame Datenstruktur* zum Austausch von Zeitsignalen zwischen *gleichzeitigen Prozessen*.

  gleichzeitige Prozesse *Prozesse*, deren Ausführung sich zeitlich überschneidet. Gleichzeitige Prozesse heißen unabhängig, wenn sie nur zu ihren eigenen Daten zugreifen; sie sind gekoppelt, wenn sie gemeinsame Daten verwenden.

- ein(e) Semaphor(e) stellt eine prozessglobale **Koordinationsvariable** dar

Betriebsmittelverwaltung

- Prozessblockaden sind bedingt durch die Konkurrenz um Betriebsmittel
  - Prozesse { benötigen Betriebsmittel, um voranschreiten zu können / müssen ggf. warten, bis Betriebsmittel verfügbar sind
- ein Semaphor ist ein grundlegendes Instrument zur Betriebsmittelvergabe:

  binärer Semaphor verwaltet zu einem Zeitpunkt nur genau ein Betriebsmittel
  – z.B. die CPU im Falle eines atomar auszuführenden Programmabschnitts

  zählender Semaphor verwaltet zu einem Zeitpunkt mehrere Betriebsmittel
  – z.B. die in einem (begrenzten) Puffer jeweils noch verfügbaren Bytes

- Ko{operation,mmunikation} bedeutet Betriebsmittelaustausch/-weitergabe

Betriebsmittelarten

wiederverwendbare Betriebsmittel werden angefordert und freigegeben

- ihre Anzahl ist begrenzt: Prozessoren, Geräte, Speicher (z.B. Puffer)

$$\left.\begin{array}{r}\textit{teilbar}\\ \textit{unteilbar}\end{array}\right\}\ \text{wenn zu einem Zeitpunkt}\ \left\{\begin{array}{c}\text{von mehreren Prozessen}\\ \text{nur von einem Prozess}\end{array}\right\}\ \text{belegbar}$$

konsumierbare Betriebsmittel werden erzeugt und zerstört

- ihre Anzahl ist (logisch) unbegrenzt: Signale, Nachrichten, Interrupts
- *Produzenten*-Prozesse können beliebig viele davon erzeugen
- Zerstörung erfolgt bei Inanspruchnahme durch *Konsumenten*-Prozesse
- letztere müssen ggf. auf die Verfügbarkeit der Betriebsmittel warten

Dijkstra's Semaphor{e,s}

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 the merit of the Dutch physicist and computer designer Drs. C. S. Scholten to have demonstrated a considerable field of applicability for semaphores that can also take on larger values. When there is a need for distinction, we shall talk about "*binary semaphores*" and "*general semaphores*" respectively. [3]

P Definition V

The P-operation is an operation with one argument, which must be the identification of a semaphore. [. . .] Its function is to *decrease* the value of its argument semaphore *by 1* as soon as the resulting value would be non-negative. The completion of the P-operation—i.e. the decision that this is the appropriate moment to effectuate the decrease and the subsequent decrease itself—is to be regarded as an *indivisible operation*.

The V-operation is an operation with one argument, which must be the identification of a semaphore. [. . .] Its function is to *increase* the value of its argument semaphore *by 1*; this increase is to be regarded as an *indivisible operation*. [3]

P Protokoll V

It is the P-operation, which represents the *potential delay*, viz. when a process initiates a P-operation on a semaphore, that at that moment is $= 0$, in that case this P-operation cannot be completed until another process has performed a V-operation on the same semaphore and has given it the value "1". [3]

Datentyp Semaphor

```
class Semaphore {
    unsigned int s;
public:
    Semaphore (unsigned int = 0);

    void prolaag ();
    void verhoog ();

    void P ()       { prolaag(); }
    void V ()       { verhoog(); }

    void wait   () { P(); }
    void signal () { V(); }
};
```

} Namensgebung durch Dijkstra [2]

} Begriff seit Dijkstra [3]

} Synonym von Hansen [5]

P Implementierung (1) V

„erniedrige“[1], *down*, *wait*

```
void Semaphore::prolaag () {
    lock.enter();
    if (s == 0)
        soul->sleep(this);
    s -= 1;
    lock.leave();
}
```

erhöhe, *up*, *signal*

```
void Semaphore::verhoog () {
    Thread* gain;
    lock.enter();
    s += 1;
    if ((s == 1)
    &&  (gain = soul->awake(this)))
        gain->ready();
    lock.leave();
}
```

† Dabei sei `soul` der Zeiger auf den (die Klasse `tocObject` spezialisierenden) `Thread`-Kontrollblock des aktuell laufenden Prozesses (siehe auch [9], „Prozesse“, S. 17).

