Semaphor - *semaphore*

- Eine ”nicht-negative ganze Zahl“, für die zwei Operationen definiert sind :

  P (hol. prolaag, ”erniedrige“; auch *down*, *wait*)

  - hat der Semaphor den Wert 0, wird der laufende Prozess blockiert
  - ansonsten wird der Semaphor um 1 dekrementiert

  V (hol. verhoog, erhöhe; auch *up*, *signal*)

  - inkrementiert den Semaphor um 1
  - auf den Semaphor ggf. blockierte Prozesse werden deblockiert

- Ein abstrakter Datentyp zum Austausch von Zeitsignalen zwischen gleichzeitigen Prozessen (deren Ausführung sich zeitlich überschneidet).

Semaphor (2)

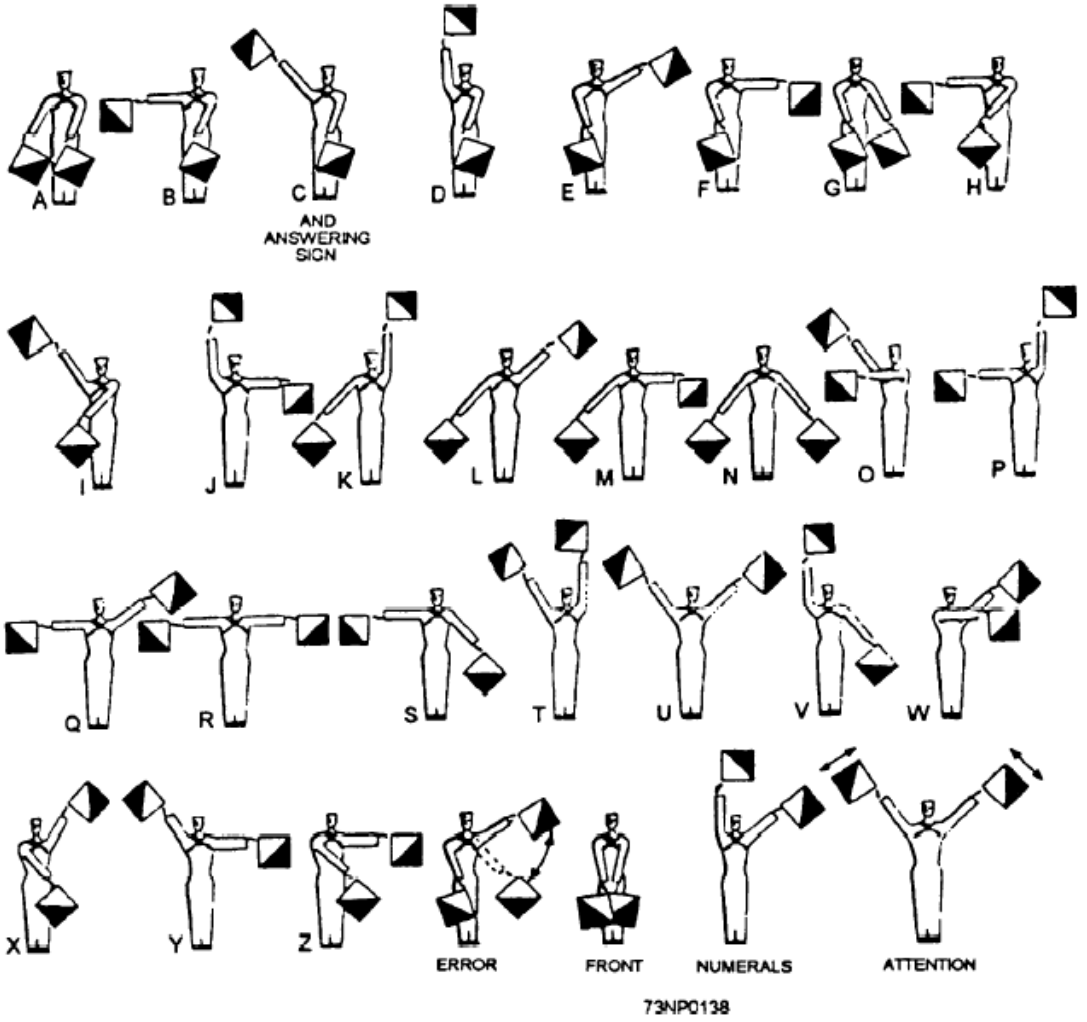

Figure AII-2.—Semaphore alphabet and special signals.

