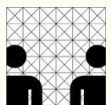


# **INHALT:**

## ***Bereich A "Betriebssysteme"***

---

- A1. Betriebssysteme: Einführung und Motivation
- A2. Prozesse: Scheduling und Betriebsmittelzuteilung
- A3. Prozesse: Synchronisation und Kommunikation**
- A4. Speicherverwaltung
- A5. Dateisysteme
- A6. Ein-/Ausgabe

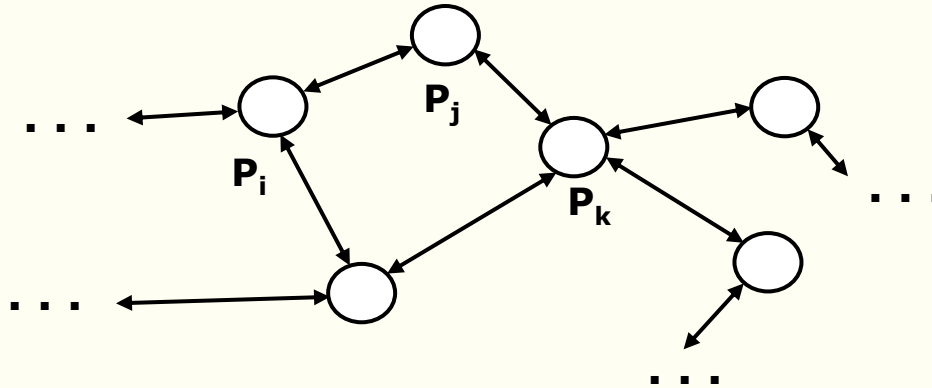


# A3 Prozesse: Synchronisation und Kommunikation

## A3.1 Grundlegende Probleme bei Interprozesskommunikation und Prozessinteraktionen

Sicht eines Betriebssystems:

Ein Betriebssystem ist eine Schar weitgehend autonomer, kooperierender Prozesse  $P_i, i \in \{1, 2, \dots, n(t)\}$ . *nota bene:*  $n(t)$ , da Anzahl Prozesse dynamisch variiert.




Verständigung der einzelnen Prozesse erfolgt über standardisierte Nachrichten („Botschaften“), z.B.



Probleme: Festlegung allgemeiner Konventionen für Formate, Protokolle, Adressen, Orte, ...

# Interprozesskommunikation: Definition und grundsätzliche Realisierungsvarianten

**Interprozesskommunikation** (*interprocess communication, IPC*): Datenaustausch (auch: "Botschaftenaustausch") zwischen Prozessen. 

- *Lokale versus entfernte IPC:*
  - **"Lokale (local) IPC"**: Kommunikation zwischen Prozessen auf gemeinsamem Rechner  
→ lokales Betriebssystem unterstützt.
  - **"Entfernte (remote) IPC"**: Kommunikation zwischen Prozessen auf verschiedenen Rechnern  
→ beide lokale Betriebssysteme und ein Kommunikations-/Rechnernetz unterstützen.
  
- Varianten bei lokaler IPC:
  - **"Feste Kopplung"** → Kommunikation über gemeinsamen Speicher(bereich); *nota bene*: nicht möglich bei remote IPC
  - **"Lose Kopplung"** → Kommunikation durch Botschaftenaustausch; *nota bene*: evtl. Botschaftenverluste bzw. signifikante Botschaftenverzögerungen zwischen Sende- und Empfangsprozess bei remote IPC

# Probleme und Anforderungen bei IPC

Wir betrachten Kommunikation zwischen sendendem Prozess P und empfangendem Prozess Q .

→ *Aufgaben und Schwierigkeiten:*

- **Botschaftentransport**
  - von P nach Q mittels Kommunikationsnetz
  - von P nach Q durch Botschaftenablage in bzw. -entnahme aus einem benutzten gemeinsamen Speicher.
- **Synchronisation zwischen P und Q** (insbes. Empfangsbereitschaft von Q sicherzustellen, Botschaftenübergabe an Q bei vorgesehener expliziter Zustellung oder aber Notwendigkeit der Synchronisation der Zugriffe auf den gemeinsamen Speicher bei direkter Kopplung → vgl. auch "Producer-Consumer"-Problem, s.u.)

Nota bene: Synchronisation zwischen Prozessen dient allgemein dazu, eine partielle zeitliche Ordnung zwischen den Prozessen herzustellen

- **Vermeiden von Deadlocksituationen** (z.B. bei Botschaftenverlust oder infolge fehlerhafter Synchronisation).

# Klassische Synchronisationsaufgaben für nebenläufige Prozesse bzw. Threads: Beispiele

- I. *Wechselseitiger Ausschluss von Prozessen* bei Eintritt in einen kritischen Abschnitt.
  - II. *Leser-Schreiber-Problem* (z.B. bei Zugriff auf gemeinsame Datenobjekte).
  - III. *Erzeuger-Verbraucher-Problem*.
  - IV. *Verwaltung gleichartiger Betriebsmittel* (z.B. bei Zugriff von Prozessen zu einem Pool gleichartiger Betriebsmittel).
- Detailliertere Behandlung der entsprechenden Problemstellungen und Lösungsansätze, s.u.

# Das Problem des wechselseitigen Ausschlusses: Kritische Abschnitte

Beispiele aus "tägl. Leben":

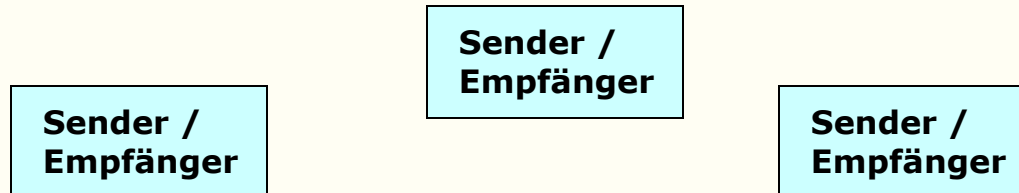
## 1. **Eingleisiger Schienenstrang:**



Mittels Fahrplan und Signalanlage verhindert man Zusammenstöße der Züge.

## 2. **Ein-Frequenz-Funkstrecke** (z.B. bei Mobilkommunikation):

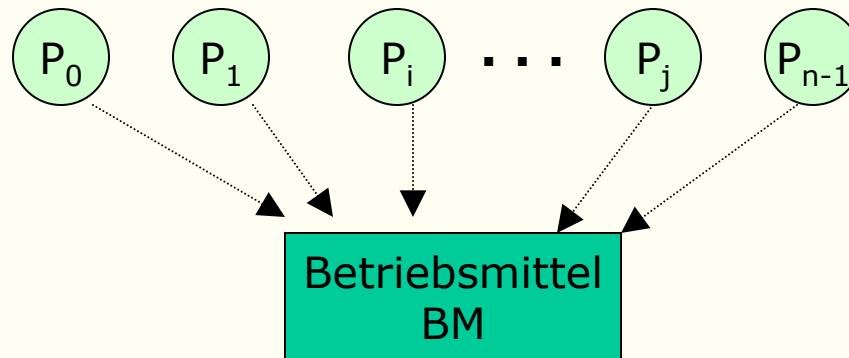
Viele Sender-Empfänger-Paare kommunizieren unkoordiniert über eine (!) Funkfrequenz.



