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

Koordination == Reihenschaltung

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

- als kritisch erachtete nebenläufige Aktivitäten der Reihe nach ausführen:
  - überlappendes Zählen
  - verdrängende Prozesseinplanung
- ”der Reihe nach“ bedeutet, gewisse Prozesse bewusst zeitlich zu verzögern
  - je nach Verfahren trifft es den {überlappenden,überlappten} Prozess
- die Verfahren arbeiten {,nicht-}wartend bzw. {,nicht-}blockierend

Arten der Koordination

”Koordination der Kooperation und Konkurrenz zwischen Prozessen“ . . .

1. derselben Inkarnation == Intraprozess-Synchronisation
   - nicht-blockierende Synchronisation ist zwingend
     - Beispiel: asynchrone Programmunterbrechung (interrupt)
2. verschiedener Inkarnationen + Interprozess-Synchronisation
   - {,nicht-}blockierende Synchronisation ist verwendbar

- Differenzierung: ein- und mehrseitige Synchronisation nebenläufiger Prozesse

Einseitige Synchronisation

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

  Bedingungssynchronisation:

  - 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*

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

Mehrseitige Synchronisation

- 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

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“) bzw. Ebene 2-Befehle: cli, sti (x86)
- Verzögerung des unterbrochenen Prozesses
  - Ebene 2-Befehle:
    - cas (IBM370, 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

```
int wheel = 0;

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

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

```
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

```
int wheel = 0;

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

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

```
int incr (int *p) {
    int x;
    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 CAS

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

- 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.

Schlossvariablen

- 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) - Probleme

```
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(„sti“); }
  void admitIRQ() { asm(„cli“); }
```

- Überprüfen und Schliessen des Schlosses bilden eine ununterbrechbare Einheit
- 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

Schlossalgorithmus (3) - Verdrängungssteuerung

```
void acquire (bool
  *lock) {
    avert();

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

```
  void avert() { preempt = false;}
  void admit() { preempt = true;}
```

- Überprüfen und Schliessen des Schlosses bilden eine überlappungsfreie Einheit
- die Schleife muss überlappbar sein, damit das Schloss aufgeschlossen werden kann

- Verdrängung des Prozesses wird abgewendet, obwohl ggf. nur einer von vielen laufähigen 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 Befehl und kein Speicherzugriff ausgeführt

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 Effzienzeinbußen sind nur dann unproblematisch, wenn jedem Prozess ein eigener realer Prozessor zur Verf¨ugung 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;
     <RACE CONDITION!>
     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:

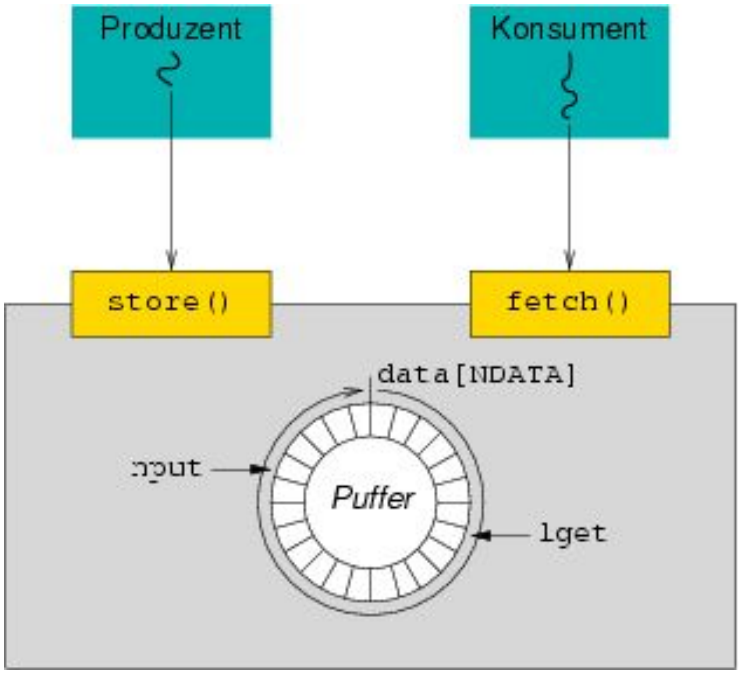
  await (*wait*) lässt einem Prozess ein Ereignis (passiv) erwarten

  - 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) - Ring Puffer

```
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?

$\Rightarrow$ 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) – Log. 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  ((bufp->lget + 1) % NDATA
             == bufp->nput)
      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.