Semaphor (3)

- Webster's New World Dictionary:

  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

  . . . der Bedeutung des Wortes nach Signale, die vor der Einführung des Telegraphendienstes vor allem in der Seefahrt zur optischen Übermittlung von Nachrichten über große Entfernungen verwendet wurden.

Semaphor – Implementierung P/V

```
typedef unsigned int Semaphore;

void P (Semaphore *sema) {
    avert();
    while (*sema == 0)
        sleep(sema);
    *sema -= 1;
    admit();
  }
```

```
void V (Semaphore *sema) {
    avert();
    if ((*sema)++ == 0)
      awake(sema);
    admit();
  }
```

Verschiedentlich enthält jeder Semaphor oft zusätzlich noch eine eigene Warteschlange, die von ihm eigenständig verwaltet wird — was Vor- aber auch Nachteile haben kann. In dem Fall ”benutzt“ nämlich die Einplanungsstrategie des Schedulers die Semaphorimplementierung und umgekehrt. Die hier skizzierte Implementierung entspricht der Originalvorlage von Dijkstra, ist frei von gegenseitiger ”Benutzung“ und belässt die Ablaufkontrolle beim Scheduler.

Instrumente zur Betriebsmittelvergabe

binärer Semaphor (*binary semaphore*)

- verwaltet zu einem Zeitpunkt immer nur genau ein Betriebsmittel
  - gegenseitiger Ausschluss (mutual exclusion, mutex)
- vergibt unteilbare Betriebsmittel an Prozesse
- besitzt den Wertebereich [0, 1]

zählender Semaphor (*counting semaphore, general semaphore*)

- verwaltet zu einem Zeitpunkt mehr als ein Betriebsmittel (derselben Art)
- vergibt teil- bzw. konsumierbare Betriebsmittel an Prozesse
- besitzt den Wertebereich [0,N], für N Betriebsmittel

Arten von Betriebsmitteln

wiederverwendbare Betriebsmittel werden angefordert und freigegeben

- ihre Anzahl ist begrenzt: Prozessoren, Geräte, Speicher (z.B. Puffer)
- **teilbar** wenn zu einer Zeit von **mehreren** Prozessen belegbar
- **unteilbar** wenn zu einer Zeit von nur **einem** Prozess belegbar

konsumierbare Betriebsmittel werden erzeugt und zerstört

- ihre Anzahl ist (logisch) unbegrenzt: Signale, Nachrichten, Interrupts:
- **Produzenten** können beliebig viele davon erzeugen
- **Konsumenten** zerstören sie wieder bei Inanspruchnahme
- sind abhängig vom Produzenten und werden ggf. blockierend warten

Ausschließender Semaphor

```
void thread1 () {
    P(&mutex);
    ...
    V(&mutex);
  }
```

```
void thread2 () {
    P(&mutex);
    ...
    V(&mutex);
  }
```

```
Semaphore mutex = 1;
```

mehrseitige Synchronisation der Initialwert des Semaphors gibt die Anzahl der Prozesse an, die maximal zu einer Zeit den kritischen Abschnitt durchlaufen und darin Berechnungen durchführen dürfen ($\Rightarrow$ *wiederverwendbare Betriebsmittel*)

$\Rightarrow$ **unteilbares Betriebsmittel** : der Initialwert des Semaphors ist 1

Signalisierender Semaphor

```
void thread1 () {
    ...
    P(&mutex);
    ...
  }
```

```
void thread2 () {
    ...
    V(&mutex);
    ...
  }
```

```
Semaphore mutex = 0;
```

einseitige Synchronisation einander zugeordnete Semaphoroperationen werden von verschiedenen Prozessen ausgeführt ($\Rightarrow$ *konsumierbare Betriebsmittel*):

- **P** von dem Prozess (*Konsument*), der das Eintreten einer Bedingung erwartet und dadurch das zugehörige Signal konsumieren wird
- **V** von dem anderen Prozess (*Produzent*), der das Eintreten der Bedingung anzeigen muss und dadurch das zugehörige Signal produziert