Eine Lösung ist das ALOHA-Protokoll.

# Problem des gegenseitigen Ausschlusses (bei Zugriff auf ein gemeinsam genutztes Betriebsmittel)

Annahme:  $n$  Prozesse wollen auf BM zugreifen



Bemerkungen:

- Die Prozesse  $P_0, P_1, \dots, P_{n-1}$  benötigen für kurze, unvorhersehbare Zeitabschnitte den exklusiven Zugriff auf das Betriebsmittel BM.
- Die Prozesse  $P_0, P_1, \dots, P_{n-1}$  sind autonom und gleichberechtigt, daher ist der Einsatz eines zentralen Verwalters nicht angebracht.

Verallgemeinerung:

Prozesse konkurrieren nicht um ein einziges, sondern um  $k$  ( $=\text{const.}$ ) gleichartige BM.

# Formulierung der Aufgabe des gegenseitigen Ausschlusses als Koordinierungsaufgabe für Prozesse

Man habe  $n \geq 2$  zyklische Prozesse der Form:

$P_i (1 \leq i < n)$ :

*while true do*

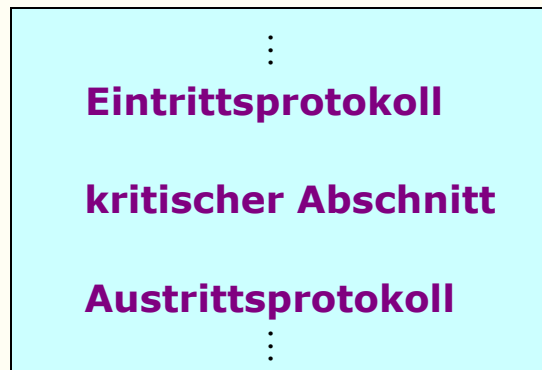
$\vdots$

kritischer Abschnitt

$\vdots$

*end*

Jeder kritische Abschnitt wird eingefasst durch ein Eintritts- und Austrittsprotokoll, wie folgt:





# Bedingungen für eine gute Lösung des MUTEX-Problems

⇒ 4 BEDINGUNGEN :

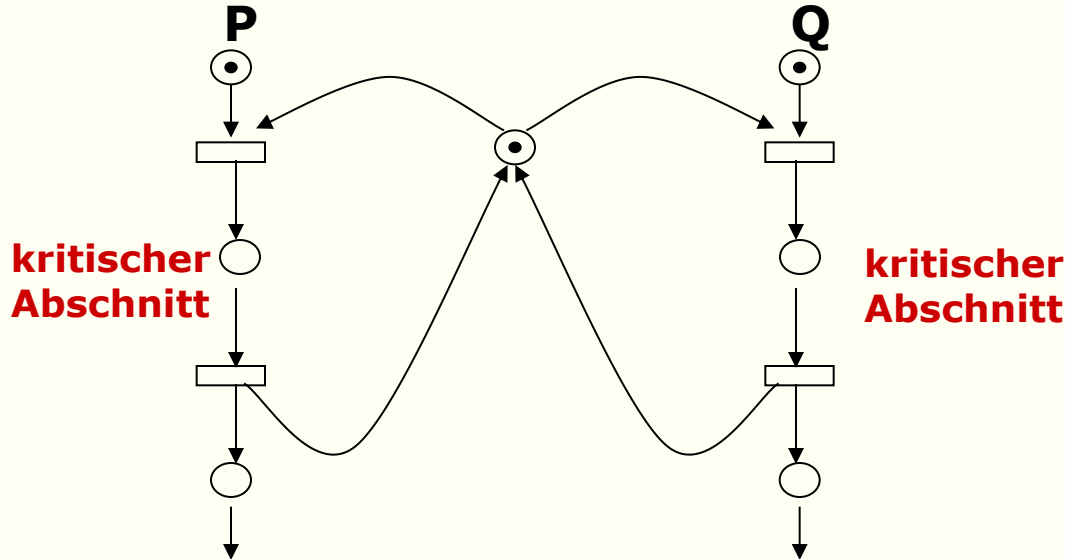
- B1. Zu jedem Zeitpunkt befindet sich höchstens ein Prozess in seinem kritischen Abschnitt.
- B2. Jeder Prozess muss sein Eintrittsprotokoll, seinen kritischen Abschnitt und sein Austrittsprotokoll in endlicher Zeit durchlaufen. Dies beinhaltet, dass die Entscheidung, welcher Prozess seinen kritischen Abschnitt betritt, nicht beliebig lange hinausgeschoben werden darf und dass jeder Prozess, der es wünscht, auch tatsächlich in seinen kritischen Abschnitt gelangt.
- B3. Außerhalb des Blocks Eintrittsprotokoll, kritischer Abschnitt, Austrittsprotokoll darf ein Prozess keinen Prozess (auch nicht sich selbst) behindern.
- B4. Die Arbeitsgeschwindigkeit jedes Prozesses ist größer als Null; ansonsten sind die Arbeitsgeschwindigkeiten der Prozesse unbekannt.

# Zusammenfassung: Begriffe zu wechselseitigem Ausschluss

- **kritischer Abschnitt** (*critical section*): Bereich (z.B. Programm-Code, Sequenz von Updates von Datenobjekten, ...), den zu einem Zeitpunkt nur max. 1 Prozess betreten darf
- **gegenseitiger Ausschluss** (*mutual exclusion*): Verfahren, das andere Prozesse daran hindert, den kritischen Abschnitt zu betreten, sofern sich bereits ein Prozess im kritischen Abschnitt befindet.

*Am Rande:* kurzer Bezug zu Vorlesungen mit Methoden zur formalen Beschreibung des MUTEX-Problems:

→ Bsp. Petri-Netz mit ○ : Stellen, □ : Transitionen, • : Marken



# Beispiel Flugbuchung

- **Gegeben :** Flugzeug mit  $c_0$  Plätzen und Passagierliste  
PASSENGER\_NAME  $[1, \dots, c_0]$   
z.B. als TEXT-Array

→ **bei neuer Buchung :**

**Schritt 1:** Test, ob für die Anzahl der bereits gebuchten Plätze ( $c$ ) gilt,  $c < c_0$  ? (\*)

**Schritt 2a:** wenn *ja* bei Test (\*):  $c := c + 1$ ;  
write PASSENGER\_NAME ( $c$ );

**Schritt 2b:** wenn *nein* bei Test (\*):  
keine Aktion, da Flug ausgebucht.

## **Angenommene Ausgangssituation :**

- Zu  $t = t_0$  sei  $c = 99$  (bei  $c_0 = 100$ ) und Prozess P möchte für Passagier Müller und Q für Meyer buchen
- Chaos entsteht, z.B. sofern:
  1. P testet und wird direkt anschließend von Q unterbrochen
  2. Q testet, erhöht  $c$  und bucht Meyer als 100. Passagier, Q terminiert anschließend normal
  3. P setzt seinen Programm-Code nach erneuter Aktivierung fort; da Test  $c < c_0$  bereits früher abgeschlossen, wird nunmehr direkt  $c$  erhöht und Müller als 101. Passagier gebucht (evtl. Laufzeitfehler durch Überschreiten der Array-Grenzen).

