# Teil VIII

## Koordination von Kooperation und Konkurrenz





## Überblick

Synchronisation
- Verfahrensweisen
- Schlossvariable
- Bedingungsvariable
- Semaphor
- Monitor
- Zusammenfassung

Verklemmung
- Grundlagen
- Vorbeugung
- Vermeidung
- Erkennung und Erholung
- Zusammenfassung





## Koordinierung ≡ „Reihenschaltung“

Koordination der Kooperation und Konkurrenz zwischen Prozessen ⇝ Synchronisation

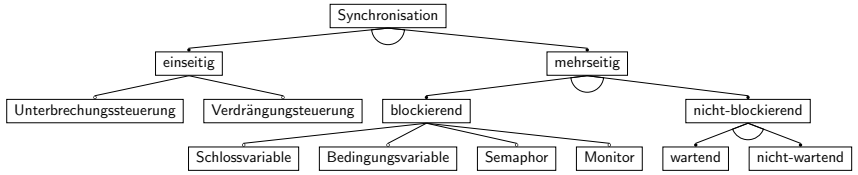
ko·or·di'nie·ren beiordnen; in ein Gefüge einbauen; aufeinander abstimmen; nebeneinanderstellen; Termine ∼.

- sich ggf. überlappende Aktivitäten der Reihe nach ausführen
  S. V-78 Nebenläufigkeit, kritischer Abschnitt
  - nebenläufiges Zählen (asynchrone Programmunterbrechung)
  - Verwaltung der Bereitliste (verdrängende Prozesseinplanung)
- „der Reihe nach“ meint, die Verzögerung von Prozessen erzwingen
  - die überlappende oder die überlappte Aktivität, je nach Verfahren

**Synchronisationsverfahren...**
- wirken einseitig oder mehrseitig
  - und blockierend oder nicht-blockierend
    - und wartend oder nicht-wartend

## Arten der Synchronisation

Klassifikation




Synchronisationsverfahren, die nachfolgend betrachtet werden, arbeiten...

einseitig Unterbrechungssteuerung, Verdrängungsteuerung

mehrseitig blockierend (die „Klassiker“), nicht-blockierend (wartend)

## Einseitige Synchronisation
Unilateral

Auswirkung haben die Verfahren nur auf einen der beteiligten Prozesse:

Bedingungssynchronisation
- der Ablauf des einen Prozesses ist abhängig von einer Bedingung
- der andere Prozess erfährt keine Verzögerung in seinem Ablauf

logische Synchronisation
- die Maßnahme resultiert aus der logischen Abfolge der Aktivitäten
- vorgegeben durch das „Rollenspiel“ der beteiligten Prozesse

- andere Prozesse sind jedoch nicht gänzlich unbeteiligt:
  - die Veränderung einer Bedingung, auf die ein Prozess wartet, ist z.B. von einem anderen Prozess herbeizuführen





## Mehrseitige Synchronisation
Multilateral

Auswirkung haben die Verfahren ggf. auf alle beteiligten Prozesse:
- welche Prozesse verzögert werden, ist i.A. unvorhersehbar
- allgemein gilt: „wer zuerst kommt, mahlt zuerst“
  - d.h., schreitet (relativ) unverzögert in seiner weiteren Ausführung fort

Prinzip vom gegenseitigen Ausschluss (engl. *mutual exclusion*)
- erzwungen wird die atomare Ausführung von Anweisungsfolgen
  - d.h. von Programmsequenzen, die einen kritischen Abschnitt bilden
- „abschnittweise“ wird die CPU exklusiv von einem Prozess genutzt
  - der kritische Abschnitt wird „unteilbar durchlaufen“

☞ Modularisierung kritischer Abschnitte schafft Elementaroperationen





## Unterbrechungssynchronisation
Typischer Fall von einseitiger Synchronisation

Unterbrechungen verhindern oder tolerieren, durch Verzögerung der...

überlappenden Aktivität Unterbrechungssteuerung

hart Spezialbefehle der Ebene 2: `cli`, `sti` (x86)

weich ohne Spezialbefehle, z.B. „Schleusen“ [62]

überlappten Aktivität nicht-blockierende Synchronisation
- Spezialbefehle der Ebene 2:
  - CISC `cas` (IBM 370, m68020+), `cmpxchg` (i486+)
  - RISC `ll/sc` (DEC Alpha, MIPS, PowerPC)
- erweitert um wartebehaftete Algorithmen

Unterbrechungen sperren ist einfach, jedoch nicht immer zweckmäßig
- Faustregel: harte Synchronisation ist möglichst zu vermeiden





## Unterbrechungssynchronisation (Forts.)
Wiedersehen mit einem alten Problem (S. III-49): nebenläufiges Zählen

```
unsigned int wheel = 0;

void __attribute__ ((interrupt)) train () {
    wheel++;
}

int main () {
    for (;;)
        printf("%10u", incr(&wheel));
}
```

`main` die Elementaroperation `incr()` stellt unteilbares Zählen sicher

`train` unterbricht den laufenden Prozess — nicht beliebig...
- die verschachtelte Unterbrechung muss ausgeschlossen sein

## Unterbrechungssynchronisation (Forts.)

Verhinderung vs. Tolerierung von asynchronen Programmunterbrechungen

Verhinderung

```
int incr (int *ip) {
    int bar;
    asm ("cli");
    bar = *ip += 1;
    asm ("sti");
    return bar;
}
```

- harte Synchronisation
- Ereignisverlust droht
  - *Interrupts* sind ausgeschlossen

Tolerierung

```
int incr (int *ip) {
    int foo, bar;
    do {
        bar = (foo = *ip) + 1;
    } while (!cas(ip, foo, bar));
    return bar;
}
```

- nicht-blockierende Synchronisation
- wartebehafteter Algorithmus
  - der unterbrochene Prozess wird ggf. unbestimmt lang verzögert





## Multiprozessorsynchronisation

Vergleichen und bedingt überschreiben (engl. *compare and swap*, CAS)

```
bool cas (word *ref, word old, word new) {
    bool srZ;
    atomic();
    if (srZ = (*ref == old)) *ref = new;
    cimota();
    return srZ;
}
```

Komplexbefehl (einer CPU), der scheitern kann:

| | |
|---|---|
| true | Operation ist gelungen, das Speicherwort wurde überschrieben |
| false | Operation ist gescheitert |

Elementaroperation eines CISC, atomarer „*read-modify-write*“-Zyklus:

`atomic()` verhindert (Speicher-) Buszugriffe durch andere Prozessoren

`cimota()` lässt (Speicher-) Buszugriffe anderer Prozessoren wieder zu

Lese-/Schreibzyklen des Prozessors werden unteilbar ausgeführt

- auf *Interrupts* wird, wie sonst auch, erst am Befehlsende reagiert





## Gegenseitiger Ausschluss

Kennzeichnend für mehrseitige Synchronisation

**Kritischer Abschnitt (KA) [55, S. 137]**

- sich gegenseitig ausschließende Aktivitäten werden nie parallel ausgeführt und verhalten sich zueinander, als seien sie unteilbar, weil keine Aktivität die andere unterbricht
- Anweisungen, deren Ausführung einen gegenseitigen Ausschluß erfordert, heißen kritische Abschnitte
  - engl. *critical sections*, *critical regions*

„Synchronisationsklammern“ werden verwendet, um kritische Abschnitte vor nebenläufigen Ausführungen zu schützen

- Schlossvariable, Bedingungsvariable, Semaphor, Monitor





## Kritischer Abschnitt

Protokolle regeln den Ein- und Austritt

Betreten (engl. *enter*) und Verlassen (engl. *leave*) kritischer Abschnitte unterliegen bestimmten Verhaltensregeln:

Eintrittsprotokoll (engl. *entry protocol*)