Bounded Buffer (7) - Semaphor

```
struct buffer {
    Any data[NDATA];
    unsigned nput;
    unsigned lget;
    Semaphor lock;
    Semaphor free;
    Semaphor full;
  }
```

```
void reset (struct buffer *bufp) {
    bufp->nput = 0;
    bufp->lget = NDATA - 1;
    bufp->lock = 1;
    bufp->free = NDATA;
    bufp->full = 0;

}
```

- ein *wiederverwendbares* Betriebsmittel für *konsumierbare* Betriebsmittel:

  binärer Semaphor `lock` zum gegenseitigen Ausschluss

  zählende Semaphore `free` und `full` zur Vermeidung von Über-/Unterlauf

Bounded Buffer (8) – Ein/Mehrseitige Synchronisation

```
void store (struct buffer *bufp, Any item)
   {
     P(&bufp->lock);
     P(&bufp->free);
     ...
     V(&pufp->lock);
     V(&bufp->full);
   }

void fetch (struct buffer *bufp, Any *item)
   {
     P(&bufp ->lock);
     P(&bufp->full);
     ...
     V(&bufp->lock);
     V(&bufp->free);
   }
```

- einseitig:

  free Produzent wartet auf ggf. auf Konsument

  full Konsument wartet ggf. auf Produzent

- mehrseitig

  lock Produzenten und Konsumenten warten ggfs. auf andere Produzenten oder Konsumenten

Semaphore „*considered harmful*“

- auf Semaphore basierende Lösungen werden schnell komplex und fehleranfällig
  - kritische Abschnitte neigen dazu, mit ihren Synchronisations-anweisungen quer über das nicht-sequentielle Programm verstreut vorzuliegen
  - das Schützen gemeinsamer Variablen bzw. Freigeben kritischer Abschnitte kann dabei leicht vergessen werden
- die Gefahr der Verklemmung (*deadlock*) nebenläufiger Prozesse ist recht hoch
  - umso zwingender ist die Notwendigkeit von Verfahren zur Vorbeugung, Vermeidung und/oder Erkennung solcher Verklemmungen
  - nicht-blockierende Synchronisation ist nicht immer durchgängig praktizierbar
- ”linguistische Unterstützung“ ($\Rightarrow$ *Monitor*) beugt den möglichen Fehlern vor

Monitor

- Ein abstrakter Datentyp mit impliziten Synchronisationseigenschaften:

  mehrseitige Synchronisation an der Schnittstelle zum Monitor

  - gegenseitiger Ausschluss der Ausführung aller Schnittstellenfunktionen

  einseitige Synchronisation innerhalb des Monitors (Bedingungsvariable)

    wait blockiert einen Prozess auf das Eintreten eines Signals/einer Bedingung und gibt den Monitor implizit wieder frei

    signal zeigt das Eintreten eines Signals/einer Bedingung an und deblockiert ggf. (genau einen oder alle) darauf blockierte Prozesse

- Sprachgestützter Mechanismus: Concurrent Pascal, PL/I, CHILL, . . . , Java.

Monitor == Modul/Klasse

Ein Modulkonzept/Klassenbegriff erweitert um eine Synchronisationssemantik:

- die Prozeduren eines Monitors schließen sich bei konkurrierenden Zugriffen durch mehrere Prozesse (auf eben diesen Monitor) gegenseitig aus
  - der erfolgreiche Prozeduraufruf sperrt den Monitor
  - bei Prozedurr[ckkehr wird der Monitor wieder entsperrt
  - ein Kompilierer setzt die daf¨ur notwendigen Anweisungen ab
- Monitorprozeduren stellen per Definition kritische Abschnitte dar
  - deren Integrität wird vom Kompilierer garantiert
  - die ”Klammerung“ kritischer Abschnitte erfolgt automatisch