# Naheliegende Lösungen für Mutex – die nicht funktionieren! (Teil I)

## Sperren von Unterbrechungen (als Hardwarelösung)

### Idee:

- Vor Eintritt in krit. Abschnitt (seitens Prozess P) sperrt P alle Unterbrechungen
- Bei Austritt aus krit. Abschnitt gibt P die Unterbrechungen wieder frei

### Aber:

- Was geschieht, wenn Unterbrechungen nicht mehr freigegeben werden? (z.B. bei „Absturz“ von P)
- Überdies: bei Multiprozessorsystem wirken sich Unterbrechungssperren nur auf eine einzige CPU aus

### Resümee:

- für Betriebssystemkern Sperren von Unterbrechungen akzeptabel zur Lösung des MUTEX-Problems, da bei „Absturz“ des Betriebssystems ohnehin gilt: „Rien ne va plus!“

### jedoch:

für Benutzerprozesse Sperren von Unterbrechungen völlig inakzeptable Lösung!

# Naheliegende Lösungen für Mutex – die nicht funktionieren! (Teil II)

## Synchronisationsvariable LOCK (als Softwarelösung)

### Idee:

Gegeben 2-wertige Synchronisationsvariable LOCK mit Wertebelegung

- 0 (bzw. false): kein Prozess im kritischen Abschnitt
- 1 (bzw. true): ein Prozess im kritischen Abschnitt

→ Vor Eintritt in kritischen Abschnitt (seitens Prozess P) liest P den aktuellen Wert von LOCK:

- ist LOCK = 0 :  
P setzt LOCK: = 1 und betritt kritischen Abschnitt; direkt nach Verlassen des kritischen Abschnitts setzt P wieder LOCK: = 0
- ist LOCK = 1 :  
P wartet solange bis gilt LOCK = 0

### Aber:

... bei zeitlicher Abfolge :

1. Prozess P prüft LOCK und es gilt LOCK = 0
2. P wird durch Prozess Q unterbrochen
3. Q prüft LOCK und es gilt noch immer LOCK = 0
4. Q setzt LOCK: = 1 und betritt kritischen Abschnitt
5. P erhält CPU-Zeit, ehe Q kritischen Abschnitt verlassen hat und somit setzt P nochmals LOCK: = 1 und betritt kritischen Abschnitt

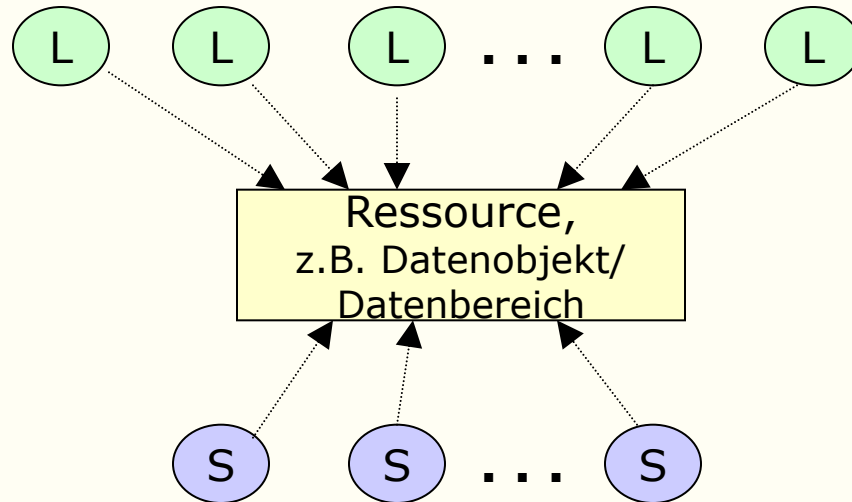
... haben wir ein Problem !!!

... das auch nicht verschwindet, wenn P vor Setzen von LOCK: = 1 den Wert von LOCK erneut prüfen würde

→ weshalb bleibt das Problem bestehen ?

# Das Leser-Schreiber-Problem

*Annahme* :  $n$  Leseprozesse (L) und  $m$  Schreibprozesse (S) wollen auf BM (z.B. auf ein Datenobjekt) zugreifen



*Bedingungen:*

- B1.** Jeder Schreibprozess benötigt exklusiven Zugriff auf die Ressource.
- B2.** Mehrere Leseprozesse können gleichzeitig auf die Ressource zugreifen.
- B3.** Es existiert keine zwischen den Prozessen vermittelnde Instanz.

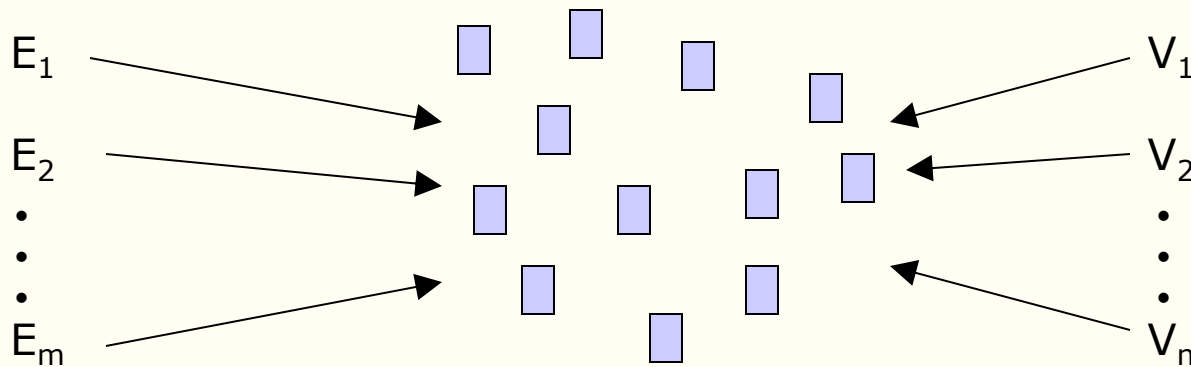
# Das Erzeuger-Verbraucher-Problem

alias: Produzenten-Konsumenten ("Producer-Consumer")-Problem

Erzeuger

Puffer  
beschränkter  
Kapazität

Verbraucher

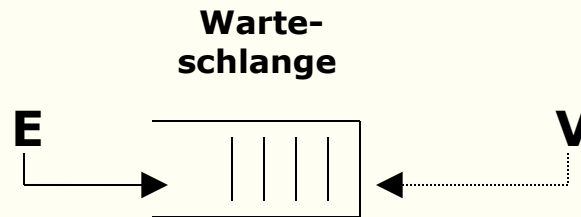


Erzeuger erzeugen Elemente und legen sie im Pufferbereich ab;  
Verbraucher entnehmen Elemente aus dem Puffer und verarbeiten sie.

Randbedingung: Ordentlicher Durchlauf der Elemente durch den Puffer;  
es darf kein Element vernichtet werden, es darf kein Element zweimal  
aus dem Puffer entnommen werden.

# Das Erzeuger-Verbraucher-Problem: Puffer als Elementwarteschlange

*Sonderfall:*



Puffer als Elementwarteschlange (WS) und nur jeweils ein Erzeuger (E) und ein Verbraucher (V).

