Koordination nebenläufiger Prozesse

Die Koordination der Kooperation und Konkurrenz zwischen Prozessen wird **Synchronisation** (*synchronization*) genannt.

- Eine Synchronisation bringt die Aktivitäten verschiedener nebenläufiger Prozesse in eine Reihenfolge.

- Durch sie erreicht man also prozeßübergreifend das, wofür innerhalb eines Prozesses die Sequentialität von Aktivitäten sorgt.

Quelle: Herrtwich/Hommel (1989), *Kooperation und Konkurrenz*, S. 26

Koordinierung ≡ „Reihenschaltung“

ko·or·di'nie·ren bei<u>ordnen</u>; in ein Gefüge einbauen; aufeinander abstimmen; nebeneinanderstellen; Termine ∼.

- als kritisch erachtete nebenläufige Aktivitäten *der Reihe nach* ausführen
  ☞ überlappendes Zählen (✗ Kap. 5)
  ☞ verdrängende Prozesseinplanung (✗ Kap. 6)
- „der Reihe nach“ bedeutet, gewisse Prozesse bewusst zeitlich zu verzögern
  – je nach Verfahren trifft es den $\left\{ \begin{array}{c} \text{überlappenden} \\ \text{überlappten} \end{array} \right\}$ Prozess
- die Verfahren arbeiten {,nicht-}wartend bzw. {,nicht-}blockierend

Arten der Synchronisation

- „Koordination der Kooperation und Konkurrenz zwischen Prozessen“ . . .

1. *derselben Inkarnation* ☞ <u>Intra</u>prozess-Synchronisation
   – nicht-blockierende Synchronisation ist zwingend
   – Beispiel: asynchrone Programmunterbrechung (*interrupt*)
2. *verschiedener Inkarnationen* ☞ <u>Inter</u>prozess-Synchronisation
   – {,nicht-}blockierende Synchronisation ist verwendbar

☞ Differenzierung: *ein- und mehrseitige Synchronisation* nebenläufiger Prozesse

Einseitige Synchronisation **Unilateral**

- die Verfahren wirken sich nur auf einen der beteiligten Prozesse aus:

  Bedingungssynchronisation bzw.
  – das Weiterarbeiten 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 an der Synchronisation unbeteiligt[47]

[47] Die Veränderung einer Bedingung, auf die ein Prozess wartet, ist z. B. von einem anderen Prozess herbeizuführen.

Mehrseitige Synchronisation — Multilateral

- die Verfahren wirken sich auf (ggf.) alle beteiligten Prozesse aus
  - welche Prozesse im weiteren Ablauf verzögert werden, ist i. A. unvorhersehbar
  - allgemein gilt: „wer zuerst kommt, mahlt zuerst“
- die betroffenen Aktivitäten stehen miteinander im **gegenseitigen Ausschluss**
  - erzwungen wird die *atomare Ausführung* von Anweisungsfolgen
  - „abschnittweise“ werden diese niemals nebenläufig/parallel durchgeführt
- die atomar ausgeführten Anweisungsfolgen bilden eine *Elementaroperation*

Asynchrone Programmunterbrechungen (✗ Kap. 5)

- je nach Verfahren erfährt die eine oder andere Seite eine Verzögerung:

  Verzögerung des unterbrechenden Prozesses
  - weiche/harte Synchronisation der „Hardware/Software-Interrupts“
  - logischer Ansatz („Schleusen“ [28]) bzw. Ebene $_2$-Befehle: `cli`, `sti` (x86)

  Verzögerung des unterbrochenen Prozesses
  - Ebene $_2$-Befehle:
    - ☞ `cas` (IBM 370, m68020+), `cmpxchg` (i486+) — CISC
    - ☞ `ll`/`sc` (DEC Alpha, MIPS, PowerPC) — RISC
  - nicht-blockierende Synchronisation nebenläufiger Aktivitäten
- die Verfahren synchronisieren *einseitig*, d. h., sie arbeiten *unilateral*