Einseitige Synchronisation - *wait*

- notwendiger Seiteneffekt beim Warten ist die implizite Freigabe des Monitors
  - andere Prozesse wären sonst weiterhin an den Monitoreintritt gehindert
  - als Konsequenz könnte die zu erfüllende Bedingung nie erfüllt werden
  - der sich schlafenlegende Prozess würde nie mehr erwachen == *deadlock*
- desweiteren sind Monitordaten in einem konsistenten Zustand zu hinterlassen
  - andere Prozesse werden den Monitor während der Blockadephase betreten
  - als Folge davon sind (je nach Funktion) Zustandsänderungen zu erwarten
  - vor Eintritt in die Wartephase muss der Datenzustand konsistent sein
- aktives Warten im Monitor wäre logisch komplex **und** ist leistungsmindernd

Einseitige Synchronisation - *signal*

- die Operation signalisiert die Erfüllung einer Wartebedingung und bewirkt ggf.
  - die Deblockierung mindestens eines Prozesses
  - im Falle wartender Prozesse sind als Anforderungen zwingend zu erfüllen:
    - wenigstens ein Prozess deblockiert an der Bedingungsvariablen und
    - höchstens ein Prozess rechnet nach der Operation im Monitor weiter
  - es gibt verschiedene Lösungsvarianten, jeweils mit verschiedener Semantik
    - Anzahl der befreiten Prozesse (d. h., alle oder nur einer)
    - Besitzwechsel des Monitors, kein Besitzwechsel (Besitzwahrung)
- erwartet kein Prozess ein Signal/eine Bedingung, ist die Operation wirkungslos
  - d. h., Signale dürfen in Bedingungsvariablen nicht gespeichert werden

Semantiken der Signalisierung (1)

genau einen wartenden Prozess befreien . . . nur welchen?

- bei mehr als einen wartenden Prozess ist eine Auswahl zu treffen
- die Auswahlentscheidung muss im Ergebnis der Fadeneinplanung entsprechen
- ggf. ist bereits bei Prozessblockierung möglichen Konflikten vorzubeugen

alle wartenden Prozesse befreien ($\Rightarrow$ Hansen)

- die Auswahlentscheidung ist unter alleiniger Kontrolle des Schedulers
- Konflikte, die der Fadeneinplanung entgegenwirken, werden ausgeschlossen
- verschiedene Belange sind voneinander getrennt (separation of concerns)

$\Rightarrow$ in beiden Fällen ist die Neuauswertung der Wartebedingung notwendig

$\Rightarrow$ die signalisierten Prozesse bewerben sich erneut um den Monitorzutritt

Semantiken der Signalisierung (2)

Wechsel vom signalisierenden zum signalisierten Prozess ($\Rightarrow$ Hoare)

- genau einer von ggf. mehreren wartenden Prozessen wird signalisiert
  - der signalisierte Prozess setzt seine Berechnung sofort im Monitor fort
  - als Konsequenz muss der signalisierende Prozess den Monitor verlassen
- dem signalisierten Prozess wird seine Fortführungsbedingung garantiert
  - seit Signalisierung konnte kein anderer Prozess den Monitor betreten
  - demzufolge konnte auch kein anderer Prozess die Bedingung entkräften

$\Rightarrow$ die Neuauswertung der Wartebedingung entfällt

$\Rightarrow$ eine erhöhte Anzahl von Fadenwechseln ist in Kauf zu nehmen

$\Rightarrow$ der signalisierende Prozess bewirbt sich erneut um den Monitorzutritt

Monitor vs. Semaphor

- von mehreren Prozessen gemeinsam bearbeitete Daten müssen in Monitoren organisiert vorliegen
  - die Programmstruktur macht die kritischen Abschnitte explizit sichtbar
  - wie auch die zulässigen (an zentraler Stelle definierten) Zugriffsfunktionen
- wie ein Modul, so kapselt auch ein Monitor f¨ur mehrere Funktionen Wissen über gemeinsame Daten
  - information hiding, d. h., Datenabstraktion wird unterstützt
  - Auswirkungen lokaler Programmänderungen bleiben (eng) begrenzt

$\Rightarrow$ ein Monitor ist Konzept der Ebene 5, ein Semaphor Konzept der Ebene 3