P Funktionen des Schedulers (1) V

`sleep(void*)` legt den applizierten/laufenden Prozess schlafen

- der Prozess erwartet das Auftreten eines Ereignisses[1]
- ein anderer Prozess wird ggf. zur Ausführung gebracht

`awake(void*)` weckt einen Prozess auf

- einer der das Ereignis erwartenden Prozesse wird ggf. selektiert
- die Ereignisbedingung des ggf. selektierten Prozesses wird invalidiert

`ready()` setzt den applizierten Prozess laufbereit

- der Prozess wird der Strategie des *short-term scheduling* unterzogen

[1] Das Ereignis wird repräsentiert durch die Adresse *dieser* (`this`) Semaphorinstanz.

Kritischer Bereich „Semaphor“

Da eine Semaphore eine gemeinsame Variable für ihre Sender und ihre Empfänger ist, müssen wir fordern, daß die Operationen „signal“ und „wait“ mit derselben Semaphore sich wechselseitig zeitig ausschließen. Sie sind in bezug auf die Semaphore kritische Bereiche. [5]

P ***race conditions*** **V**

P überlappt P aus $s = 2$ wird ggf. $s = 1$ anstatt $s = 0$ *oder*

- ein kritischer Abschnitt wird ggf. (mehrfach) nebenläufig ausgeführt

V überlappt P aus $s = 1$ wird ggf. $s = 0$ anstatt $s = 1$ *oder*

- der unterbrochene Prozess wird ggf. bleibend blockiert

P überlappt V aus $s = 1$ wird ggf. $s = 2$ anstatt $s = 1$ *oder*

- der P ausführende Prozess wird unterbrochen und blockiert ggf. bleibend

V überlappt V aus $s = 0$ wird ggf. $s = 1$ anstatt $s = 2$ *und*

- falls Prozesse deblockiert werden, wird aus $s = 1$ ggf. $s = -1$

Alternative Semaphorstruktur

- die *blockierende Semantik* legt eine semaphoreigene Warteschlange nahe:

  Semaphorschlange führt Buch über die ein **V** erwartenden Prozesse
  - Prozesse, die in **P** blockieren, werden in die Warteschlange eingereiht
  - ein **V** hat damit direkten Zugriff auf den ggf. zu deblockierenden Prozess

- diese Variante wird oft auch als die „Standardimplementierung“ angesehen:

  Vorteile – die Schnittstelle zur Prozessorzuteilung vereinfacht sich
  - an die Prozessverwaltung werden weniger Anforderungen gestellt
  - die Auswahl des nächsten zu deblockierenden Prozesses ist effizienter

  Nachteile – die Semaphorverwaltung selbst ist (etwas) aufwendiger
  - *die Einreihungsstrategie muss verträglich sein* zur Prozessorzuteilung, d.h. *zur Strategie der Einreihung von Prozessen in die Bereitliste*

P Implementierung (2) V

„erniedrige“[1], *down*, *wait*

```
void Semaphore::prolaag () {
    lock.enter();
    if (s > 0)
        s -= 1;
    else {
        q = *(Chain*)soul;
        soul->block();
    }
    lock.leave();
}
```

erhöhe, *up*, *signal*

```
void Semaphore::verhoog () {
    Thread* gain;
    lock.enter();
    if (!(gain = (Thread*)(Chain*)q))
        s += 1;
    else
        gain->ready();
    lock.leave();
}
```

† Dabei sei q ein Semaphore-Attribut vom Typ (FIFO) Queue (siehe auch [9], „Nebenläufigkeit“, S. 10).