Verzögerung des unterbrechenden Prozesses

```c
int wheel = 0;

void __attribute__ ((interrupt)) tip () {
    wheel += 1;
}

int main () {
    for (;;)
        printf("%d\n", incr(&wheel));
}
```

```c
int incr (int *p) {
    int x;
    asm("cli");
    x = *p += 1;
    asm("sti");
    return x;
}
```

„blockierende Synchronisation“ Interrupts werden zeitweilig unterbunden

Verzögerung des unterbrochenen Prozesses

```c
int wheel = 0;

void __attribute__ ((interrupt)) tip () {
    incr(&wheel);
}

int main () {
    for (;;)
        printf("%d\n", incr(&wheel));
}
```

```c
int incr (int *p) {
    int x;
    do x = *p;
    while (!cas(p, x, x + 1));
    return x + 1;
}
```

nicht-blockierende Synchronisation Wiederholung der Berechnung findet statt, wenn nebenläufig eine andere Aktivität erfolgreich beendet wurde

Spezialbefehl

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

CAS (*compare and swap*)

- unteilbare Operation
  - *read-modify-write*
- Komplexbefehl ☞ CISC

Ist blockierungsfrei nur in Bezug auf Prozesse!

Multiprozessor-Synchronisation (gemeinsamer Speicher, *shared memory*)

`atomic()` verhindert (Speicher-) Buszugriffe durch andere Prozessoren
`cimota()` lässt (Speicher-) Buszugriffe anderer Prozessoren wieder zu

☞ Auf Interrupts wird, wie sonst auch, erst am Befehlsende reagiert!

Gegenseitiger Ausschluss — *mutual exclusion*

- ein Ansatz, der kennzeichnend ist für die *mehrseitige Synchronisation*:

  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** (*critical sections*, *critical regions*). *KA*

  Quelle: Herrtwich/Hommel (1989), *Kooperation und Konkurrenz*, S. 137

- die kritischen Abschnitte sind durch „Synchronisationsklammern" zu schützen

Kritischer Abschnitt

Beim Betreten (*enter*) und Verlassen (*leave*) gelten bestimmte Vorgehensweisen:

Eintrittsprotokoll (*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 (*exit protocol*)

- regelt die Freigabe des kritischen Abschnitts durch einen Prozess
- andere erhalten die Möglichkeit zum Betreten des kritischen Abschnitts

Die Vorgehensweisen variieren mit dem realisierten Synchronisationsverfahren.

Schlossvariable — *lock variable*

Ein *abstrakter Datentyp*, auf dem zwei Operationen definiert sind:

acquire (*lock*) ☞ Eintrittsprotokoll

- verzögert einen Prozess, bis das zugehörige Schloss offen ist
  - bei bereits geöffnetem Schloss fährt der Prozess unverzögert fort
- verschließt das Schloss („von innen"), wenn es offen ist

release (*unlock*) ☞ Austrittsprotokoll

- öffnet das zugehörige Schloss, ohne den öffnenden Prozess zu verzögern

Implementierungen werden als **Schlossalgorithmen** (*lock algorithms*) bezeichnet.