Bounded Buffer (7) - „Concurrent C++“

```
Monitor buffer {
    Any data[NDATA];
    unsigned nput;
    unsigned lget;
    condition free;
    condition full;
  public:
    Buffer();
    void store(Any item);
    void fetch(Any& item);
  }
```

```
Buffer::Buffer {
    bufp->nput = 0;
    bufp->lget = NDATA - 1;
    }
```

- Der Konstruktor `Buffer::Buffer()` wird bei der Instanzenbildung eines Monitors vom Typ Buffer automatisch aufgerufen und initialisiert die Monitorvariablen. Die mit monitor implizit vorhandene Schlossvariable wie auch die beiden Bedingungsvariablen (condition) erhalten die Werte 1 (”lock“) bzw. `0` (`free, full`) entsprechend ihrer Bedeutung automatisch zugewiesen

Bounded Buffer (10) - Hoare´scher Monitor

```
void Buffer::store (Any item) {
    if (nput == lget) free.wait();
    data[nput]  = item;
    nput = (nput + 1) % NDATA;
    full.signal();
   }

void Buffer::fetch (Any *item) {
    if ((lget + 1) % NDATA == nput)
      full.wait();
    lget = (lget + 1) % NDATA;
    *item = data[lget];
    free.signal();
   }
```

- Dem signalisierten Prozess wird garantiert, dass er nach seiner Deblockierung die Bedingung für seine Fortführung vorfindet, da kein anderer Prozess in der Zwischenzeit den Monitor betreten konnte:
  - die Wartebedingung ist nur einmal zu prüfen (`if`).

Bounded Buffer (10) - Hansen´scher Monitor

```
void Buffer::store (Any item) {
    while (nput == lget) free.wait();
    data[nput]  = item;
    nput = (nput + 1) % NDATA;
    full.signal();
   }

void Buffer::fetch (Any *item) {
    while ((lget + 1) % NDATA == nput)
      full.wait();
    lget = (lget + 1) % NDATA;
    *item = data[lget];
    free.signal();
   }
```

- ”Hoare’sche Garantie“ erhält ein signalisierter Prozess nicht:
  - die Wartebedingung ist wiederholt zu prüfen (`while`).
- Dafür werden falsche Signalisierungen aber toleriert.

Bounded Buffer (11) - Unified Monitor

```
void Buffer::store (Any item) {
    when (nput == lget) free.wait();
    data[nput]  = item;
    nput = (nput + 1) % NDATA;
    full.signal();
   }

void Buffer::fetch (Any *item) {
    when ((lget + 1) % NDATA == nput)
      full.wait();
    lget = (lget + 1) % NDATA;
    *item = data[lget];
    free.signal();
   }
```

- Je nach Monitorart wertet der Monitor mit `when` die Bedingung ein- oder mehrmals aus.
  - kein syntaktischer Unterschied sichtbar
  - semantischer Unterschied bleibt

Prioritätsverletzung

Annahme: Fadeneinplanung erfolgt prioritätsbasiert

- Prozesse höherer Priorität haben Vorrang vor Prozessen niedrigerer Priorität
- die CPU-Warteschlange ist absteigend nach Prozessprioritäten sortiert

Problem: FIFO-Warteschlange(n) bei ein-/mehrseitiger Synchronisation

- berücksichtigt die zeitliche Reihenfolge der Eintrittswünsche in einen KA
  - am Kopf der KA-Warteschlange ist der nächste zu deblockierende Prozess
  - dieser muss nicht die höchste Priorität aller wartenden Prozesse haben
- die Deblockierung kann eine falsche Zuteilungsentscheidung zur Folge haben

Konsequenz: gleiche Einreihungsverfahren für KA- und CPU-Warteschlangen: enge Verzahnung von Koordination und Einplanung

Prioritätsumkehr – *priority inversion*

Problem: mindestens drei nebenläufige Prozesse unterschiedlicher Priorität

low belegt den kritischen Abschnitt KA

high verdrängt low und bewirbt sich um KA $\Rightarrow$ blockiert

middle verdrängt low für unbestimmte Zeit: da *high* auf *low* wartet und *low* von *middle* verdrängt wurde, muss *high* auch auf *middle* warten

$\Rightarrow$ *middle* dominiert über *high* ; Widerspruch zur Einplanungsstrategie

Konsequenz: Prioritätsvererbung (*priority inheritance*)