E fügt Elemente in WS ein (z.B. an Ende);  
V entnimmt Element an Kopf von WS

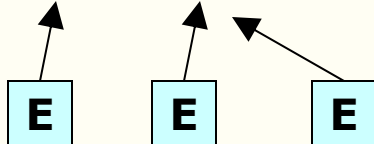
→ beschriebenes Procedere entspricht einer *First-Come-First-Served* alias *First-In-First-Out* Bedien-(Bearbeitungs-)Strategie



# Das Erzeuger-Verbraucher-Problem: Organisation des Pufferzugangs

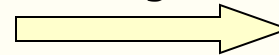
## **Warteschlange für Erzeuger:**

... | E08 | E17 | ... | E11 | E15



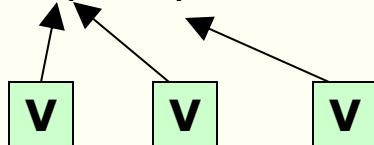
Einordnen der Erzeuger in Warteschlange

Einfügen



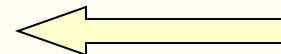
## **Warteschlange für Verbraucher:**

... | V13 | V12 | ... | V06



Einordnen der Verbraucher in Warteschlange

Entnehmen



**Exklusiv-  
Bereich  
für Puffer**

# A3.2 Mechanismen zur Realisierung von wechselseitigem Ausschluss

## Die "Test and Set Lock (TSL)"-Instruktion

Lösung des MUTEX-Problems auf Basis eines speziellen Maschinenbefehls

- **TSL: Test and Set Lock**

→ Funktionalität von TSL:

1. Lesen von Speicherwort (z.B. an Speicheradr. X) in Register
2. Schreiben eines Wertes  $\neq 0$  (z.B. den Wert 1) in die Speicheradr. X,

wobei die Schritte 1. und 2. eine **unteilbare** Operation bilden.

*Nota bene:*

TSL beseitigt die für die Lösung des MUTEX-Problems schädliche Unterbrechbarkeit von Prozessen zwischen dem Testen des alten und Setzen eines neuen Wertes einer Variablen.

# Realisierung von gegenseitigem Ausschluss mittels TSL

- Grobstruktur der Prozesse, die kritischen Abschnitt betreten möchten:

```
enter_region      /* Unterroutine zur Koordination des Eintritts in krit. Abschnitt
Anw. 1           }
...             } Anweisungen des krit. Abschnitts
Anw. n           }
leave_region     /* Unterroutine zur Aktualisierung des Belegungszustands
                   des krit. Abschnittes (... nunmehr: frei)
```

- Nutzung einer Variablen *flag*, die anzeigt, ob sich ein Prozess in krit. Abschnitt befindet (*flag* = 1) oder nicht (*flag* = 0) sowie – beispielsweise – dem folgenden Maschinenbefehlscode für die benutzten Unterroutinen *enter\_region* und *leave\_region* (s.o.)

## **enter\_region**

```
tsl R1, flag      /* kopiert flag in das Register R1 und setzt flag auf 1
cmp R1, #0        /* war flag = 0 ?
jnz enter_region  /* wenn ≠ 0, dann war Verriegelung gesetzt
                   → Schleife durchlaufen
ret               /* Rückkehr zum Aufrufer; krit. Abschn. wird betreten
```

## **leave\_region**

```
mov flag, #0      /* Speichern einer 0 in flag
ret               /* Rückkehr zum Aufrufer
```

# Beurteilung der "Test and Set Lock (TSL)"-Lösung

---

- relativ ineffizient, infolge des aktiven Wartens auf den Zustand „*flag = 0*“ (s.o.)
- für korrekte Funktionsweise müssen konkurrierende Prozesse die Unterroutrinen *enter\_region* und *leave\_region* immer paarweise, vollständig und in korrekter Reihenfolge aufrufen  
→ was tun, wenn Prozess in kritischem Abschnitt „abstürzt“ ?!

Ergo: Wir wünschen uns programmiersprachliche bzw. Betriebssystem-Unterstützung für die Lösung des MUTEX-Problems, wie z.B. durch das SEMAPHOR-Konzept (s.u.), das MONITOR-Konzept, o.ä.

# Das "Semaphor"-Konzept

Lösung des MUTEX-Problems auf Basis von Semaphoren

**Semaphor:** Spezieller Variablentyp, für den 3 Operationen definiert sind:

- **Initialisierung** (mit ganzzahligem Wert)
- **P-Operation** (*P* für „prolagen“ aus „proberen“ und „verlagen“, holländ.; besagt: „prüfen“ und „erniedrigen“)
- **V-Operation** (*V* für „verhogen“, holländ.; besagt: „erhöhen“)

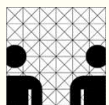
SEMAPHOR-Konzept erfunden durch: **E.W. Dijkstra** (1965), niederländ. Math./Informatiker

Semaphore insbes. geeignet zur Lösung des relativ allg. Synchronisationsproblems des Typs:

- es existieren  $n$  gleichartige Betriebsmittel  $B_1, B_2, \dots, B_n$  ( $n = \text{const.}$ )
- jeder der konkurrierenden Prozesse benötigt zur weiteren Ausführung temporär genau ein Betriebsmittel  $B_i$  ( $i$  beliebig)
- ist beim ersten Versuch, ein Betriebsmittel zu erhalten, keines verfügbar, so legt sich der Prozess schlafend und ist zu wecken, sofern wieder Betriebsmittel vorhanden sind.

*Nota bene:* Für  $n=1$  entspricht das beschriebene Problem dem MUTEX-Problem

**Idee :** {  
- Initialisierung des Semaphors mit der Anzahl  $n$  der insgesamt verfügbaren BM  
- Aufwecksignale schlafend gelegter Prozesse zählen

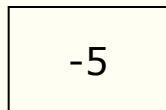


# Präzisierung des „Semaphor“-Konzepts

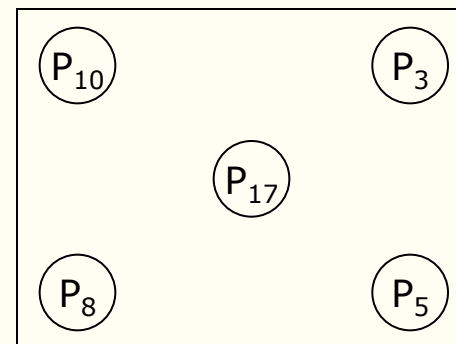
Interpretation der Semaphore unter Nutzung einer Warteschlange für temporär blockierte Prozesse:

„Gedächtnis“ der Warteschlange:

Zähler Z



Ungeordneter Warteraum W  
(mit blockierten Prozessen)



Interpretation des Zählerwertes (z):

- $z \geq 0$ : Anzahl der „Gutschriften“, um am Warteraum vorbeizugehen
- $z < 0$ :  $|z|$  ist Zahl der Prozesse im Warteraum

# Präzisierung des „Semaphor“-Konzepts (Forts.)

Verfeinerung der P- und V-Operationen:

- **P-Operation:**
  1.  $z := z - 1$
  2. if  $z < 0$  then ausführenden Prozess in W einreihen
- **V-Operation:**
  1.  $z := z + 1$
  2. if W nicht leer then wartenden Prozess aus W entfernen und in Zustand „bereit“ versetzen; eigene Tätigkeit fortsetzen

nota bene:

Die *Unteilbarkeit sowohl der P- als auch der V-Operation* ist zu gewährleisten (z.B. seitens des Betriebssystems)

# "Semaphor": Präzisierung I

**semaphore** (n: integer) : class

var            s: integer, := n  
              q: queue of processes

## **operation P**

begin

  s := s - 1;

  if s < 0 then

    "P-Operation ausführender Prozess wird in q eingereiht"

  end

end P;

## **operation V**

begin

  s := s + 1;

  if not empty (q) then

    "Ein Prozess wird aus q entfernt, der V ausführende Prozess setzt  
    seine Tätigkeit fort."

  end

end V;

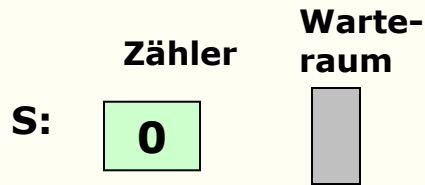
end semaphore

*Bemerkung:* Die Initialisierung eines Semaphors erfolgt mit seiner Schaffung.



# "Semaphor"-Präzisierung I: Veranschaulichung

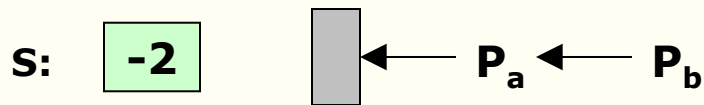
Startzustand des Semaphors S:      leerer Warteraum, Zähler = 0.



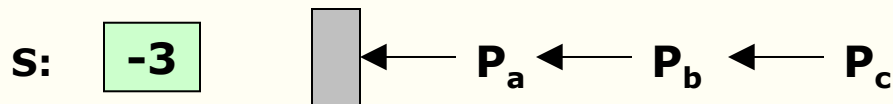
Prozess  $P_a$  führt  $P(S)$  aus.



Prozess  $P_b$  führt  $P(S)$  aus.

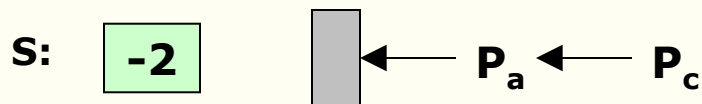


Prozess  $P_c$  führt  $P(S)$  aus.



Prozess  $P_d$  führt Freigabeoperation  $V(S)$  aus.

Bemerkung: Durch die V-Operation in  $P_d$  wird einer der Prozesse  $P_a$ ,  $P_b$ ,  $P_c$  lauffähig.  
Die Auswahl ist implementierungsabhängig.



# "Semaphor": Präzisierung II

**semaphore** (n: natural) : class

var            s: natural, := n;

**operation P**

begin

  if s > 0 then

    s := s - 1

  else

    "Warte bis s > 0 wird und  
    führe dann s := s - 1 aus."

  end

end P;

**operation V**

begin

  s := s + 1

end V;

end semaphore

*Bemerkungen:*

- Die Initialisierung eines Semaphors erfolgt mit seiner Schaffung.
- „natural“ steht für die Menge der natürlichen Zahlen.

# "Semaphor"-Präzisierung II: Veranschaulichung

Semaphor S wird ins Leben gerufen.

S: **Zähler**  
**0**

Drei Prozesse versuchen, auf S eine P-Operation auszuführen.

S: **0**

$P_a : \dots P(S)$   
 $P_b : \dots P(S)$   
 $P_c : \dots P(S)$



Alle drei Prozesse warten in ihrer P-Operation

Prozess  $P_d$  führt nun Freigabeoperation  $V(S)$  aus. Damit kann ein wartender Prozess (hier:  $P_c$ ) seine P-Operation beenden.

S: **0**

$P_a : \dots P(S)$   
 $P_b : \dots P(S)$



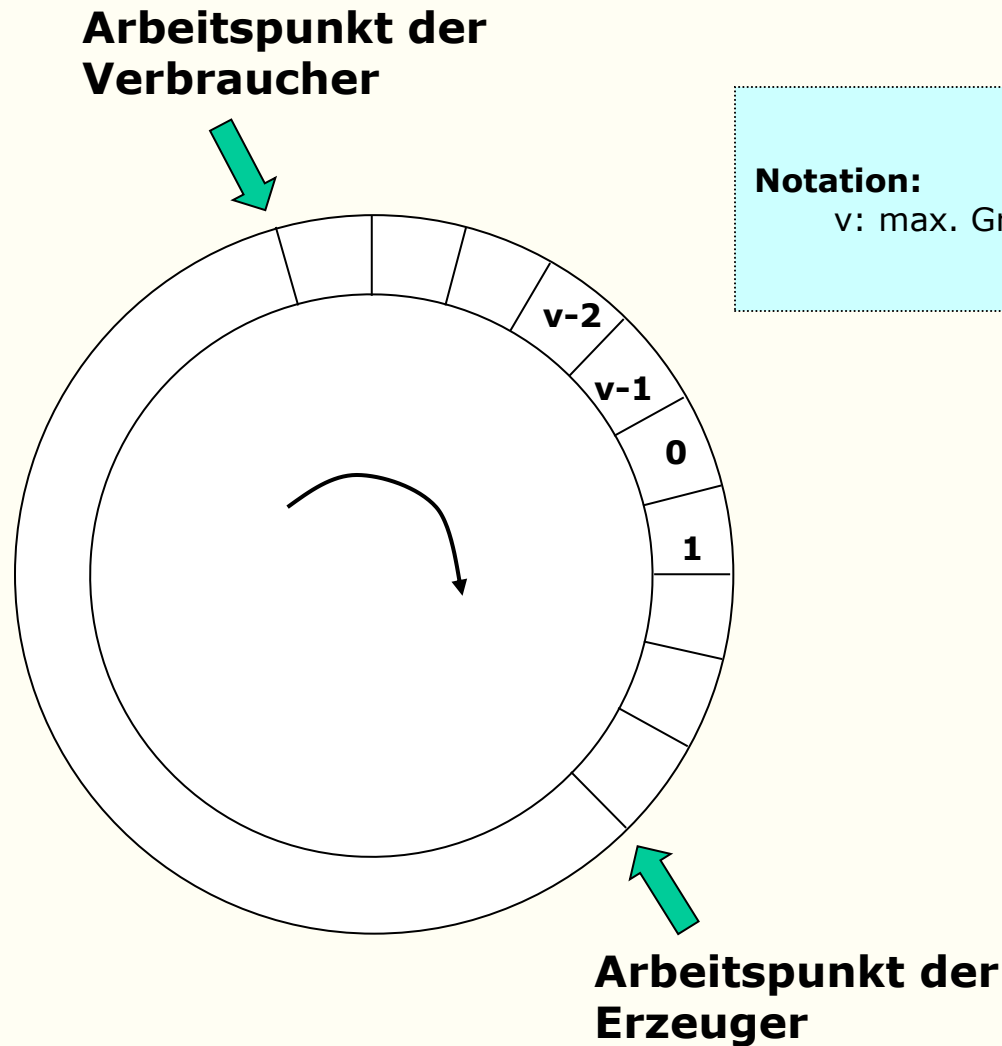
Beide Prozesse warten in ihrer P-Operation

# Bemerkungen

---

1. Die Operationen P und V sind *unteilbar*.
2. Warten mehrere Prozesse auf eine V-Operation, dann sagt die Definition des Semaphors nichts darüber aus, welcher wartende Prozess seine Tätigkeit wieder aufnimmt.
3. Man unterscheidet zwischen starker und schwacher Semaphorimplementation, eine starke Implementation vermeidet "ewiges" Warten.
4. Darf die Ganzzahlvariable des Semaphors nur die Werte 0 und 1 annehmen, dann spricht man von einem *binären Semaphor*.
5. Es gibt noch weitere Präzisierungen des Semaphorbegriffs.