Schlossalgorithmus (1) — Prinzip mit Problem(en)

```
typedef char bool;

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

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

`acquire()` soll einen kritischen Abschnitt schützen, ist dabei aber selbst kritisch:

- Problem macht die Phase vom Verlassen der Kopfschleife (`while`) 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, durch `acquire()` zu schützenden kritischen Abschnitt überlappt ausführen

Schlossalgorithmus (2) — 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 bilden eine ununterbrechbare Anweisungsfolge
- die Schleife muss unterbrechbar sein, damit das Schloss aufgeschlossen werden kann

- asynchrone Programmunterbrechungen werden abgewendet, obwohl diese nie den durch `acquire()` geschützten kritischen Bereich betreten dürfen [Warum?]

Schlossalgorithmus (3) — 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 bilden eine überlappungsfreie Anweisungsfolge
- die Schleife muss überlappbar sein, damit das Schloss aufgeschlossen werden kann

- Verdrängung des Prozesses wird abgewendet, obwohl ggf. nur einer von vielen lauffähigen Prozessen das Schloss öffnen wird

Schlossalgorithmus (4) — Komplexbefehl

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

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

TAS (*test and set*)

- atomarer Lese-Modifikations-Schreibzyklus
  - *read-modify-write*
- geeignet für Ein- *und* Mehrprozessorsysteme

☞ bei der Befehlsausführung wird kein anderer $\left\{\begin{array}{l}\text{Befehl abgearbeitet}\\ \text{Speicherzugriff durchgeführt}\end{array}\right.$

Aktives Warten — *busy waiting*

- Unzulänglichkeit der Schlossalgorithmen: der aktiv wartende Prozess . . .
  - kann selbst keine Änderung der Bedingung herbeiführen, auf die er wartet
  - behindert daher 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.

- die dadurch entstehenden *Effizienzeinbußen* sind nur dann unproblematisch, wenn jedem Prozess ein eigener realer Prozessor zur Verfügung steht

Passives Warten

- Prozesse geben die Kontrolle über die CPU ab während sie Ereignisse erwarten
  - im Synchronisationsfall blockiert sich ein Prozess auf ein Ereignis
    ☞ ggf. wird der PD des Prozesses in eine Warteschlange eingereiht
  - tritt das Ereignis ein, wird ein darauf wartender Prozess deblockiert
- die *Wartephase* eines Prozesses ist als *Blockadephase* („E/A-Stoß") ausgelegt
  - ggf. wird der Ablaufplan für die Prozesse aktualisiert (*scheduling*)
  - ein anderer, lauffähiger Prozess wird plangemäß abgefertigt (*dispatching*)
  - ist kein Prozess mehr lauffähig, läuft die CPU „leer" (*idle phase*)
- mit Beginn der Blockadephase eines Prozesses endet auch sein CPU-Stoß

Schlossalgorithmus (5) — Blockadephase

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

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

```
void sleep (void *flag) {
    racer()->wait = flag;
    block();
}

void awake (void *flag) {
    unsigned next;
    for (next = 0; next < NTASK; next++)
        if (task[next].wait == flag) {
            task[next].wait = 0;
            ready(&task[next]);
        }
}
```

☞ ***race condition*!**

Schlossalgorithmus (6) — „Bedingungsvariable"

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

```
void sleep (void *flag) {
    racer()->wait = flag;
    admit();
    block();
    avert();
}
```

- Verdrängung des laufenden Prozesses innerhalb der Kopfschleife (`acquire()`) bis zum Setzen der Wartebedingung (`sleep()`) wird abgewendet
- einer möglichen Überlappung von `acquire()` mit `release()` und der ggf. anhaltenden Blockade eines sich schlafenlegenden Prozesses wird vorgebeugt

☞ warten innerhalb kritischer Abschnitte bei gleichzeitiger Abschnittsfreigabe

Bedingungsvariable — *condition variable*

Ein *abstrakter Datentyp*, der mit einer Schlossvariablen verknüpft ist und auf dem zwei Operationen [29] definiert sind:

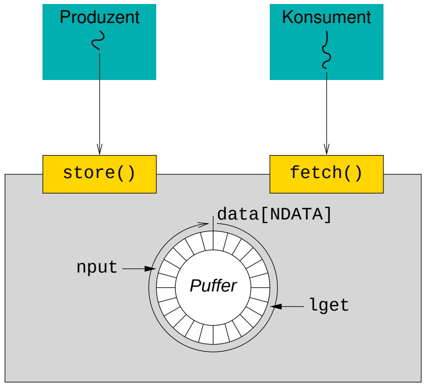
await (*wait*) lässt einem Prozess ein Ereignis (passiv) erwarten

- gibt den mit der Schlossvariablen gesperrten kritischen Abschnitt frei
- blockiert den laufenden Prozess auf die Bedingungsvariable
- bewirbt den deblockierten Prozess um Eintritt in den kritischen Abschnitt

cause (*signal*) zeigt ein Ereignis an und 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.

Fallstudie *Bounded Buffer*



Zwei Prozesse kommunizieren über einen Ringpuffer fester Größe (`data[NDATA]`):

Produzent speichert ein Datum in `data[nput]` und inkrementiert dann `nput` (*next put*)
☞ *Ausnahme*: Puffer voll (Überlauf)

Konsument inkrementiert `lget` (*last get*) und holt dann ein Datum aus `data[lget]`
☞ *Ausnahme*: Puffer leer (Unterlauf)

Bounded Buffer (1) Ringpuffer