- beim Blockieren vererbt *high* seine Priorität an *low*, der auf seine alte Priorität beim Verlassen des KA zurückwechselt
- *low* kann nicht mehr von *middle* verdrängt werden, *high* kommt voran

$\Rightarrow$ aufwendige Lösungen für komplexe Systeme

Blockierende Synchronisation „considered harmful“

Leistung (*performance*) insb. in SMP-Systemen ist teils stark beeinträchtigt

- ”spin locking“ (while (tas(lock));) reduziert massiv die Busbandbreite

Robustheit (*robustness*): single point of failure

- ein im KA scheiternder Prozess, kann das ganze System lahm legen

Einplanung (*scheduling*) wird behindert bzw. nicht durchgesetzt

- un- bzw weniger wichtige Prozesse können wichtige Prozesse ”ausbremsen“
- Prioritätsverletzung, Prioritätsumkehr (Beispiel: Mars Pathfinder)

Verklemmung (*deadlock*) einiger oder sogar alller Prozesse

Nichtblockierende Synchronisation

nicht-blockierende Algorithmen garantieren die Ausführung einiger sich überlappender Operationen auf gemeinsamen Daten in endlicher Zeit

- **überlappenden** (d. h., verdrängenden) Prozessen gelingt die Operation
  - sie werden nicht verzögert und genießen Vorrang
- **überlappte** (d. h., verdrängte) Prozesse wiederholen die Operation
  - sie werden verzögert und können ggf. aushungern (starvation)
- Grundlage bilden Spezialbefehle der CPU: z. B. `cas, cmpxchg, ll/sc`
  - Komplexbefehle mit unteilbarem ”read-modify-write“-Zyklus

$\Rightarrow$ die mit blockierenden Verfahren bestehenden Probleme werden ausgeschlossen

$\Rightarrow$ die Alternativlösungen sind oft jedoch logisch (erheblich) komplizierter

$\Rightarrow$ nicht alle Koordinierungsaufgaben können so gelöst werden . . .

„Wiederholtes Versuchen“ ⇔ „Aktives Warten“

- überlappte kritische Operationen auf gemeinsame Daten werden wiederholt:
  - der aktive Prozess ist weiterhin ”beschäftigt“, bis die erste Wiederholung der betreffenden Operation für ihn erfolgreich war
    - er betreibt praktisch aktives Warten, was eigentlich zu vermeiden ist
- Prozesse sollen auf die Zuteilung von Betriebsmitteln nicht aktiv warten
  - wann ist das Betriebsmittel
    - ... ”Puffer“ wiederverwendbar?
    - ... ”Nachricht“ konsumierbar?
    - ... ”CPU“ verfügbar?
- Prozesse dürfen aber wiederholt versuchen, gemeinsame Daten zu aktualisieren

Dualität von Koordinierungstechniken

Problem	**Methode**
gegenseitiger Ausschluß	Schloßvariablen, blockierungsfreie Algorithmen
explizite Prozesssteuerung	Bedingsvariable
bedingte Verzögerung	bedingter kritischer Abschnitt
Austausch von Zeitsignalen	Semaphor
Austausch von Daten	Nachrichtenpuffer (Bounded Buffer)

- **logisch betrachtet** sind alle Methoden äquivalent, da jede von ihnen hilft, ein beliebiges Steuerungsproblem zu lösen
- **praktisch betrachtet** sind die Methoden nicht äquivalent, da einige von ihnen für ein gegebenes Problem zu komplexen und ineffzienten Lösungen führen

Zusammenfassung

- Synchronisation ist die Koordination von Kooperation und Konkurrenz
  - unterschieden werden **blockierende** und **nicht-blockierende** Verfahren
- blockierende Verfahren lassen Prozesse passiv warten
  - Schlossvariable, Bedingungsvariable, Semaphor, Monitor
- nicht-blockierende Verfahren profitieren von Spezialbefehlen der CPU
  - CISC $\Rightarrow$ cas, cas2 (dcas), cmpxchg
  - RISC $\Rightarrow$ ll/sc

$\Rightarrow$ nicht-sequentielle Programmierung ist nicht nur ein Betriebssystemfall