P Funktionen des Schedulers (2) V

`block()` legt den applizierten/laufenden Prozess schlafen

- der laufende Prozess wird blockiert (bzw. blockiert sich selbst)
- ein anderer Prozess wird ggf. zur Ausführung gebracht[2]

`ready()` setzt den applizierten Prozess laufbereit

- der Prozess wird der Strategie des *short-term scheduling* unterzogen

[2] Sollte kein anderer Prozess zur Zeit mehr lauffähig sein, wird der Prozessor in den Zustand der Untätigkeit versetzt (*idle loop*), bis eine asynchrone Programmunterbechung auftritt und mindestens einen laufbereiten Prozess hinterlassen hat. Dieser Verlauf gilt übrigens auch bereits im Zusammenhang mit `sleep(void*)` (→ p. 11) analog.

Semaphor{e,s} „*considered harmful*“

- das Semaphorkonzept ist einfach — aber auch sehr fehleranfällig:

  Verklemmungsgefahr überlappende Betriebsmittelvergabe 18–22
  - z.B. wenn Prozesse zum Zeitpunkt mehr als ein Betriebsmittel benötigen
  - *und* wenn die Betriebsmittelbelegung unabhängig und teilbar erfolgt

  Prioritätsverletzung unverträgliche Einreihungsstrategien 24–25
  - z.B. wenn die Semaphorschlange nach der FIFO-Strategie arbeitet
  - *und* wenn die Prozessorzuteilung prioritätsbasiert erfolgt

  Prioritätsumkehr blockierender Synchronisationsmechanismus 26–32
  - z.B. wenn ein Prozess niedriger Priorität einen kritischen Abschnitt belegt
  - *und* wenn ein Prozess höherer Priorität auf den Eintritt warten muss

- die Probleme sind nicht immer offensichtlich und deren Lösung selten trivial

Verklemmungsgefahr — Konvertierungsprogramm(e)

```
#include <string.h>

main (int argc, char* argv[]) {
    if (argc = 3) {
        int fd[2];
        if (acquire(&argv[1], fd) != -1) {
            if (!strcmp(argv[0], "ascii2int")) {
                /* convert from ASCII to int */
            } else if (!strcmp(argv[0], "int2ascii")) {
                /* convert from int to ASCII */
            }
        }
        release(fd);
    }
}
```

Betriebsmittelanforderung und -freigabe

```
#include <fcntl.h>
#include <unistd.h>
#include <stdlib.h>

int acquire (char* argv[2], int fd[2]) {
    fd[0] = open (argv[0], O_RDONLY);
    fd[1] = open (argv[1], O_WRONLY|O_CREAT, S_IRUSR|S_IWUSR);
    return fd[0] | fd[1];
}

void release (int fd[2]) {
    if (fd[1] != -1) close(fd[1]);
    if (fd[0] != -1) close(fd[0]);
}
```