```
struct buffer {
    Any data[NDATA];
    unsigned nput;
    unsigned lget;
};
```

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

- der Puffer begrenzt sich in zwei Dimensionen:

  physikalisch über `NDATA`, d. h., die maximale Anzahl der Puffereinträge
  logisch über den Abstand zwischen `nput` und `lget`, d. h., den Pufferfüllstand
  ☞ variiert mit dem jeweiligen Produzenten-/Konsumentenverhalten

- die Inkrementierung von `nput` bzw. `lget` erfolgt modulo `NDATA` (☞ „Ring")

Bounded Buffer (2) Kritische Abschnitte

```
void store (struct buffer *bufp, Any item) {
    bufp->data[bufp->nput] = item;
    bufp->nput = (bufp->nput + 1) % NDATA;
}

void fetch (struct buffer *bufp, Any *item) {
    bufp->lget = (bufp->lget + 1) % NDATA;
    *item = bufp->data[bufp->lget];
}
```

Nebenläufigkeit kann ...

store ungelesene Daten überschreiben lassen

fetch einmal geschriebene Daten mehrmals lesen lassen

☞ den möglichen *race conditions* ist vorzubeugen: gegenseitiger Ausschluss

Bounded Buffer (3) — Gegenseitiger Ausschluss

```
void store (struct buffer *bufp, Any item) {
    acquire(&bufp->lock);
    ⋮
    release(&bufp->lock);
}

void fetch (struct buffer *bufp, Any *item) {
    acquire(&bufp->lock);
    ⋮
    release(&bufp->lock);
}
```

Die Schlossvariable `lock` schützt den Puffer vor den nebenläufigen Zugriffen.

Ausnahmefälle beachten:

- Puffer voll?
- Puffer leer?

☞ Wartebedingungen

Bounded Buffer (4) — Wartebedingungen

```
void store (struct buffer *bufp, Any item) {
    acquire(&bufp->lock);
    while (bufp->nput == bufp->lget);
    ⋮
    release(&bufp->lock);
}

void fetch (struct buffer *bufp, Any *item) {
    acquire(&bufp->lock);
    while ((bufp->lget + 1) % NDATA == bufp->nput);
    ⋮
    release(&bufp->lock);
}
```

Warten innerhalb eines belegten (blockierten) kritischen Abschnitts ruft **Verklemmungen** hervor:

deadlock passiv warten
livelock aktiv warten

☞ log. Synchronisation

Bounded Buffer (5) — Logische Synchronisation

```
void store (struct buffer *bufp, Any item) {
    acquire(&bufp->lock);
    while (...) await(&bufp->free, &bufp->lock);
    ⋮
    cause(&bufp->full);
    release(&bufp->lock);
}

void fetch (struct buffer *bufp, Any *item) {
    acquire(&bufp->lock);
    while (...) await(&bufp->full, &bufp->lock);
    ⋮
    cause(&bufp->free);
    release(&bufp->lock);
}
```

Jeder Bedingung wird eine spezielle „Schlossvariable“ zugeordnet:

`free` *store*-Bedingung
`full` *fetch*-Bedingung

Die Belegung kritischer Abschnitte wird dadurch an Bedingungen geknüpft.

☞ *Bedingungsvariable*

[Warum weiterhin die Kopfschleifen?]

Bounded Buffer (6) — Ringpuffer *rev.*

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

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

- drei zusätzliche Pufferattribute koordinieren die Lese-/Schreiboperationen:

  Schlossvariable `lock` zum gegenseitigen Ausschluss
  Bedingungsvariablen `free` und `full` zur logischen Synchronisation

Logische Synchronisation — Grundoperationen [29]