- regelt die Belegung eines kritischen Abschnitts durch einen Prozess
  - erteilt einem Prozess die Zugangsberechtigung
- bei bereits belegtem kritischen Abschnitt wird der Prozess verzögert

Austrittsprotokoll (engl. *exit protocol*)

- regelt die Freigabe des kritischen Abschnitts durch einen Prozess
- Prozesse können den kritischen Abschnitt (wieder) belegen

☞ die Vorgehensweisen variieren mit dem jew. Synchronisationsverfahren

# Schlossvariable
(engl. *lock variable*)

Ein abstrakter Datentyp, auf dem zwei Operationen definiert sind:

acquire (auch: *lock*) $\models$ Eintrittsprotokoll
- verzögert einen Prozess, bis das zugehörige Schloss offen ist
  - bei geöffnetem Schloss fährt der Prozess unverzögert fort
- verschließt das Schloss („von innen“), wenn es offen ist

release (auch: *unlock*) $\models$ Austrittsprotokoll
- öffnet ein Schloss, ohne den öffnenden Prozess zu verzögern

Implementierungen dieses abstrakten Datentyps werden auch als Schlossalgorithmen (engl. *lock algorithms*) bezeichnet





# Schlossalgorithmus
Prinzip — mit Problem(en)

```
void acquire (bool *lock) {
    while (*lock);
    *lock = 1;
}

void release (bool *lock) {
    *lock = 0;
}
```

- kritisch ist die Phase vom Verlassen der Kopfschleife bis zum Setzen der Schlossvariablen
- Verdrängung des laufenden Prozesses kann einem anderen Prozess ebenfalls das Schloss geöffnet vorfinden lassen
- im weiteren Verlauf könnten (mindestens) zwei Prozesse den eigentlichen zu schützenden kritischen Abschnitt überlappt ausführen

Schutz eines kritischen Abschnitts bildet selbst einen kritischen Abschnitt:

acquire muss als Elementaroperation implementiert sein
- das Eintrittsprotokoll muss unteilbar ausgeführt werden





# Schlossalgorithmus (Forts.)
Totale Unterbrechungssteuerung

```
void acquire (bool *lock) {
    avertIRQ();
    while (*lock) {
        admitIRQ();
        avertIRQ();
    }
    *lock = 1;
    admitIRQ();
}
```

```
void avertIRQ () { asm("cli"); }
void admitIRQ () { asm("sti"); }
```

- Überprüfen und Schließen des Schlosses verläuft unteilbar
  - nur bei Monoprozessorsystemen
- der Schleifenrumpf muss jedoch teilbar sein, damit der Planer aufgerufen werden und ggf. eine Umplanung vornehmen kann

*Interrupts* werden abgewendet, obwohl im Zuge ihrer Behandlung der überlapppte Prozess nie einen Schlossalgorithmus durchlaufen dürfte
- darüberhinaus können flankengesteuerte *Interrupts* verloren gehen





# Schlossalgorithmus (Forts.)
Totale Verdrängungssteuerung

```
void acquire (bool *lock) {
    avert();
    while (*lock) {
        admit();
        avert();
    }
    *lock = 1;
    admit();
}
```

```
void avert () { preemption = 0; }
void admit () { preemption = 1; }
```

- Überprüfen und Schließen des Schlosses verläuft unteilbar
- der Schleifenrumpf muss jedoch teilbar sein, damit der laufende Prozess verdrängt werden kann

Verdrängungsereignisse werden abgewendet, obwohl genau nur einer von ggf. vielen Prozessen das Schloss öffnen wird
- darüberhinaus arbeitet das System nicht voll verdrängend

## Schlossalgorithmus (Forts.)

Spezialbefehl des Prozessors

```
void acquire (bool *lock) {
    while (tas(lock));
}
```

- Überprüfen und Schließen des Schlosses verläuft unteilbar

tas (*test and set*) testet den Inhalt der adressierten Speicherzelle und setzt ihren Wert auf 1, wenn der Wert 0 ist:

- `return *lock ? 1 : !(*lock = 1);`
- atomarer Maschinenbefehl für Ein- oder Mehrprozessorsysteme

### „Drehschloss“, Umlaufsperre (engl. *spin lock*)

Vorsicht ist geboten, im Falle eines Mehrprozessorsystems:

- pausenloses Schleifen hindert andere Prozessoren am Buszugang
  - im Schleifenrumpf muss eine Pause eingelegt werden — nur wie lange?
- starke Leistungseinbußen können die Folge sein [63]





## Multiprozessorsynchronisation

Bedingtes setzen (engl. *test and set*, TAS)

```
bool tas (bool *flag) {
    bool old;
    atomic();
    old = *flag;
    *flag = 1;
    cimota();
    return old;
}
```

Komplexbefehl (einer CPU), der den aktuellen Wert einer Schlossvariablen liefert und diese (auf 1) setzt:

true das Schloss ist bereits verschlossen
- die Schlossvariable ist unverändert

false das Schloss wurde verschlossen
- die Schlossvariable wurde verändert

Analogie zu CAS (S. VIII-10): atomarer „read-modify-write“-Zyklus

- Lese-/Schreibzyklen des Prozessors werden unteilbar ausgeführt
  - auf *Interrupts* wird, wie sonst auch, erst am Befehlsende reagiert





## Aktives Warten

(engl. *busy waiting*)

Unzulänglichkeit der Schlossalgorithmen: der aktiv wartende Prozess...

- kann keine Änderung der Bedingung herbeiführen, auf die er wartet
- behindert unnütz andere Prozesse, die sinnvolle Arbeit leisten könnten
- schadet damit letztlich auch sich selbst

Je länger der Prozess den Prozessor für sich behält, umso länger muss er darauf warten, dass andere Prozesse die Bedingung erfüllen, auf die er selbst wartet.

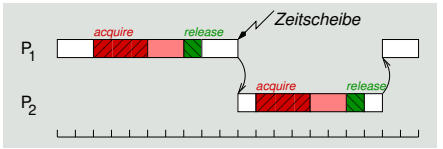
- in den meisten Fällen sind Effizienzeinbußen in Kauf zu nehmen
  - es sei denn, jeder Prozess hat seinen eigenen realen Prozessor
  - ist nicht unrealistisch: „*gang scheduling*“ und „*barrier synchronization*“



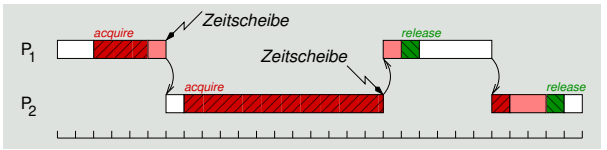


## Aktives Warten ohne Prozessorabgabe

„*Spin locking considered harmful*“




| | $T_s$ | $T_q$ | $T_q/T_s$ |
|---|---|---|---|
| $P_1$ | 12 | 12 | 1.0 |
| $P_2$ | 8 | 8 | 1.0 |




| | $T_s$ | $T_q$ | $T_q/T_s$ |
|---|---|---|---|
| $P_1$ | 12 | 24 | 2.0 |
| $P_2$ | 17 | 23 | 1.35 |

Faustregel: in der Warteschleife die Kontrolle über den Prozessor abgeben

laufend $\mapsto$ bereit in Laufbereitschaft bleiben (S. VII-17)

laufend $\mapsto$ blockiert selektive Verdrängungssteuerung

# Aktives Warten mit Prozessorabgabe

Kooperative Ausführung der Warteschleife

```
void acquire (bool *lock) {
    while (tas(lock))
        relinquish();
}
```

- der laufende Prozess gibt freiwillig den Prozessor ab, wenn die Schlossvariable nicht gesetzt werden konnte