Implementierungsskizze

`open()`

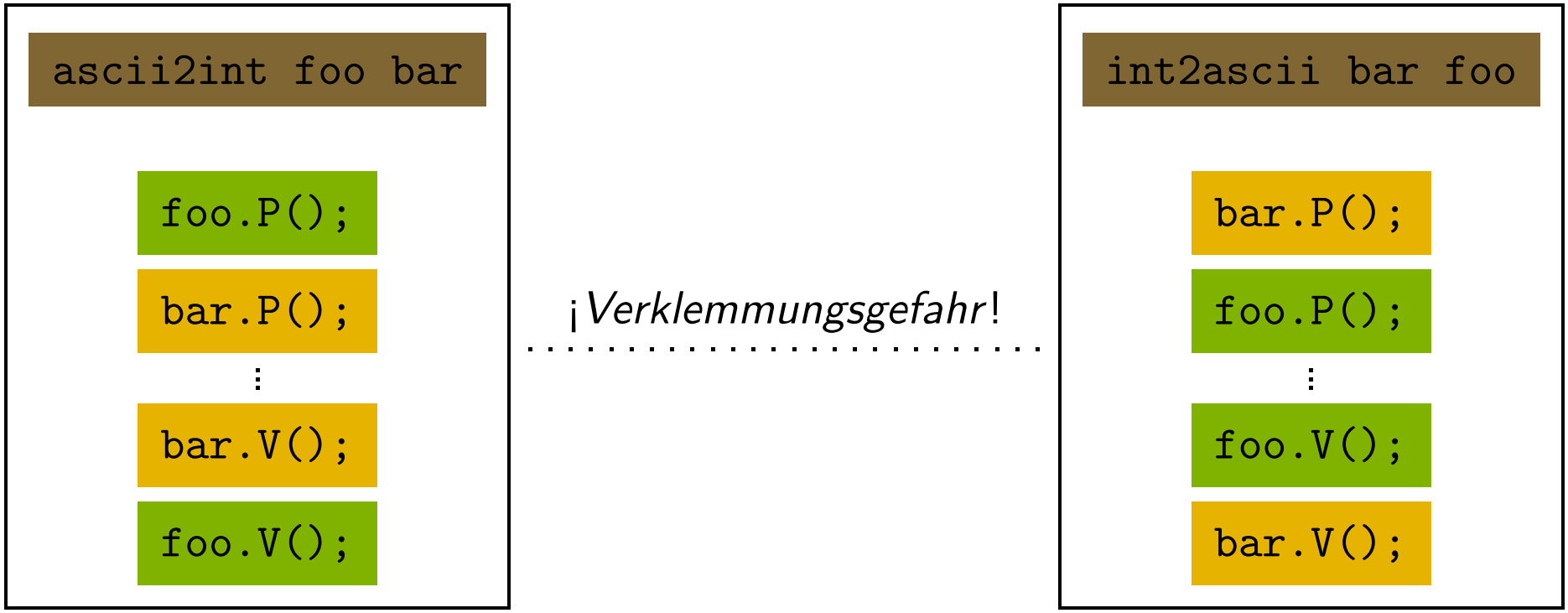
```
int open (const char* path, int flags, mode_t mode) {
    ⋮
    file->P();
    ⋮
}
```

Annahme: Die Instanz einer Datei ist ein *unteilbares Betriebsmittel*. Das soll bedeuten, dass eine Datei zu einem Zeitpunkt nur einmal vergeben und demzufolge eröffnet werden darf (`P()`). Die Freigabe (`V()`) erfolgt, wenn die Datei geschlossen wird.

`close()`

```
int close (int fd) {
    ⋮
    file->V();
    ⋮
}
```

`ascii2int` **Anwendungsszenario** `int2ascii`



Verklemmungsvorbeugung — „*mutual exclusion*“

```
#include "Semaphore.h"
Semaphore mutex(1);
```

```
int acquire (..., int fd[2]) {
    mutex.P();
    fd[0] = open (...);
    fd[1] = open (...);
    mutex.V();
    return fd[0] | fd[1];
}
```

```
void release (int fd[2]) {
    mutex.P();
    if (fd[1] != -1) close(fd[1]);
    if (fd[0] != -1) close(fd[0]);
    mutex.V();
}
```

Asynchrone Programmunterbrechungen — *interrupts*

- Szenario: ein kritischer Bereich wird von einem Interrupt getroffen
  - **P** hinterlässt Interrupts ungesperrt, sperrt ggf. aber Prozesse aus
  - die Behandlung der Unterbrechung kann somit unverzögert erfolgen
  - problematisch sind im weiteren Verlauf liegende kritische Abschnitte
  - beim Betreten dieser Abschnitte droht Verklemmungsgefahr

- eine Unterbrechungsbehandlung darf im Ablauf niemals auf ein **P** treffen
  - ein **V** dagegen ist vom Ansatz her unkritisch und ggf. höchst sinnvoll
  - das konsumierbare Betriebsmittel „Signal“ ist dadurch einfach produzierbar
  - Prozesse können sich mit **P** auf das Eintreten dieses Signals schlafen legen
  - d.h., durch **P** ist das Betriebsmittel „Interrupt“ sehr wohl konsumierbar