```
void await (bool *flag, bool *lock) {
    label(flag);
    release(lock);
    block();
    acquire(lock);
}

void cause (bool *flag) {
    awake(flag);
}
```

```
void label (void *flag) {
    racer()->wait = flag;
}

void sleep (void *flag) {
    label(flag);
    :
}

void awake (void *flag) {
    :
}
```

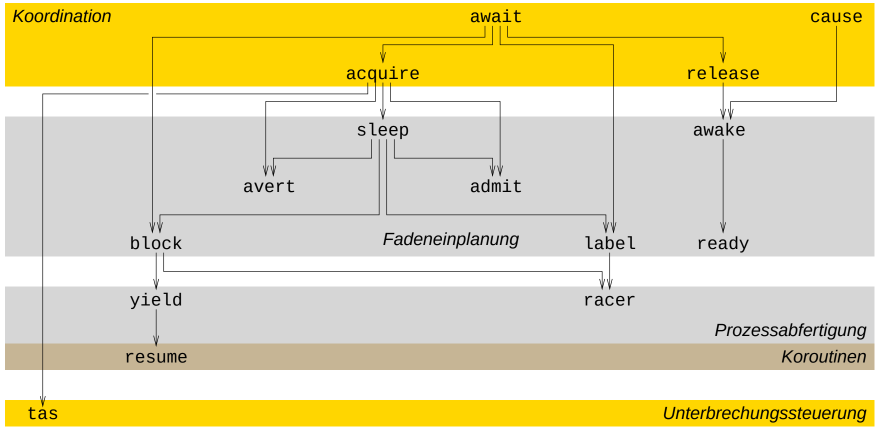
☞ *bedingter kritischer Abschnitt*

Bedingter kritischer Abschnitt

conditional critical section (resp. *region*, [30])

- ein durch (mind.) eine *Bedingungsvariable* kontrollierter kritischer Abschnitt
  - der Eintritt in den KA wird von einer Bedingung abhängig gemacht
  - die Bedingung ist als Prädikat über die im KA enthaltenen Daten definiert
- die Auswertung der Bedingung muss selbst im kritischen Abschnitt erfolgen
  - bei Nichterfüllung der Bedingung . . .
    - ☞ blockiert der Prozess auf eine zweite Schlossvariable und
    - ☞ gibt aber vorher die erste Schlossvariable frei
  - bei (genauer: nach) Erfüllung der Bedingung . . .
    - ☞ fordert der Prozess die erste Schlossvariable wieder an
- ggf. muss ein deblockierter Prozess die Bedingung neu auswerten

Logische Synchronisation — Funktionale Hierarchie



Semaphor — *semaphore*

Eine „*nicht-negative ganze Zahl*“, für die zwei Operationen definiert sind [31]:

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).

„Zeichenträger“, Signalmast

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. ([26], S. 201)

P/V Kritische Abschnitte

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

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

mit: `typedef unsigned int Semaphore;` [48]

[Warum die Kopfschleife im `P()`?]

[48] 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 [31], 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)

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

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 thread () {
    P(&mutex);
    ⋮
    V(&mutex);
}
```

```
void fibril () {
    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 (☞ *wiederverwendbare Betriebsmittel*)

☞ *unteilbares Betriebsmittel*: der Initialwert des Semaphors ist 1

Signalisierender Semaphor

```
void consumer () {
    ⋮
    P(&event);
    ⋮
}
```

```
void producer () {
    ⋮
    V(&event);
    ⋮
}
```

```
Semaphore event = 0;
```

einseitige Synchronisation einander zugeordnete Semaphoroperationen werden von verschiedenen Prozessen ausgeführt (☞ *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) Nachrichten

```
struct buffer {
    Any data[NDATA];
    unsigned nput;
    unsigned lget;
    Semaphore lock;
    Semaphore free;
    Semaphore 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->free);
    P(&bufp->lock);
    ⋮
    V(&bufp->lock);
    V(&bufp->full);
}

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

einseitig:

`free` Produzent wartet ggf. auf Konsumenten

`full` Konsument wartet ggf. auf Produzenten

mehrseitig:

`lock` Produzent/Konsument wartet ggf. auf andere Produzenten und/oder Konsumenten

[Warum `lock` nicht als „äußere Klammer"?]