- die Effektivität des Ansatzes hängt ab von der Umplanungsstrategie:
  - RR der aufgebende Prozess kommt ans Ende der Bereitliste ✓
  - sonst seine (stat./dyn.) Priorität bestimmt seine Listenposition ?
    - damit erhält er mehr oder weniger schnell wieder den Prozessor
    - hat er die höchste Priorität, gibt er den Prozessor nicht ab
    - seine Wartepriorität muss niedriger sein als seine Laufpriorität
- suboptimal: wartende Prozesse belasten nur seltener den Prozessor
  - besser wäre, wenn wartende Prozesse „schlafen“, d.h., blockieren





# Aktives Warten mit Prozessorabgabe (Forts.)

Selektive Verdrängungssteuerung

```
void acquire (bool *lock) {
    tired();
    while (tas(lock))
        sleep(lock);
    awake();
}

void release (bool *lock) {
    lock = 0;
    rouse(lock);
}
```

Zustandsmaschine der Prozesseinplanung:

`tired()` unterbindet zeitweilig die mögliche Verdrängung des laufenden Prozesses

`sleep()` blockiert den laufenden Prozess auf das angegebene Ereignis

`awake()` ermöglicht die Verdrängung des laufenden Prozesses

- der laufende Prozess legt sich schlafen, wenn das Setzen der Schlossvariablen scheitert
- bei Freigabe des kritischen Abschnitts werden all die Prozesse aufgeweckt, die auf das Freigabeereignis blockiert sind





# Bedingungsvariable

(engl. *condition variable*)

Ein mit einer Schlossvariablen verknüpfter abstrakter Datentyp auf dem zwei Operationen definiert sind [64]:

await (auch: *wait*) $\models$ Unterbrechungssprotokoll
- gibt den durch die Schlossvariable gesperrten kritischen Abschnitt automatisch frei
- blockiert den laufenden Prozess auf eine Bedingungsvariable
- bewirbt einen durch Ereignisanzeige deblockierten Prozess erneut um den Eintritt in den kritischen Abschnitt

cause (auch: *signal*) $\models$ Signalisierungsprotokoll
- zeigt das mit der Bedingungsvariable assoziierte Ereignis an
- deblockiert die ggf. auf das Ereignis wartenden Prozesse

Ermöglicht einem Prozess, innerhalb eines kritischen Abschnitts zu warten, ohne diesen während der Wartephase belegt zu halten.





# Datenpuffer ohne Pufferbegrenzung

Virtuell unendlich großer Puffer: Ringpuffer

```
typedef struct ringbuffer {
    char     data[NDATA];
    unsigned nput;
    unsigned nget;
} ringbuffer;

void rb_reset (ringbuffer *bufp) {
    bufp->nput = bufp->nget = 0;
}

char rb_fetch (ringbuffer *bufp) {
    return bufp->data[bufp->nget++ % NDATA];
}

void rb_store (ringbuffer *bufp, char item) {
    bufp->data[bufp->nput++ % NDATA] = item;
}
```

Problemstellen:

Füllstand log. Ablauf
- voll?
- leer?

füllen Zählen
- `nput++`

leeren Zählen
- `nget++`

☞ Synchronisation

# Datenpuffer mit Pufferbegrenzung
(engl. *bounded buffer*)

Datenpuffer begrenzter Speicherkapazität als Ringpufferspezialisierung:

```
typedef struct buffer {
    ringbuffer    ring;
    unsigned char free;
    bool          lock;
} buffer;

void bb_reset (buffer *bufp) {
    rb_reset(&bufp->ring);
    bufp->free = NDATA;
    bufp->lock = 0;
}
```

`free` Bedingungsvariable
- Füllstandkontrolle
  - voll `free` $= 0$
  - leer `free` $= NDATA$
  - frei $0 <$ `free` $\leq NDATA$
- Puffer ist initial leer

`lock` Schlossvariable
- Absicherung
- KA ist initial offen





# Datenpuffer mit Pufferbegrenzung (Forts.)
Koordiniertes Leeren

```
char bb_fetch (buffer *bufp) {
    char item;
    acquire(&bufp->lock);
    while (bufp->free == NDATA)
        await(&bufp->ring, &bufp->lock);
    item = rb_fetch(&bufp->ring);
    bufp->free++;
    cause(&bufp->free);
    release(&bufp->lock);
    return item;
}
```

Puffer leeren ist ein KA:
- darf sich weder selbst noch mit dem Füllen überlappen
- gegenseitiger Ausschluss

Wartebedingung:
- Puffer ist leer

Entnahme eines Datums gibt ein wiederverwendbares Betriebsmittel frei
- die Anzahl der freien Puffereinträge erhöht sich um 1
- die Wartebedingung zum Füllen kann signalisiert werden

☞ das Datum selbst ist ein konsumierbares Betriebsmittel





# Datenpuffer mit Pufferbegrenzung (Forts.)
Koordiniertes Füllen

```
void bb_store (buffer *bufp, char item) {
    acquire(&bufp->lock);
    while (!bufp->free)
        await(&bufp->free, &bufp->lock);
    rb_store(&bufp->ring, item);
    bufp->free--;
    cause(&bufp->ring);
    release(&bufp->lock);
}
```

Puffer füllen ist ein KA:
- darf sich weder selbst noch mit dem Leeren überlappen
- gegenseitiger Ausschluss

Wartebedingung:
- Puffer ist voll

Pufferung des Datums macht ein konsumierbares Betriebsmittel verfügbar
- die Anzahl der freien Puffereinträge erniedrigt sich um 1
- die Wartebedingung zum Leeren kann signalisiert werden





# Bedingter kritischer Abschnitt [65]
(engl. *conditional critical section*, resp. *region*)

Betreten des kritischen Abschnitts wird von einer Wartebedingung abhängig gemacht, die nicht erfüllt sein darf, um den Prozess fortzusetzen
- die Bedingung ist als Prädikat über die im kritischen Abschnitt enthaltenen bzw. verwendeten Daten definiert

Auswertung der Wartebedingung muss im kritischen Abschnitt erfolgen
- bei Nichterfüllung der Bedingung wird der Prozess auf Eintritt eines zur Wartebedingung korrespondierenden Ereignisses blockiert
  - damit das Ereignis später signalisiert werden kann, muss der kritische Abschnitt beim Schlafenlegen jedoch freigegeben werden
- bei (genauer: nach) Erfüllung/Signalisierung der Bedingung versucht der Prozess den kritischen Abschnitt wieder zu belegen
  - ggf. muss ein deblockierter Prozess die Bedingung neu auswerten

## Unterbrechungs- und Signalisierungsprotokoll

Ereigniserwartung und -anzeige

```
void await (void *beep, bool *lock) {
    abide(beep);
    release(lock);
    block();
    acquire(lock);
}

void cause (void *beep) {
    rouse(beep);
}
```

`abide()` gibt bekannt, dass der laufende Prozess die Regel befolgen wird, sich in endlicher Zeit auf das angegebene Ereignis zu blockieren

`block()` blockiert den Prozess

- trat das Ereignis ein und wurde der Prozess aufgeweckt, versucht er erneut den kritischen Abschnitt zu betreten

Die Paarung `abide()`/`block()` verhindert die mögliche „*race condition*“, wenn nach Freigabe des kritischen Abschnitts der Prozess verdrängt und vor seiner eigentlichen Blockierung die Wartebedingung signalisiert wird.





## Semaphor

Semaphor (engl. *semaphore*)

Eine „nicht-negative ganze Zahl“, für die zwei unteilbare Operationen definiert sind [54]:

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 zur Signalisierung von Ereignissen zwischen gleichzeitigen Prozessen (deren Ausführung sich zeitlich überschneidet).

☞ S. V-82





## Kompositer Datentyp

Zusammengesetzt aus Bedingungsvariable und Schlossvariable

```
typedef struct semaphore {
    unsigned int load;
    bool         lock;
} semaphore;

void initial (semaphore *sema, unsigned int load) {
    sema->load = load;
    sema->lock = 0;
}
```

Vorbelegung...