# Zyklischer Puffer zur Illustration des Erzeuger-Verbraucher-Problems



**Notation:**

$v$ : max. Größe des Puffers (Kapazität)

# Programmierung des Erzeuger-Verbraucher-Problems mittels Semaphoren

```
var
    buffer:    array 0 ... v-1 of ...
    in:        0 ... v-1, := 0;
    out:       0 ... v-1, := 0;

    empty:     semaphore (v);
    full:      semaphore (0);
    mutex:     semaphore (1);
```

## Notation:

e: Erzeugtes/verbrauchtes Element  
v: max. Größe des Puffers (Kapazität)  
buffer [x]: Element an Position x

### Struktur eines Erzeugers:

```
loop
    produce element e;
    P (empty);
    P (mutex);

    buffer [in] := e;
    in := (in + 1) mod v;
    V (mutex);
    V (full);

end
```

### Struktur eines Verbrauchers:

```
loop
    P (full);
    P (mutex);

    e := buffer [out];
    out := (out + 1) mod v;
    V (mutex);
    V (empty);
    consume element e;

end
```

**Bemerkungen:** (i) Aufbau von Erzeuger und Verbraucher sind symmetrisch.  
(ii) Die gezeigte Lösung ist unnötig restriktiv.

# Beispiele für fehlerhafte Benutzung von Semaphoren

**Beispiel 1** : Aufgrund eines Schreibfehlers wird aus einer V-Operation eine P-Operation.

```
S:  semaphore (1);  
    :  
    P(S)  
    "kritischer Abschnitt"  
    P(S) (* Verklemmungsgefahr *)  
    :
```

**Beispiel 2** : Ein Benutzer verwechselt die Bedeutung der P- und V-Operationen.

```
S:  semaphore (1);  
    :  
    V(S)  
    "kritischer Abschnitt"  
    P(S)  
    : (* Zwangssequentialisierung *)  
    : (* unkritischer Teile *)
```

# Weiteres Beispiel für die Benutzung von Semaphoren

**Beispiel 3** : Schachtelung kritischer Abschnitte.

**S1, S2: semaphore (1);**

**Prozess 1:**            . . . P(S1) . . .  
                                 . . . P(S2) . . . V(S2)  
                                 V(S1) . . .

**Prozess 2:**            . . . P(S2) . . .  
                                 . . . P(S1) . . . V(S1)  
                                 V(S2) . . .

Bemerkung: Im Mittel müssen in betrachteten Zeitabschnitten etwa gleich viele P- und V-Operationen stattfinden. Eine falsch platzierte P-Operation kann zu einer partiellen Verklemmung führen, eine falsch platzierte V-Operation kann den Zweck des Semaphoreinsatzes zunichte machen. Daher wird häufig die Forderung aufgestellt, dass P- und V-Operationen nur paarweise benutzt werden dürfen.

Der Hauptnachteil des Semaphorkonzeptes:

*Das Semaphorkonzept ist zu primitiv !*

→ Ausweg : „Monitor“-Konzept u.ä., zu Details, vgl. [Tan 08], [NeS 01]



# Zusammenfassende Bemerkungen

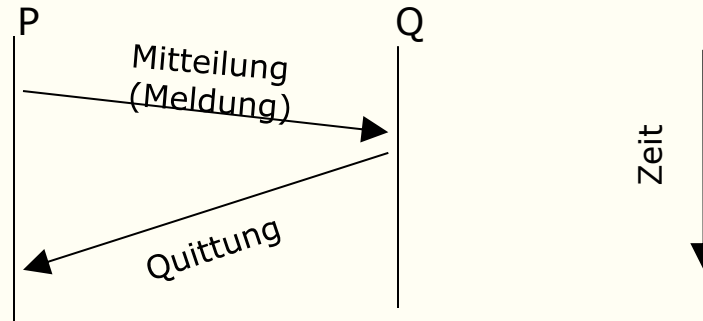
1. Ein Betriebssystem ist strukturierbar als eine Schar kooperierender Prozesse.
2. Damit ein Prozess in seiner Ausführung voranschreiten kann, müssen sinnvolle Umgebungsbedingungen erfüllt sein.
3. Jeder Prozess prüft die Erfüllung seiner Arbeitsbedingungen, bei Nichtvorliegen derselben wechselt er in einen Wartezustand.
4. Aus einem Wartezustand wird ein Prozess nur von einem anderen Prozess befreit.
5. Durch die Verpflichtung zur Kooperation wird aktives Warten überflüssig.
6. Programmtechnisch lassen sich Wartebedingungen über Semaphore realisieren.
7. Semaphore sind hardwarenahe Synchronisationsprimitive. Zur Strukturierung komplexer Sachverhalte bedient man sich *höherer Konzepte*, wie z.B.
  - der **Monitore**,
  - der **bedingten kritischen Regionen** oder
  - eines **Nachrichtensystems**.(zu Details, vgl. einschlägige Literatur wie [NeS 01], [Tan 08], ...),

# A3.3 Zur Realisierung lokaler Interprozesskommunikation

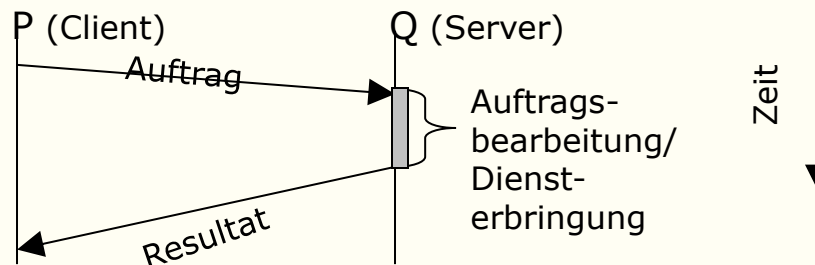
## Kommunikationsmuster

Formen der Prozessinteraktion mittels Nachrichtenaustausch zwischen Prozessen P und Q:

- **Kommunikationsmuster 1:**  
Einfache Mitteilung von P → Q mit anschließender Quittierung (pos., neg.) von Q → P



- **Kommunikationsmuster 2:**  
Client-Server-Kommunikation zwischen „Client“ P (Auftraggeber, Dienstbenutzer) und „Server“ Q (Dienstleister)