Prioritätsverletzung

- Annahme: die Prozessorzuteilung an Prozesse erfolgt prioritätsbasiert
  - Prozesse höherer Priorität haben Vorrang vor Prozessen niedrigerer Priorität
  - die Strategie ist auch beim Wettstreit um kritische Bereiche durchzusetzen
- FIFO-basierte Organisation der Semaphorschlange ist dazu unverträglich
  - **P** ordnet Prozesse nach der zeitlichen Reihenfolge der Eintrittswünsche
  - am Kopf der Semaphorschlange ist der nächste, Eintritt erwartende Prozess
  - dieser muss nicht die höchste Priorität aller wartenden Prozesse haben
  - **V** kann dadurch zu einer falschen Zuteilungsentscheidung führen
- die Einreihung in die Semaphorschlange muss zum Scheduling korrespondieren

Verträglicher Semaphor

- Verzicht auf die Semaphorschlange umgeht die Gefahr der Prioritätsverletzung
  - die „Originalimplementierung“ nach Dijkstra entspricht dem Ansatz (→ p. 10)
- die Alternative ist eine mit dem Scheduler gemeinsame Schlangenverwaltung
  - die ursprünglichen **P**/**V** (→ p. 10) müssen leicht modifiziert werden:
    `sleep(Q&)` um dem Scheduler zu ermöglichen, den jeweils zu blockierenden Prozess verträglich in die Semaphorschlange einzureihen
    `awake(Q&)` um dem Scheduler zu ermöglichen, den nächsten laufbereit zu setzenden Prozess verträglich der Semaphorschlange zu entnehmen
  - somit wäre jeder Semaphor dann auch eine Warteschlange des Schedulers
- in C++: `template<class Q> class Semaphore : public Q {...};`

Prioritätsumkehr — *priority inversion*

- ein Prozess niedriger Priorität bremst einen Prozess hoher Priorität aus
  - der Prozess niedriger Priorität läuft (nach **P**) im kritischen Abschnitt
  - dieser Prozess wird von einem Prozess hoher Priorität verdrängt
  - der hoch priorisierte Prozess führt ein **P** zum selben Abschnitt aus
  - er läuft erst nach dem **V** des niedrig priorisierten Prozesses weiter

- in Echtzeitsystemen ist ein solches Fehlverhalten strikt nicht zu tolerieren
  - derartige Vorgänge sind schwer zu erkennen und zu diagnostizieren
  - nicht selten sind sie erst mit Auftreten der Fehlfunktion erkennbar (→ p. 27)

- zur Vermeidung dieser Situation kommt oft *Prioritätsvererbung* [7] zum Einsatz

„What really happened on Mars?“

- Prioritätsumkehr war ein klassiches Softwareproblem im Mars Pathfinder [10]:

  bc_sched eine Task mit höchster Priorität[3]
  - kontrollierte den Aufbau der Transaktionen über den „1553“-Bus
  - dieser Bus koppelte Fahr- und Landeeinheit der Raumsode

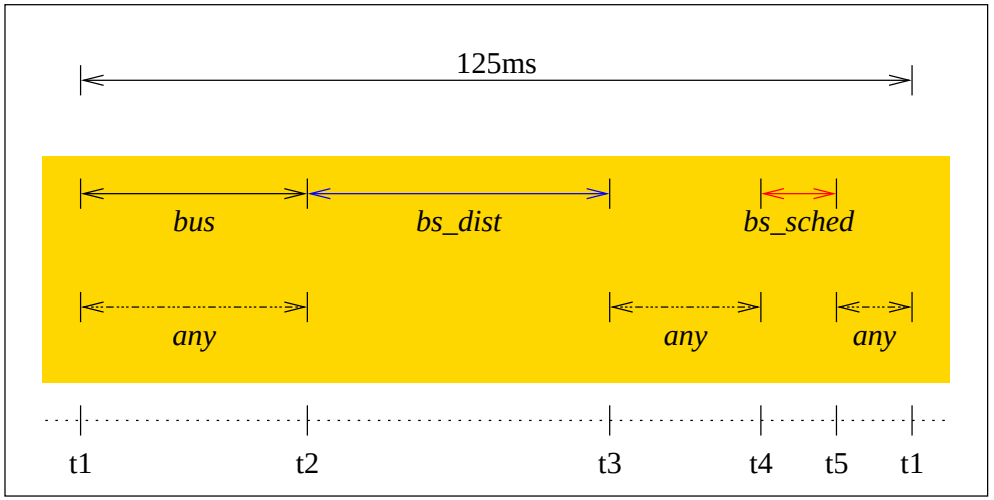
  bc_dist eine Task mit dritthöchster Priorität
  - steuerte die Einsammlung der Transaktionsergebnisse (d.h., der Daten)
  - Peripherie schrieb dazu in doppelt gepufferten gemeinsamen Speicher