- definiert die Anzahl der vom Semaphor zu verwaltenden Betriebsmittel
- entriegelt das Schloss

`load` Bedingungsvariable

- implementiert das Protokoll zwischen P und V

`lock` Schlosssvariable

- sorgt für die Unteilbarkeit der Operationen: kritischer Abschnitt





## Kritische Abschnitte P und V

Verhungerungsgefahr — auch kurze nebenläufige Programme sind nicht einfach...

```
void prolaag (semaphore *sema) {
    acquire(&sema->lock);
    while (sema->load == 0)
        await(&sema->load, &sema->lock);
    sema->load--;
    release(&sema->lock);
}

void verhoog (semaphore *sema) {
    acquire(&sema->lock);
    if (sema->load++ == 0)
        cause(&sema->load);
    release(&sema->lock);
}
```

Wartebedingung

- Semaphorwert ist 0
- muss wiederholt ausgewertet werden

Signalisierung

- Semaphorwert war 0
- läuft ggf. ins Leere
  - Warteschlange würde dem vorbeugen
  - muss verträglich zur Einplanung sein

Wiedereintritt nach erfolgter Signalisierung ist damit konfrontiert, dass andere Prozesse ggf. vorbeigezogen sind

## Instrumente zur Betriebsmittelvergabe

Differenziert nach dem Wertebereich eines Semaphors

binärer Semaphor (engl. *binary semaphore*)
- verwaltet zu einem Zeitpunkt immer nur genau ein Betriebsmittel
  - gegenseitiger Ausschluss (engl. *mutual exclusion*, *mutex*)
- vergibt unteilbare Betriebsmittel an Prozesse
- besitzt den Wertebereich $[0, 1]$

zählender Semaphor (engl. *counting semaphore*, *general semaphore*)
- verwaltet zu einem Zeitpunkt mehr als ein Betriebsmittel
  - d.h., mehrere Betriebsmittelinstanzen desselben Typs
- vergibt konsumier- bzw. teilbare Betriebsmittel an Prozesse
- besitzt den Wertebereich $[0, N]$, für $N$ Betriebsmittel





## Arten von Betriebsmitteln

Semaphore und Betriebsmittelverwaltung

wiederverwendbare Betriebsmittel werden angefordert und freigegeben
- ihre Anzahl ist begrenzt: Prozessoren, Geräte, Speicher (z.B. Puffer)

  teilbar zu einer Zeit von mehreren Prozessen belegbar

  unteilbar zu einer Zeit von einem Prozess belegbar
- auch ein kritischer Abschnitt ist solch ein Betriebsmittel
  - von jedem Typ gibt es jedoch nur eine einzige Instanz

konsumierbare Betriebsmittel werden erzeugt und zerstört
- ihre Anzahl ist (log.) unbegrenzt: Signale, Nachrichten, Interrupts

  Produzent kann beliebig viele davon erzeugen

  Konsument zerstört sie wieder bei Inanspruchnahme
- Produzent und Konsument sind voneinander abhängig (S. V-89)





## Ausschließender Semaphor

Vergabe unteilbarer Betriebsmittel

```
semaphore mutex = {1, 0};

void chain (chainlink **next, chainlink *item) {
    prolaag(&mutex);
    *next = (*next)->link = item;
    verhoog(&mutex);
}
```

Beispiel von S. V-83:

`P()` `prolaag(&mutex)`

`V()` `verhoog(&mutex)`

unteilbares Betriebsmittel von dem es nur eine Instanz gibt
- der Initialwert des Semaphors ist 1

mehrseitige Synchronisation in welcher Reihenfolge die nebenläufigen Prozesse den kritischen Bereich betreten werden, ist unbestimmt
- gleichzeitig können jedoch nicht mehrere Prozesse drin sein





## Signalisierender Semaphor

Vergabe konsumierbarer Betriebsmittel

```
semaphore mite = {0, 0};
char      data;

char consumer () {
    prolaag(&mite);
    return data;
}

void producer (char item) {
    data = item;
    verhoog(&mite);
}
```

konsumierbares Betriebsmittel muss vor dem Verbrauch erst erzeugt werden
- der Initialwert des Semaphors ist 0

einseitige Synchronisation nur einer von beiden Prozessen wird ggf. blockieren
- der Konsument, wenn noch kein Datum verfügbar ist

Der Datenpuffer ist begrenzt, jedoch wird die Pufferbegrenzung ignoriert:
- Daten gehen verloren, wenn die Prozesse nicht im gleichen Takt arbeiten: $Konsument* \rightarrow (Produzent \rightarrow Konsument)+$

# Datenpuffer mit Pufferbegrenzung

*Bounded buffer revisited...*

Ringpufferspezialisierung: „Dreiergespann“ von Semaphore...

```
typedef struct buffer {
    ringbuffer ring;
    semaphore  lock;
    semaphore  free;
    semaphore  full;
} buffer;

void bb_reset (buffer *bufp) {
    rb_reset(&bufp->ring);
    initial(&bufp->lock, 1);
    initial(&bufp->free, NDATA);
    initial(&bufp->full, 0);
}
```

`lock` sichert die Pufferoperationen
- gegenseitiger Ausschluss von lesen/schreiben

`free` verhindert Pufferüberlauf
- stoppt den Schreiber beim vollen Puffer

`full` verhindert Pufferunterlauf
- stoppt den Leser beim leeren Puffer





# Datenpuffer mit Pufferbegrenzung (Forts.)

Koordiniertes Leeren

```
char bb_fetch (buffer *bufp) {
    char item;
    prolaag(&bufp->full);
    prolaag(&bufp->lock);
    item = rb_fetch(&bufp->ring);
    verhoog(&bufp->lock);
    verhoog(&bufp->free);
    return item;
}
```

Szenario beim Leeren:
- einem leeren Puffer kann nichts entnommen werden
- freigewordener Pufferplatz soll wiederverwendbar sein
- den Puffer zu leeren, ist ein kritischer Abschnitt

einseitige Synchronisation $\mapsto$ zwei signalisierende Semaphore
- durch `full` ein konsumierbares Betriebsmittel anfordern
- durch `free` ein wiederverwendbares Betriebsmittel bereitstellen

mehrseitige Synchronisation $\mapsto$ ausschließender Semaphor `lock`
- sich selbst überlappendes Leeren und Leeren überlappendes Füllen





# Datenpuffer mit Pufferbegrenzung (Forts.)

Koordiniertes Füllen

```
void bb_store (buffer *bufp, char item) {
    prolaag(&bufp->free);
    prolaag(&bufp->lock);
    rb_store(&bufp->ring, item);
    verhoog(&bufp->lock);
    verhoog(&bufp->full);
}
```

Szenario beim Füllen:
- voll ist voll...
- gepufferte Daten sollen konsumierbar sein
- Puffer füllen ist kritisch

einseitige Synchronisation $\mapsto$ zwei signalisierende Semaphore
- durch `free` ein wiederverwendbares Betriebsmittel anfordern
- durch `full` ein konsumierbares Betriebsmittel bereitstellen

mehrseitige Synchronisation $\mapsto$ ausschließender Semaphor `lock`
- sich selbst überlappendes Füllen und Füllen überlappendes Leeren





# Semaphore „*considered harmful*“

Nicht alles „Gold“ glänzt...