Semaphore „*considered harmful*“

- auf Semaphore basierende Lösungen werden schnell komplex und fehleranfällig
  - kritische Abschnitte neigen dazu, mit ihren Synchronisationsanweisungen 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“ (☞ *Monitor*) beugt den möglichen Fehlern vor

Monitor

Ein *abstrakter Datentyp* mit impliziten Synchronisationseigenschaften [33, 34]:

mehrseitige Synchronisation an der Schnittstelle zum Monitor

- *gegenseitiger Ausschluss* der Ausführung aller Schnittstellenfunktionen

einseitige Synchronisation innerhalb des Monitors (*Bedingungsvariable*, ✗ 7-21)

wait blockiert einen Prozess auf das Eintreten eines Signals/einer Bedingung und gibt den Monitor implizit wieder frei [Warum die Monitorfreigabe?]

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 bzw. 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 Prozedurrückkehr wird der Monitor wieder entsperrt
  - ein Kompilierer setzt die dafür 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 <u>und</u>
    - ∗ 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 Besitzwahrung

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 (☞ Hansen [29])

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

☞ in beiden Fällen ist die Neuauswertung der Wartebedingung notwendig
☞ die signalisierten Prozesse bewerben sich erneut um den Monitorzutritt

Semantiken der Signalisierung Besitzwechsel

Wechsel vom signalisierenden zum signalisierten Prozess (☞ Hoare [34])

- 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

☞ die Neuauswertung der Wartebedingung entfällt
☞ eine erhöhte Anzahl von Fadenwechseln ist in Kauf zu nehmen
☞ der signalisierende Prozess bewirbt sich erneut um den Monitorzutritt

(pros) **Monitor vs. Semaphor** (cons)

- 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 für mehrere Funktionen Wissen über gemeinsame Daten
  - *information hiding*, d. h., Datenabstraktion wird unterstützt
  - Auswirkungen lokaler Programmänderungen bleiben (eng) begrenzt

☞ ein Monitor ist Konzept der Ebene $_5$, ein Semaphor Konzept der Ebene $_3$

Bounded Buffer (9) „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 () {
    nput = 0;
    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. [Warum gerade diese Werte?]

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 (10) {Hoare,Hansen}’scher 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 (Hoare, Hansen) wertet `when` die Bedingung ein- oder mehrmalig aus:

- syntaktisch ist kein Unterschied sichtbar,
- semantisch bleibt er jedoch bestehen.

Blockierende Synchronisation „*considered harmful*“

Leistung (*performance*) insb. in SMP-Systemen ist teils stark beeinträchtigt [35]

- „*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 ✗ Mars Pathfinder [36]

Verklemmung (*deadlock*) einiger oder sogar allller Prozesse

Prioritätsumkehr — *priority inversion*

Problem: mindestens drei nebenläufige Prozesse unterschiedlicher Priorität
- ***low*** belegt den kritischen Abschnitt KA
- ***high*** verdrängt *low* und bewirbt sich um KA ☞ blockiert
- ***middle*** verdrängt *low* für unbestimmte Zeit: da *high* auf *low* wartet und *low* von *middle* verdrängt wurde, muss *high* auch auf *middle* warten

☞ *middle* dominiert über *high* ⇝ Widerspruch zur Einplanungsstrategie

Konsequenz: *Prioritätsvererbung* (*priority inheritance*, [37])
- 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

☞ aufwendige Lösungen für komplexe Systeme

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 [Warum sollte das ein Problem sein?]

Nicht-blockierende 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

☞ die mit blockierenden Verfahren bestehenden Probleme werden ausgeschlossen
☞ die Alternativlösungen sind oft jedoch logisch (erheblich) komplizierter
☞ nicht alle Koordinierungsaufgaben können so gelöst werden . . .