  ASI/MET eine Task mit sehr niedriger Priorität
  - sammelte in seltenen Durchläufen meteorologische Daten ein
  - interoperierte mit *bc_dist* (blockierend) auf IPC-Basis

- hardware-bedingt mussten die Aktionen im 8 Hz-Rhytmus eingeplant ablaufen

[3] Mit Ausnahme der VxWorks Task „tExec“.

Mars Pathfinder



Aufbau eines Buszyklus

t1 Der Buszyklus startet hardware-kontrolliert an einer 8 Hz Grenze. Die Transaktion für diesen Zyklus wurde von *bc_sched* im vorigen Zyklus aufgesetzt.

t2 Der Busverkehr ist zur Ruhe gekommen und *bc_dist* wird aufgeweckt.

t3 *bc_dist* hat die Datenverteilung abgeschlossen.

t4 *bc_sched* wird aufgeweckt, um die Transaktion für den nächsten Buszyklus aufzusetzen.

t5 *bc_sched* hat seine Aufgabe für diesen Zyklus beendet.

Strikte Randbedingung *bc_dist* muss die Datenverteilung abgeschlossen haben, wenn *bc_sched* aufgeweckt wird, um die Transaktion des folgenden Zyklus aufzusetzen. Stellt *bc_sched* fest, dass *bc_dist* noch nicht abgeschlossen ist, wird ein Total-*reset* durchgeführt. Der *reset* hat die Initialisierung der gesamten Hard- und Software zur Folge, insbesondere den Abbruch aller bodengesteuerten Aktivitäten. Bereits aufgezeichnete wissenschaftliche Daten sind dann zwar gesichert, jedoch die noch anstehende Tagesarbeit kann nicht mehr vollbracht werden.

Mars Pathfinder — Der Fehler

- *bc_dist* (hohe Priorität) wurde durch *ASI/MET* (niedrige Priorität) blockiert:
  - *ASI/MET* besaß ein gem. wiederverwendbares, unteilbares Betriebsmittel
  - der Prozess wurde jedoch von anderen Prozessen mittlerer Priorität verdrängt
  - dadurch verlängerte sich die Blockierungszeit für *bc_dist* entsprechend
  - als Folge war *bc_dist* noch nicht abgeschlossen als *bc_sched* startete
  - *bc_sched* stellte die Zeitverletzung fest und löste einen *reset* aus

- Fehlererkennung und -beseitigung waren ingenieursmäßige Glanzleistung [10]
  - letztlich brauchte „nur“ die Semaphorinitialisierung korrigiert zu werden
  - der Semaphor wurde bodengesteuert auf Prioritätsvererbung umgestellt **!**

Prioritätsvererbung — *priority inheritance*

- die Priorität des sich im kritischen Bereich befindlichen Prozesses wird erhöht
  - Auslöser ist ein höher priorisierter Prozess, der den Bereich betreten möchte
  - die Priorität dieses Prozesses wird auf einen anderen Prozess übertragen
  - der empfangende Prozess ist der, der sich im kritischen Bereich befindet
- der kritische Bereich wird dadurch mit höherer Priorität durchlaufen
  - konsequenterweise kommt es schneller zur Bereichsfreigabe durch das **V**
  - mit dem **V** wird dem Prozess seine ursprüngliche Priorität wiedergegeben
  - dem Priorität vererbenden Prozess wird schnellerer Eintritt ermöglicht
- Sempahor und Prozessverwaltung sind umfassend aufeinander abzustimmen