- auf Semaphore basierende Lösungen sind komplex und fehleranfällig
  - Synchronisation ist Querschnittsbelang nicht-sequentieller Programme
    - kritische Abschnitte neigen dazu, mit ihren P/V-Operationen quer über die Software verstreut vorzuliegen
  - das Schützen gemeinsamer Variablen bzw. Freigeben kritischer Abschnitte kann dabei leicht vergessen werden
- hohe Gefahr der Verklemmung (engl. *deadlock*) von Prozessen
  - umso zwingender ist die Notwendigkeit von Verfahren zur Vorbeugung, Vermeidung und/oder Erkennung solcher Verklemmungen
  - nicht-blockierende Synchronisation ist mit diesem Problem nicht behaftet, dafür jedoch nicht immer durchgängig praktizierbar
- „linguistische Unterstützung“ reduziert Fehlermöglichkeiten gravierend

## Monitor
(engl. *monitor*)

Ein synchronisierter abstrakter Datentyp, d.h., ein ADT mit impliziten Synchronisationseigenschaften [66, 67]:

mehrseitige Synchronisation an der Monitorschnittstelle (Semaphor)
- gegenseitiger Ausschluss der Ausführung exportierter Prozeduren

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 Ansatz: Concurrent Pascal, PL/I, Mesa, . . . , Java.





## Monitor ≡ Modul
Monitor vs. Semaphor

Kapselung (engl. *encapsulation*)
- von mehreren Prozessen gemeinsam bearbeitete Daten müssen in Monitoren organisiert vorliegen
- als Konsequenz muss die Programmstruktur kritische Abschnitte explizit sichtbar machen
  - inkl. die zulässigen (an zentraler Stelle definierten) Zugriffsfunktionen

Datenabstraktion (engl. *information hiding*)
- wie ein Modul, so kapselt auch ein Monitor für mehrere Funktionen Wissen über gemeinsame Daten
- Auswirkungen lokaler Programmänderungen bleiben (eng) begrenzt

☞ ein Monitor ist Konzept der Ebene 5, ein Semaphor das der Ebene 3





## Modulkonzept erweitert um Synchronisationssemantik
Monitor ≡ implizit synchronisierte Klasse

Monitorprozeduren (engl. *monitor procedures*)
- schließen sich bei konkurrierenden Zugriffen durch mehrere Prozesse in ihrer Ausführung gegenseitig aus
  - der erfolgreiche Prozeduraufruf sperrt den Monitor
  - bei Prozedurrückkehr wird der Monitor wieder entsperrt
- repräsentieren per Definition kritische Abschnitte, deren Integrität vom Kompilierer garantiert wird
  - die „Klammerung“ kritischer Abschnitte erfolgt automatisch
  - der Kompilierer setzt die dafür notwendigen Steueranweisungen ab

Synchronisationsanweisungen (Semaphore, Schloss-/Bedingungsvariablen)
- sind Querschnittsbelang eines Monitors und nicht des gesamten nicht-sequentiellen Programms
  - sie liegen nicht quer über die Software verstreut vor





## Signalisierung einer Wartebedingung erwarten
*wait*

Monitorfreigabe als notwendiger Seiteneffekt beim Warten:
- andere Prozesse wären sonst an den Monitoreintritt gehindert
- als Konsequenz könnte die zu erfüllende Bedingung nie erfüllt werden
- schlafenlegende Prozesse würden nie mehr erwachen ⇝ Verklemmung

Monitordaten sind in einem konsistenten Zustand zu hinterlassen
- andere Prozesse aktivieren den Monitor während der Blockadephase
- als Folge sind (je nach Funktion) Zustandsänderungen zu erwarten
- vor Eintritt in die Wartephase muss der Datenzustand konsistent sein

☞ aktives Warten im Monitor ist logisch komplex und leistungsmindernd

# Signalisierung einer Wartebedingung

*signal*

Prozessblockaden in Bezug auf eine Wartebedingung werden aufgehoben

- im Falle wartender Prozesse sind als Anforderungen zwingend:
  - wenigstens ein Prozess deblockiert an der Bedingungsvariablen
  - höchstens ein Prozess rechnet nach der Operation im Monitor weiter
- erwartet kein Prozess ein Signal, ist die Operation wirkungslos
  - d.h., Signale dürfen in Bedingungsvariablen nicht gespeichert werden

Lösungsansätze hierzu sind z.T. von sehr unterschiedlicher Semantik

- das betrifft etwa die Anzahl der befreiten Prozesse:
  - alle auf die Bedingung wartenden oder genau nur einer
- große Unterschiede liegen auch in Besitzwechsel bzw. Besitzwahrung
  - „falsche Signalisierungen“ werden toleriert oder nicht





# Signalisierungssemantiken

Besitzwahrung

genau einen wartenden Prozess befreien . . . nur welchen?

- bei mehr als einen wartenden Prozess ist eine Auswahl zu treffen
- die Auswahlentscheidung muss zur Fadeneinplanung korrespondieren
- ggf. ist bereits bei Blockierung möglichen Konflikten vorzubeugen

alle wartenden Prozesse befreien ↦ Hansen [64]

- die Auswahlentscheidung unterliegt allein dem Planer
- Fadeneinplanung entgegenwirkende Konflikte werden ausgeschlossen
- verschiedene Belange sind sauber voneinander getrennt

- in beiden Fällen erfolgt die Neuauswertung der Wartebedingung
  - dadurch werden jedoch auch falsche Signalisierungen toleriert
- signalisierte Prozesse bewerben sich erneut um den Monitorzutritt





# Signalisierungssemantiken (Forts.)

Besitzwechsel

Wechsel vom signalisierenden zum signalisierten Prozess ↦ Hoare [67]

- nur einer von ggf. mehreren wartenden Prozessen wird signalisiert
  - der signalisierte Prozess setzte seine Berechnung im Monitor fort
  - als Folge muss der signalisierende Prozess den Monitor abgeben
- Fortführungsbedingung des signalisierten Prozesses ist garantiert
  - seit Signalisierung konnte kein anderer Prozess den Monitor betreten
  - kein anderer Prozess konnte die Fortführungsbedingung entkräften

- es erfolgt keine Neuauswertung der Wartebedingung
  - als Konsequenz werden falsche Signalisierungen nicht toleriert
- eine erhöhte Anzahl von Fadenwechseln ist in Kauf zu nehmen
- der signalisierende Prozess bewirbt sich erneut um den Monitorzutritt





# Datenmonitor mit Pufferbegrenzung

*Bounded buffer revisited. . .* „Concurrent C++“

```
class Ringbuffer {
    char     data[NDATA];
    unsigned nput, nget;
public:
    Ringbuffer ()     { nput = nget = 0; }
    char fetch ()     { return data[nget++ % NDATA]; }
    void store (char) { data[nput++ % NDATA] = item; }
};

monitor Buffer : private Ringbuffer {
    unsigned  free;
    condition null, full;
public:
    Buffer ()         { free = NDATA; }
    char fetch ();
    void store (char);
};
```

## Datenmonitor mit Pufferbegrenzung (Forts.)

Koordiniertes Leeren

```
char Buffer::fetch () {
    char item;
    while (free == NDATA) full.await();
    item = Ringbuffer::fetch();
    free++;
    null.signal();
    return item;
}
```

Bedingungsvariablen:

`full` erwartet einen Eintrag

`null` signalisiert freien Platz

Instanzenvariable:

`free` aktueller „Pegelstand“

Hansen'scher Monitor Wartebedingung ist wiederholt zu überprüfen

- bewirbt signalisierte Prozesse erneut um den Monitorzutritt
  - die Phase ab der Signalisierung von `full` bis zum Wiedereintritt in den Monitor des signalisierten (leerenden) Prozesses ist teilbar
  - der Puffer könnte zwischenzeitig geleert worden sein $\rightsquigarrow$ blockieren
- toleriert (fehlerbedingte) falsche Signalisierungen von `full`





## Datenmonitor mit Pufferbegrenzung (Forts.)

Koordiniertes Füllen

```
void Buffer::store (char item) {
    while (!free) null.await();
    Ringbuffer::store(item);
    free--;
    full.signal();
}
```

Bedingungsvariablen:

`null` erwartet freien Platz

`full` signalisiert einen Eintrag

Instanzenvariable: `free` führt Buch über den aktuellen „Pegelstand“

Hansen'scher Monitor Wartebedingung ist wiederholt zu überprüfen

- bewirbt signalisierte Prozesse erneut um den Monitorzutritt
  - die Phase ab der Signalisierung von `null` bis zum Wiedereintritt in den Monitor des signalisierten (füllenden) Prozesses ist teilbar
  - der Puffer könnte zwischenzeitig gefüllt worden sein $\rightsquigarrow$ blockieren
- toleriert (fehlerbedingte) falsche Signalisierungen von `null`





## Monitorkonzepte im Vergleich

Hansen vs. Hoare

**Hansen'scher Monitor**

```
while (free == NDATA) full.await();
while (!free) null.await();
```

Prozessen wird nicht garantiert, dass nach ihrer Signalisierung die Fortführungsbedingung gilt

- andere Prozesse können den Monitor betreten haben
- Wartebedingung erneut prüfen
- evtl. falsche Signalisierungen werden toleriert

**Hoare'scher Monitor**

```
if (free == NDATA) full.await();
if (!free) null.await();
```

Prozessen wird garantiert, dass nach ihrer Signalisierung die Fortführungsbedingung gilt

- kein anderer Prozess konnte den Monitor betreten haben
- Wartebedingung einmal prüfen
- evtl. falsche Signalisierungen werden nicht toleriert





## Blockierende Synchronisation „*considered harmful*“

Probleme von Schlossvariablen, Semaphore und Monitore

Leistung (engl. *performance*) insbesondere in SMP-Systemen [63]

- „*spin locking*“ reduziert ggf. massiv Busbandbreite

Robustheit (engl. *robustness*) „*single point of failure*“

- ein im kritischen Abschnitt scheiternder Prozess kann schlimmstenfalls das ganze System lahm legen

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

- un- bzw. weniger wichtige Prozesse können wichtige Prozesse „ausbremsen“ bzw. scheitern lassen
- Prioritätsverletzung, Prioritätsumkehr [68]
  - Mars Pathfinder [69]

Verklemmung (engl. *deadlock*) einiger oder sogar aller Prozesse

## Dualität von Koordinierungstechniken

Theorie vs. Praxis

| Problem | Methode |
|---|---|
| gegenseitiger Ausschluss | Schlossvariable, nicht blockierender Algor. |
| explizite Prozesssteuerung | Bedingungsvariable |
| 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 ineffizienten Lösungen führen





## Synchronisation

Koordination von Kooperation und Konkurrenz

- die Verfahren sind problemspezifisch und teils radikal unterschiedlich
  - einseitig oder mehrseitig
  - blockierend oder nicht-blockierend (wartend oder nicht-wartend)
- blockierende Verfahren erlauben die Wiederverwendung sequentieller Programme für nicht-sequentielle Ausführungsumgebungen
  - Schlossvariable, Bedingungsvariable, Semaphor, Monitor
  - die Gefahr von Verklemmungen ist stellenweise sehr hoch
- nicht-blockierende Verfahren sind frei von Verklemmungen, jedoch nicht unbedingt frei von Verhungerung
  - die Ansätze profitieren von Spezialbefehlen der CPU:
    CISC `cas`, `cas2` (`dcas`), `cmpxchg`
    RISC `ll/sc`
  - nicht-wartende Varianten beugen dem Verhungern von Prozessen vor
- nicht-sequentielle Programmierung ist nicht nur ein Betriebssystemfall





## Überblick

Synchronisation
- Verfahrensweisen
- Schlossvariable
- Bedingungsvariable
- Semaphor
- Monitor
- Zusammenfassung

Verklemmung
- Grundlagen
- Vorbeugung
- Vermeidung
- Erkennung und Erholung
- Zusammenfassung





## Stillstand von Prozessen

Verklemmung mit passivem Warten durch Blockade

dead·lock **1** a standstill resulting from the action of equal and opposed forces; stalemate **2** a tie between opponents in the course of a contest **3** DEADBOLT — to bring or come to a deadlock

Der Begriff bezeichnet (in der Informatik)

> *[. . . ] einen Zustand, in dem die beteiligten Prozesse wechselseitig auf den Eintritt von Bedingungen warten, die nur durch andere Prozesse in dieser Gruppe selbst hergestellt werden können. [2]*

- das „geringere Übel“ (im Vergleich zum *lifelock*), da dieser Zustand eindeutig erkennbar ist und so die Basis zur „Erholung“ gegeben ist
  - die verklemmten Prozesse sind im Einplanungszustand „blockiert“

# Stillstand von Prozessen (Forts.)

Verklemmung mit aktivem Warten

life·lock ist ...

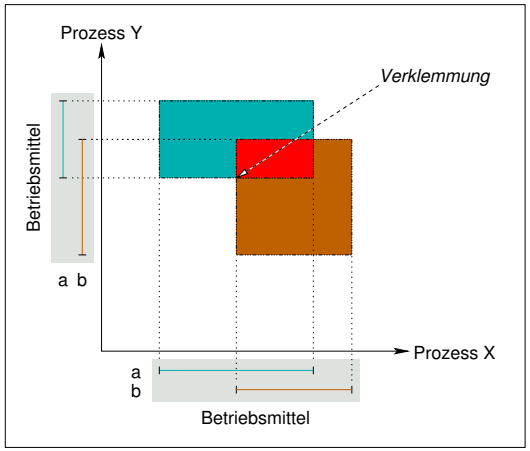
- ein *deadlock*-ähnlicher Zustand, in dem die involvierten Prozesse zwar nicht blockieren, sie aber auch keine wirklichen Fortschritte in der weiteren Programmausführung erreichen
- wenn die an der Verklemmung beteiligten Prozesse wechselseitig aktiv auf die Bereitstellung von Betriebsmitteln warten:
  - ohne Prozessorabgabe $\mapsto$ *busy waiting*
  - mit Prozessorabgabe, in Laufbereitschaft bleibend $\mapsto$ *lazy waiting*
- das „größere Übel“, da dieser Zustand nicht eindeutig erkennbar ist und damit die Basis zur „Erholung“ fehlt
  - die verklemmten Prozesse haben die Einplanungszustände „laufend“ oder „bereit“ (d.h., jeden anderen außer „blockiert“)
  - die Unterscheidung von unverklemmten Prozessen ist kaum möglich





# Entstehung von Verklemmungen

Überlappender Zugriff auf gemeinsame unteilbare Betriebsmittel




Alles hängt davon ab,

1. ob sich die Prozesse überhaupt einander überlappen und
2. wie sich die Überlappung dann in Bezug auf die gemeinsamen Betriebsmittel zeigt.

Keine Verklemmung, wenn...

$P_X$ $B_a$ erst nach $P_Y$ oder

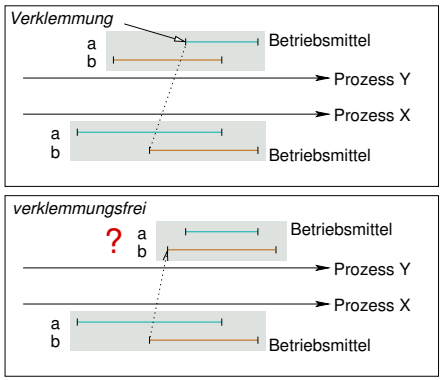
$P_Y$ $B_b$ erst nach $P_X$ belegt.





# Entstehung von Verklemmungen (Forts.)

Nicht-deterministische Prozessabläufe und Betriebsmittelanforderungen




Vorbeugung oder Vermeidung von Verklemmungen ist durch geschickte Prozesseinplanung möglich, vorausgesetzt, die Prozesse wie auch ihre Betriebsmittelanforderungen sind alle bekannt.