$\Rightarrow$ Bedingungsvariable

Bounded Buffer (6) - Ring Puffer 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 = false;
    bufp->free = false;
    bufp->full = true;

}
```

- drei zusätzliche Pufferattribute koordinieren die Lese-/ Schreiboperationen:

  Schlossvariable `lock` zum gegenseitigen Ausschluss

  Bedingungsvariablen `free` und `full` zur logischen Synchronisation

Logische Synchronisation - Grundoperationen

```
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) {
   ...
   }
```

$\Rightarrow$ bedingter kritischer Abschnitt

Bedingte kritische Abschnitte

conditional critical section (resp. region)

- 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 . . .

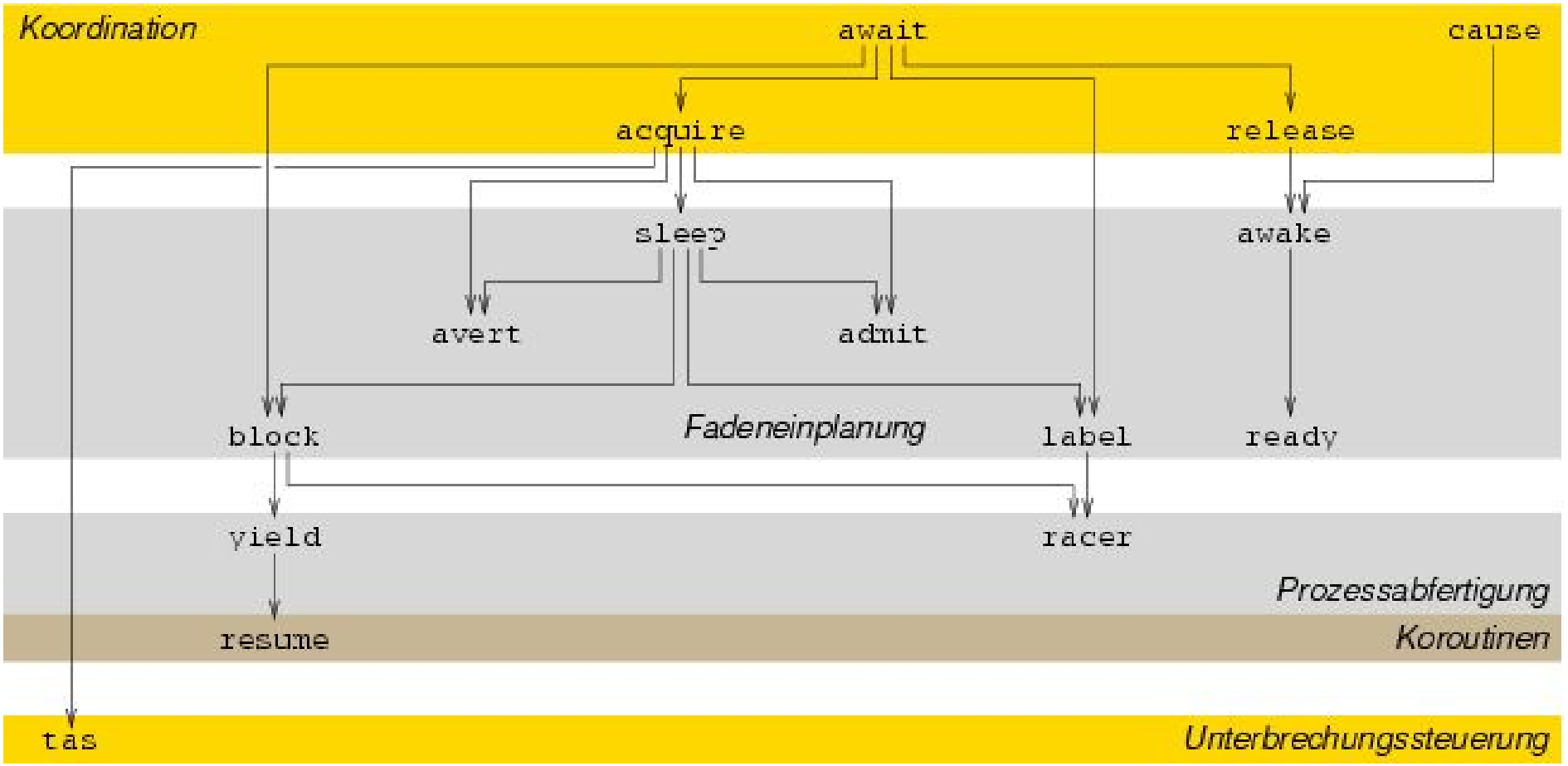
    $\Rightarrow$ blockiert der Prozess auf eine zweite Schlossvariable und

    $\Rightarrow$ gibt aber vorher die erste Schlossvariable frei

  bei (genauer: nach) Erfüllung der Bedingung . . .

    $\Rightarrow$ fordert der Prozess die erste Schlossvariable wieder an
- ggf. muss ein deblockierter Prozess die Bedingung neu auswerten

Logische Synchronisation
Funktionale Hierachie



Semaphore - *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).