Vorbeugung der Prioritätsumkehr

- Prioritätsvererbung behandelt „Symptome“ aber nicht deren Ursache
  - Vererbung und Wiederherstellung der Priorität ist relativ aufwendig
  - in extremen Fällen kann der Aufwand nicht tolerierbar sein
  - Prioritätsumkehr muss dann durch Entwurfsmaßnahmen vermieden werden
- deterministische (*offline*) Ablaufplanung schließt die „Symptome“ aus
  - die Quelltextanalyse weist problembehaftete (Echtzeit-) Programme zurück
  - sie stellt sicher, dass im System keine Prioritätsumkehr auftreten wird
  - der Ansatz ist typisch für strikte Echtzeitsysteme (*hard real-time systems*)
- allerdings ist nicht in allen Fällen die Quelltextanalyse praktikabel

Nichtblockierende Synchronisation

- Prioritätsumkehr kann nur bei blockierender Prozesssynchronisation auftreten
  - blockieren Prozesse nicht, werden andere dadurch auch nicht verzögert
  - Prozesse niedriger Priorität können dann durchgängig verdrängt werden
  - sie müssen die dann ggf. unterbrochene Aktion erneut durchlaufen[4]
  - Prozesse höherer Priorität werden also jederzeit bevorzugt bedient

- nichtblockierende Prozesssynchronisation ist jedoch nur schwer zu erreichen
  - Ansatzpunkte bilden geteilte Datenstrukturen bzw. Objekte
  - keine Ansatzpunkte bilden geteilte Kontrollstrukturen

[4] Mit prioritätsbasierter Ablaufplanung kann der einen anderen Prozess verdrängende Prozess nur höherer Priorität sein!

Zusammenfassung

- ein Semaphor ist ein grundlegendes Instrument zur Betriebsmittelvergabe
  - kontrollierte Zuteilung wiederverwendbarer/konsumierbarer Betriebsmittel
  - {im,ex}plizite Kooperation unter bzw. Kommunikation zwischen Prozessen
- Semaphor{e,s} sind definiert als „*nicht-negative ganze Zahl*“ [3]
  - mit den Operationen **P** und **V** zur Zustandskontrolle
  - ggf. erweitert um eine *Semaphorschlange* für blockierte Prozesse
- das Semaphorkonzept ist vergleichsweise einfach, aber auch sehr fehleranfällig
  - Verklemmungsgefahr, Prioritätsverletzung, Prioritätsumkehr
  - „linguistische Unterstützung“ (z.B. *monitor* [5]) beugt Fehlern vor

Referenzen

[1] K. R. Apt. Edsger Wybe Dijkstra (1930 – 2002): A Portrait of a Genius. http://arxiv.org/pdf/cs.GL/0210001, 2002.

[2] E. W. Dijkstra. Multiprogrammering en de X8, 1962. [4].

[3] E. W. 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).

[4] E. W. Dijkstra. EWD Archive: Home. http://www.cs.utexas.edu/users/EWD, 2002.

[5] P. B. Hansen. *Betriebssysteme*. Carl Hanser Verlag, erste edition, 1977. ISBN 3-446-12105-6.

[6] M. B. Jones. http://www.research.microsoft.com/∼mbj/Mars_Pathfinder, 1997.

[7] B. W. Lampson and D. D. Redell. Experiences with Processes and Monitors in Mesa. *Communications of the ACM*, 23(2):105–117, 1980.

[8] V. E. Neufeld, editor. *Webster's New World Dictionary*. Simon & Schuster, Inc., third college edition, 1988. ISBN 0-13-947169-3.

[9] W. Schröder-Preikschat. Betriebssysteme. http://www4.informatik.uni-erlangen.de, 2002.

[10] D. Wilner. Vx-Files: What really happened on Mars? Keynote at the 18th IEEE Real-Time Systems Symposium (RTSS '97), Dec. 1997. [6].