☞ kooperative Ausführung von $P_X$ und $P_Y \rightsquigarrow$ Verklemmungsfreiheit





# Voraussetzungen für Verklemmungen

Notwendige und hinreichende Bedingungen

notwendige Bedingungen (müssen erfüllt sein, damit die Aussage zutreffen kann)

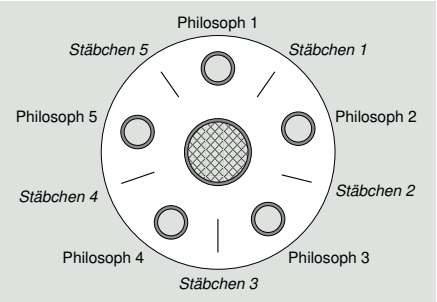
1. exklusive Belegung von Betriebsmitteln („*mutual exclusion*“)
   - die umstrittenen Betriebsmittel sind nur unteilbar nutzbar
2. Nachforderung von Betriebsmitteln („*hold and wait*“)
   - die umstrittenen Betriebsmittel sind nur schrittweise belegbar
3. kein Entzug von Betriebsmitteln („*no preemption*“)
   - die umstrittenen Betriebsmittel sind nicht rückforderbar

hinreichende Bedingung (muss erfüllt sein, damit die Aussage zutrifft bzw. „bewiesen“ ist)

4. zirkulares Warten (engl. *circular wait*)
   - Existenz einer geschlossenen Kette wechselseitig wartender Prozesse

# Speisende Philosophen

Szenario




Fünf Philosophen, die nichts anderes zu tun haben, als zu denken und zu essen, sitzen an einem runden Tisch. Denken macht hungrig — also wird jeder Philosoph auch essen. Dazu benötigt ein Philosoph jedoch stets beide neben seinem Teller liegenden Stäbchen.

Prozess $\mapsto$ Philosoph

Betriebsmittel $\mapsto$ Stäbchen

- unteilbar

Verklemmung jeder Philosoph nimmt „gleichzeitig“ das linke Stäbchen auf und greift anschließend auf das rechte zu...





# Speisende Philosophen (Forts.)

Synchronisationsaspekte

mehrseitige Synchronisation gegenseitiger Ausschluss beim Gebrauch wiederverwendbarer Betriebsmittel

- keine zwei benachbarten Philosophen können gleichzeitig dasselbe Stäbchen gemeinsam benutzen

einseitige Synchronisation

- ein Philosoph muss warten, bis seine beiden Nachbarn ihm ein Stäbchen zur Verfügung gestellt haben

Randbedingungen

- jeder Philosoph fordert die Stäbchen nacheinander an
  - kein Philosoph legt ein Stäbchen zurück, wenn er feststellt, dass das andere bereits vom Nachbarn aufgenommen worden ist
- ein Philosoph kann seinem Nachbarn ein bereits aufgenommenes Stäbchen nicht entreissen





# Speisende Philosophen (Forts.)

Umsetzung als nebenläufiges Programm

```
void phil (int who) {
    for (;;) {
        think();
        grab(who);
        eat();
        drop(who);
    }
}

void think () {}
void eat   () {}
```

```
semaphore rod[5] = {
    {1, 0}, {1, 0}, {1, 0}, {1, 0}, {1, 0}
};

void grab (int who) {
    P(&rod[who]);
    P(&rod[(who + 1) % NPHIL]);
}

void drop (int who) {
    V(&rod[who]);
    V(&rod[(who + 1) % NPHIL]);
}
```

`P()` fordert zu einem Zeitpunkt nur ein Stäbchen (engl. *rod*) an

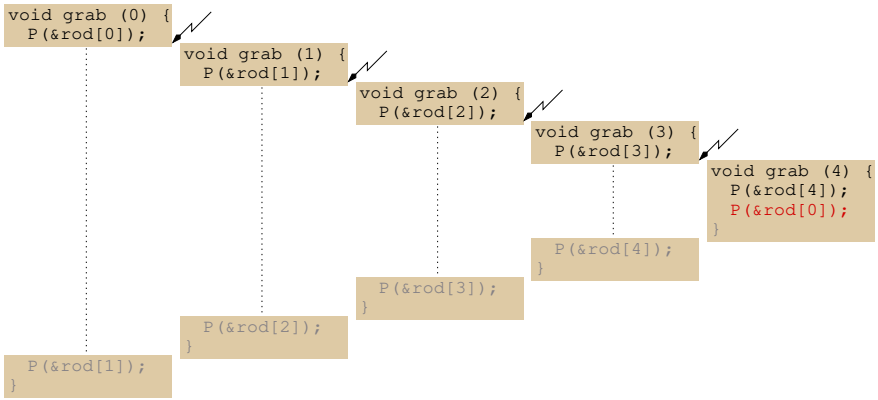
`V()` gibt ein Stäbchen frei





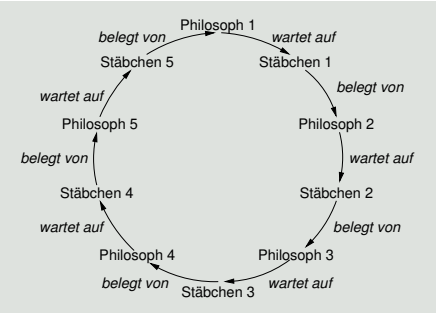
# Speisende Philosophen (Forts.)

Philosoph 4 überlappt Philosoph 3 und nimmt Stäbchen 4 auf... GAU



# Speisende Philosophen (Forts.)

## Zirkulares Warten




Betriebsmittelgraph zeigt für jedes Betriebsmittel, welcher Prozess es belegt

Wartegraph verbucht für jeden einzelnen Prozess, auf welches Betriebsmittel er wartet

- ein geschlossener Kreis (im Wartegraphen) erfasst all die Prozesse, die sich zusammen im *Deadlock* befinden.
- es muss sichergestellt sein, dass ein solcher Kreis entweder nicht entstehen oder dass er erkannt und „durchbrochen“ werden kann





# Speisende Philosophen (Forts.)

## Kritischer Abschnitt — Verklemmungsfreiheit bei minimaler Nebenläufigkeit

```
semaphore mutex = {1, 0};

void grab (int who) {
    P(&mutex);
    P(&rod[who]);
    P(&rod[(who + 1) % NPHIL]);
    V(&mutex);
}
```

Philosoph$_{who}$ greift auf die von ihm zum Essen benötigten beiden Stäbchen unteilbar zu

- ein binärer Semaphor (`mutex`) sorgt für gegenseitigen Ausschluss

Problem Philosoph$_{who}$ wird von seinen Nachbarn reihum unterbrochen

- Philosoph$_{who+1}$ muss auf Stäbchen$_{who}$ warten
  - er blockiert im kritischen Abschnitt, ohne diesen freizugeben
- Philosoph$_{who+2}$ muss auf die Freigabe des KA warten
  - ebenso ergeht es den beiden anderen Philosophen
- schlimmstenfalls kann immer nur ein Philosoph essen





# Speisende Philosophen (Forts.)

## Ein in der Praxis nicht selten anzufindendes Problem...

Überall dort, wo eine von mehreren Prozessen benutzte Routine (mindestens) zwei wiederverwendbare Betriebsmittel zugleich benötigt, diese aber nur nacheinander anfordert werden können:

- auf mehreren Magnetbändern vorliegende Daten sortieren
- einen kontinuierlichen Strom kodierter Informationen umkodieren
- Nachrichten von einem Eingangsport auf einen Ausgangsport leiten
- Pakete auf einem Ringnetz zur übernächsten Station durchschleusen
- Daten von dem einen *Backup Medium* auf ein anderes transferieren





# Verklemmungsvorbeugung

## (engl. *deadlock prevention*)

indirekte Methoden entkräften eine der Bedingungen 1–3

1. nicht-blockierende Verfahren verwenden
2. Betriebsmittelanforderungen unteilbar (atomar) auslegen
3. Betriebsmittelentzug durch Virtualisierung ermöglichen
   - virtueller Speicher, virtuelle Geräte, virtuelle Prozessoren

direkte Methoden entkräften Bedingung 4

4. lineare/totale Ordnung von Betriebsmittelklassen einführen:
   - Betriebsmittel $B_i$ ist nur dann erfolgreich vor $B_j$ belegbar, wenn $i$ linear vor $j$ angeordnet ist (d.h. $i < j$).