„Wiederholtes Versuchen“ $\Longleftrightarrow$ „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 $\left\{\begin{array}{ll} \text{„Puffer“} & \text{wiederverwendbar} \\ \text{„Nachricht“} & \text{konsumierbar} \\ \text{„CPU“} & \text{verfügbar} \end{array}\right\}$?

- Prozesse dürfen aber wiederholt versuchen, gemeinsame Daten zu aktualisieren

Bezug zum Koordinierungsproblem

gegenseitiger Ausschluss ist nicht-blockierend (gut) erreichbar und gerade bei kurzen, deterministischen Anweisungsfolgen sinnvoll:

```
bufp->data[bufp->nput] = item;
bufp->nput = (bufp->nput + 1) % NDATA;
```

sonst nicht, sofern dem aktiven Warten vorgebeut werden soll:

☞ explizite Steuerung von Prozessen
☞ Verzögerung von Prozessen in Abhängigkeit von (externen) Ereignissen
☞ Austausch von Zeitsignalen oder Daten

Bounded Buffer (11) Atomarer Laufindex

```
void store (struct buffer *bufp, Any item) {
    unsigned next;

    while (bufp->nput == bufp->lget)
        await(&bufp->free, &bufp->lock);

    do next = bufp->nput;
    while (!cas(&bufp->nput, next, (next + 1) % NDATA));

    bufp->data[next] = item;

    cause(&bufp->full);
}
```

race hazard Eine evtl. Verdrängung, nachdem `next` definiert, jedoch bevor `cas()` ausgeführt worden ist, kann zur Folge haben, dass für den Prozess die zum Anfang noch ungültige Wartebedingung jetzt gültig ist. Er müsste bei scheiterndem `cas()` die Wartephase betreten, um den Pufferüberlauf abzuwenden.

☞ `do`/`while` weiter fassen.

Bounded Buffer (12) Umfassende Kontrollschleife

```
void store (struct buffer *bufp, Any item) {
    unsigned next;

    do {
        while (bufp->nput == bufp->lget)
            await(&bufp->free, &bufp->lock);
        next = bufp->nput;
    } while (!cas(&bufp->nput, next, (next + 1) % NDATA));

    bufp->data[next] = item;

    cause(&bufp->full);
}
```

race hazard Die evtl. Prozessverdrängung kann immer noch zum Pufferüberlauf führen. Dass andere Prozesse das `while` zum Überprüfen der Bedingung für die Wartephase bereits passiert haben, muss festgestellt werden können.

☞ Generationen zählen.

Bounded Buffer (13) — Generationszähler

```
void store (struct buffer *bufp, Any item) {
    unsigned next, step;

    do {
        step = bufp->tput;
        while (bufp->nput == bufp->lget)
            await(&bufp->free, &bufp->lock);
        next = bufp->nput;
    } while (!cas2(&bufp->nput, next, (next + 1) % NDATA,
                   &bufp->tput, step, step + 1));

    bufp->data[next] = item;

    cause(&bufp->full);
}
```

`tput` (*total put*) zählt jedes erfolgreiche `store()`.

`cas2()` verhält sich wie `cas()`, nur dass jetzt nicht nur ein Wort sondern zwei Worte atomar verglichen und überschrieben werden.

Wird eine „Generation" überlappt durchlaufen, scheitert `cas2()` und die Wartebedingung wird neu überprüft.
Die `cas()`-Alternative verwaltet `tput` und `nput` als ein Wort.

DCAS (*double compare and swap*) — CAS2

```
bool cas2 (word *ref1, word old1, word new1,
           word *ref2, word old2, word new2) {
    bool srZ;
    atomic();
    if (srZ = ((*ref1 == old1) && (*ref2 == old2))) {
        *ref1 = new1;
        *ref2 = new2;
    }
    cimota();
    return srZ;
}

#define dcas cas2
```


Dualität von Koordinierungstechniken

Problem	Methode
gegenseitiger Ausschluss	Schloßvariable, blockierungsfreie Algorithmen
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

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 ☞ `cas`, `cas2` (`dcas`), `cmpxchg`
RISC ☞ `ll`/`sc`

☞ *nicht-sequentielle Programmierung* ist nicht nur ein Betriebssystemfall