# Synchronität

Betrachteter Aspekt: Grad der zeitlichen Kopplung der kommunizierenden Prozesse beim Nachrichtenaustausch

- **asynchrone Nachrichtenkommunikation:**
  - Sender und Empfänger einer Nachricht zeitlich voneinander entkoppelt  
[hier: Pufferung von Nachrichten zwischen Sender und Empfänger notwendig, da evtl. keine sofortige Empfangsbereitschaft gegeben]
- **synchrone Nachrichtenkommunikation:**
  - Sender nach Sendevorgang blockiert bis Quittung (als Antwort auf eine Meldung) bzw. Resultat (als Antwort auf einen Auftrag) erhalten wird  
[hier: Nachrichtenpufferung im Kommunikationssystem unnötig, da Empfangsbereitschaft gegeben ist; ABER: eingeschränkte Parallelität bei Nachrichtenübertragung infolge der temporären Blockierung des Senders]

# Resultierende Kommunikationsmodelle

- Aus Art des Kommunikationsmusters und der Synchronisation resultierende Kommunikationsmodelle:

		Synchronität	
		asynchron	synchron
Kommunikations- muster	Meldung	<b>asynchrone Meldungen</b>	<b>synchrone Meldungen</b>
	Auftrag	<b>asynchrone Aufträge</b>	<b>synchrone Aufträge</b>

# Datenfluss über verschiedene Adressräume

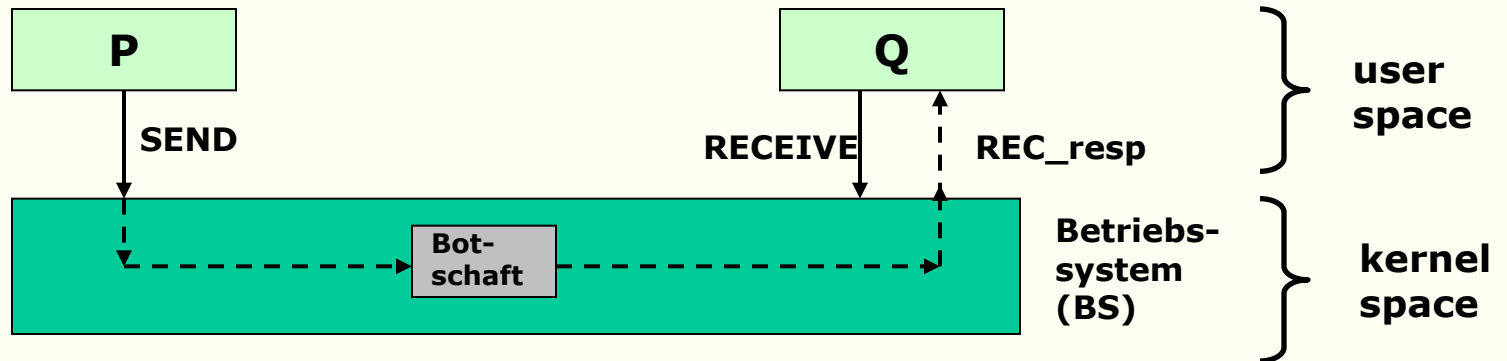
---

Annahmen:

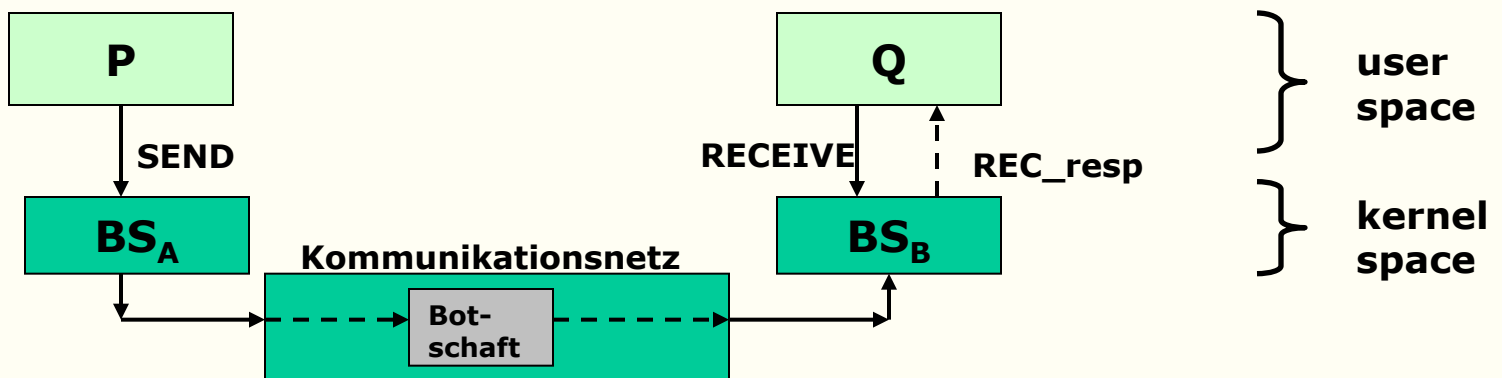
- Kommunikation zwischen Prozessen P und Q über ein Kommunikationssystem KS (rechnerintern bei lokaler IPC bzw. über Rechengrenzen hinweg bei entfernter IPC)
- Nutzung von SEND-Operationen (z.B. Funktionsaufrufe), um Sendefunktionen des KS zu initiieren
- Nutzung von RECEIVE-Operationen, um dem KS gegenüber Empfangsbereitschaft anzuzeigen
- Nutzung der REC\_resp-Operation (RECEIVE\_response), um damit eingetroffene Daten an ihren Empfänger Q auszuliefern.

# Datenfluss bei lokaler und entfernter IPC

⇒ Datenfluss bei lokaler IPC :



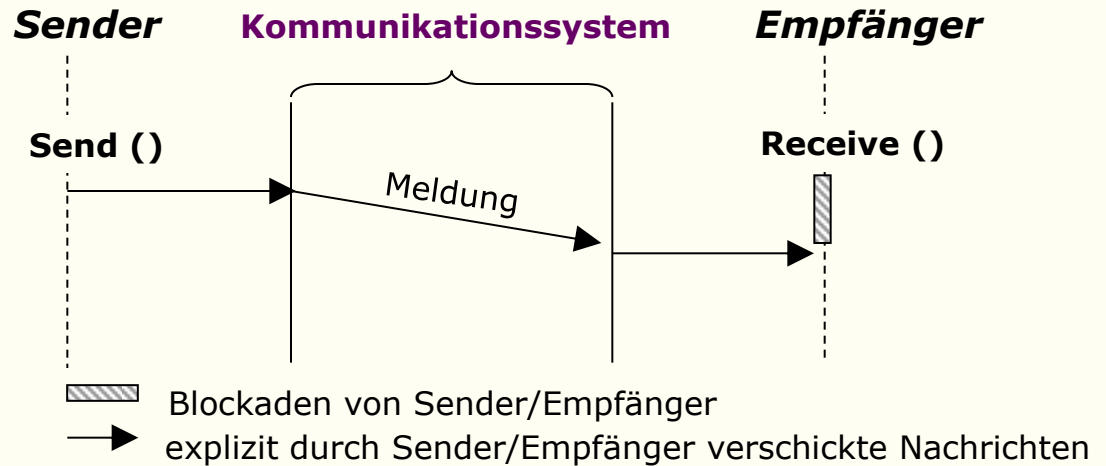
... und Datenfluss bei IPC über Rechnergrenzen hinweg :