- Regeln, die das Eintreten von Verklemmungen verhindern
- Methoden, die zur Entwurfs- bzw. Implementierungszeit greifen

# Verklemmungsvermeidung
(engl. *deadlock avoidance*)

Verhinderung von Wartezyklen durch strategische Maßnahmen:

- keine der drei notwendigen Bedingungen wird entkräftet
- fortlaufende Bedarfsanalyse schließt zirkulares Warten aus

Prozesse und ihre Betriebsmittelanforderungen sind zu steuern:

- das System wird (laufend) auf „unsichere Zustände“ hin überprüft
- Zuteilungsablehnung im Falle nicht abgedeckten Betriebsmittelbedarfs
- anfordernde Prozesse nicht bedienen bzw. frühzeitig suspendieren
- Betriebsmittelnutzung einschränken $\leadsto$ „sichere Zustände“

☞ *à priori* Wissen über den maximalen Betriebsmittelbedarf ist erforderlich





# Sicherer/Unsicherer Zustand
Speisende Philosophen

Ausgangspunkt fünf Stäbchen sind insgesamt vorhanden

- jeder der fünf Philosophen braucht zwei Stäbchen zum Essen

Situation $P_1$, $P_2$ und $P_3$ haben je ein Stäbchen; zwei Stäbchen sind frei

- $P_4$ fordert ein Stäbchen an $\rightarrow$ ein Stäbchen wäre dann noch frei
  - sicherer Zustand: einer von drei Philosophen könnte essen
  - die Anforderung von $P_4$ wird akzeptiert
- $P_5$ fordert ein Stäbchen an $\rightarrow$ kein Stäbchen wäre dann mehr frei
  - unsicherer Zustand: keiner der Philosophen könnte essen
  - die Anforderung von $P_5$ wird abgelehnt, $P_5$ muss warten
- haben vier Philosophen je ein Stäbchen, wird der fünfte gestoppt





# Sicherer/Unsicherer Zustand (Forts.)
Leitungsvermittlung

Ausgangspunkt ein Vermittlungsrechner mit 12 Kommunikationskanälen

- Prozess $P_1$ benötigt max. 10 Kanäle, $P_2$ vier und $P_3$ neun

Situation: $P_1$ belegt fünf Kanäle, $P_2$ und $P_3$ je zwei; drei Kanäle sind frei

- $P_3$ fordert einen Kanal an, zwei blieben frei $\rightarrow$ unsicherer Zustand
  - $P_3$ könnte noch sechs Kanäle anfordern: $6 > 2$
  - die Anforderung von $P_3$ wird abgelehnt, $P_3$ muss warten
- $P_1$ fordert zwei Kanäle an, einer bliebe frei $\rightarrow$ unsicherer Zustand
  - $P_1$ könnte noch drei Kanäle anfordern: $3 > 1$
  - die Anforderung von $P_1$ wird abgelehnt, $P_1$ muss warten
- sichere Prozessfolge: $P_2 \rightarrow P_1 \rightarrow P_3$





# Verklemmungsfreiheit
Verhinderung unsicherer Zustände

sicherer Zustand ist, wenn eine Folge der Verarbeitung vorhandener Prozesse existiert, in der alle Betriebsmittelanforderungen erfüllbar sind

unsicherer Zustand ist, wenn eine solche Folge nicht existiert; Erkennung dieses Zustands z.B. durch:

- Betriebsmittelbelegungsgraph (engl. *resource allocation graph*)
  - damit Vorhersage über das Eintreten von Zyklen treffen $\leadsto O(n^2)$
  - bei jeder Betriebsmittelanforderung den Graphen überprüfen
- Bankiersalgorithmus (engl. *banker's algorithm* [54])
  1. Prozesse beenden ihre Operationen in endlicher Zeit
  2. Betriebsmittelbedarf aller Prozesse übersteigt nicht den Gesamtvorrat
  3. Prozesse definieren einen verbindlichen Kreditrahmen
  4. Betriebsmittelzuteilung erfolgt variabel innerhalb dieses Rahmens
- die Verfahrensweisen führen Prozesse dem *long-term scheduling* zu

## Verklemmungserkennung
(engl. *deadlock detection*)

Verklemmungen werden (stillschweigend) in Kauf genommen...

- nichts ist im System verhindert das Auftreten von Zyklen
- keine der vier Bedingungen wird entkräftet

Ansatz: Wartegraph erstellen und auf Zyklen hin untersuchen $\rightsquigarrow O(n^2)$

- zu häufige Überprüfung verschwendet Betriebsmittel/Rechenleistung
- zu seltene Überprüfung lässt Betriebsmittel brach liegen

Zyklensuche geschieht zumeist in großen Zeitabständen, wenn...

- Betriebsmittelanforderungen zu lange andauern
- die Auslastung der CPU trotz Prozesszunahme sinkt
- die CPU bereits über einen sehr langen Zeitraum untätig ist





## Verklemmungsauflösung
Erholungsphase nach der Erkennungsphase

Prozesse abbrechen und dadurch Betriebsmittel frei bekommen

- verklemmte Prozesse schrittweise abbrechen (gr. Aufwand)
  - mit dem „effektivsten Opfer“ (?) beginnen
- alle verklemmten Prozesse terminieren (gr. Schaden)

Betriebsmittel entziehen und mit dem „effektivsten Opfer“ (?) beginnen

- der betreffende Prozess ist zurückzufahren bzw. wieder aufzusetzen
  - Transaktionen, *checkpointing*/*recovery* (gr. Aufwand)
- ein Aushungern der zurückgefahrenen Prozesse ist zu vermeiden

Gratwanderung zwischen Schaden und Aufwand:

- Schäden sind unvermeidbar und die Frage ist, wie sie sich auswirken





## Nachlese ...

Verfahren zum Vermeiden/Erkennen sind eher praxisirrelevant

- sie sind kaum umzusetzen, zu aufwändig und damit nicht einsetzbar
- zudem macht die Vorherrschaft sequentieller Programmierung diese Verfahren wenig notwendig

Verklemmungsgefahr ist lösbar durch Virtualisierung von Betriebsmitteln

- Prozesse beanspruchen/belegen ausschließlich logische Betriebsmittel
- der Trick besteht darin, in kritischen Momenten den Prozessen (ohne ihr Wissen) physische Betriebsmittel entziehen zu können
- dadurch wird die Bedingung der Nichtentziehbarkeit entkräftet

☞ eher praxisrelevant/verbreitet sind die Vorbeugungsmaßnahmen





## Zusammenfassung

- Verklemmung bedeutet „*deadlock*“ oder „*lifelock*“
  - „[...] einen Zustand, in dem die beteiligten Prozesse wechselseitig auf den Eintritt von Bedingungen warten, die nur durch andere Prozesse in dieser Gruppe selbst hergestellt werden können“ [2]
  - dabei ist der *Lifelock* das größere Problem beider Verklemmungsarten
- für eine Verklemmung müssen vier Bedingungen gleichzeitig gelten
  - exklusive Belegung, Nachforderung, kein Entzug von Betriebsmitteln
  - zirkulares Warten der die Betriebsmittel beanspruchenden Prozesse
- Verklemmungsbekämpfung meint: Vorbeugen, Vermeiden, Erkennen
  - die Verfahren können im Mix zum Einsatz kommen