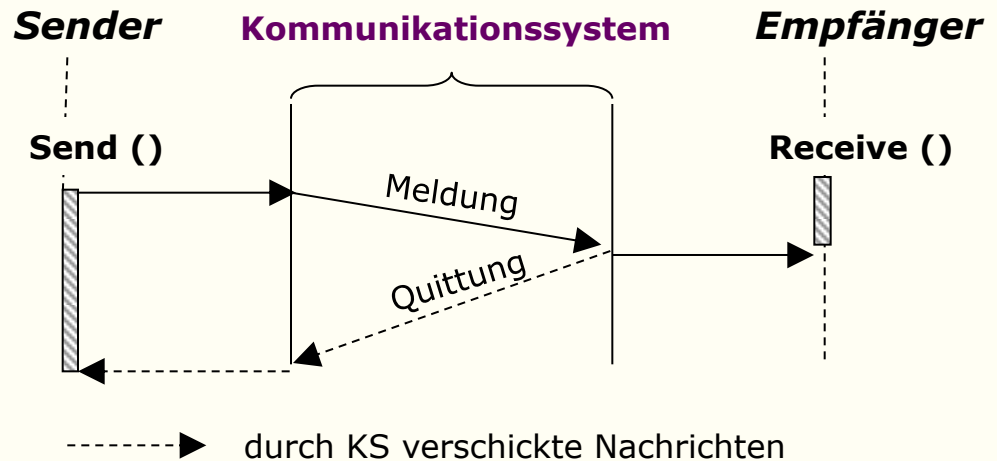
# Synchrone versus asynchrone Meldung

(siehe [NeS98])

## ➤ Asynchrone Meldung:

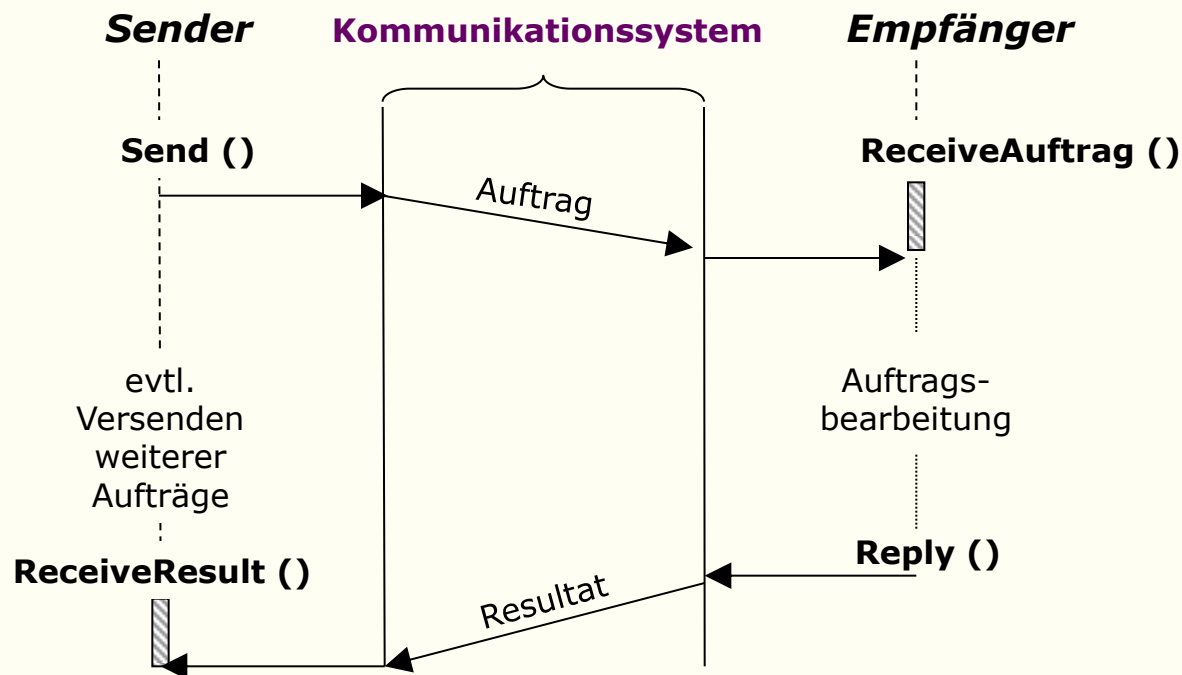


## ➤ Synchrone Meldung:



# Synchroner versus asynchroner Auftrag I

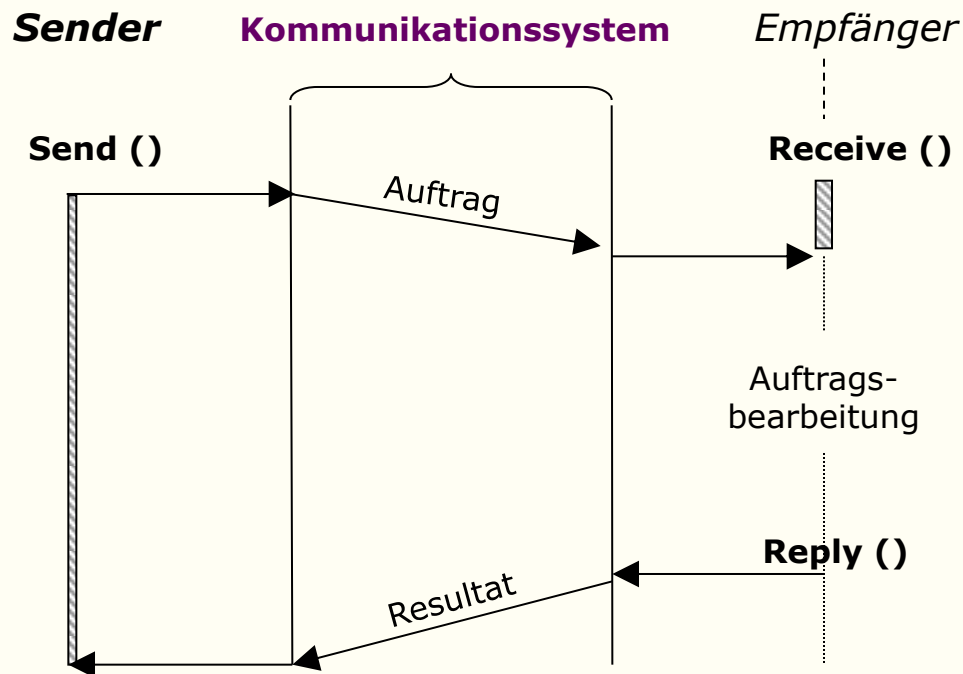
## ➤ Asynchroner Auftrag:





# Synchroner versus asynchroner Auftrag II

## ➤ Synchroner Auftrag:



*Bemerkung:* weitere Details zu lokaler und nicht-lokaler IPC in Kapiteln zu Rechnernetzen !