Work in progress

Nachrichtenaustausch

Ute Bormann, TI2

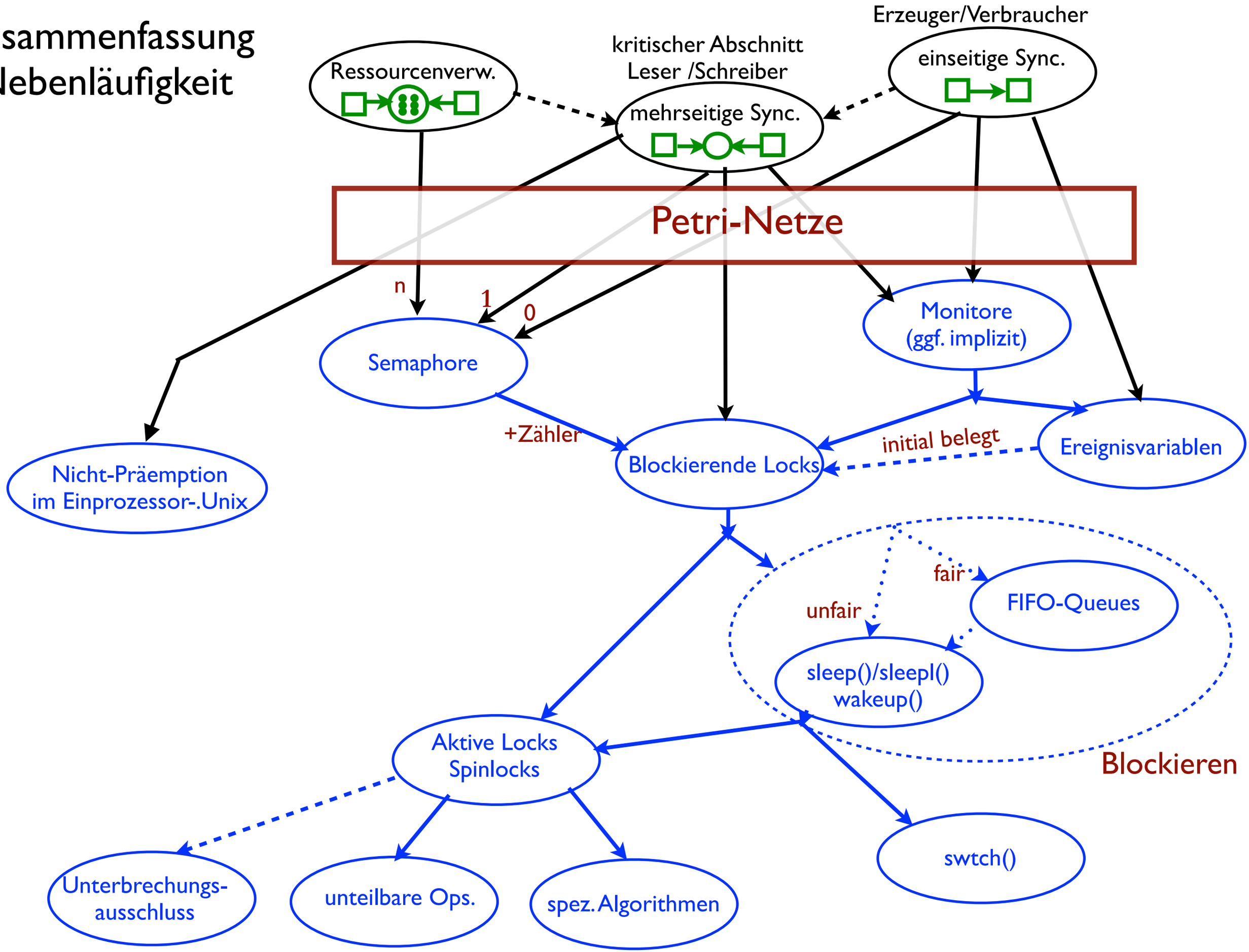
2023-10-13

Inhalt

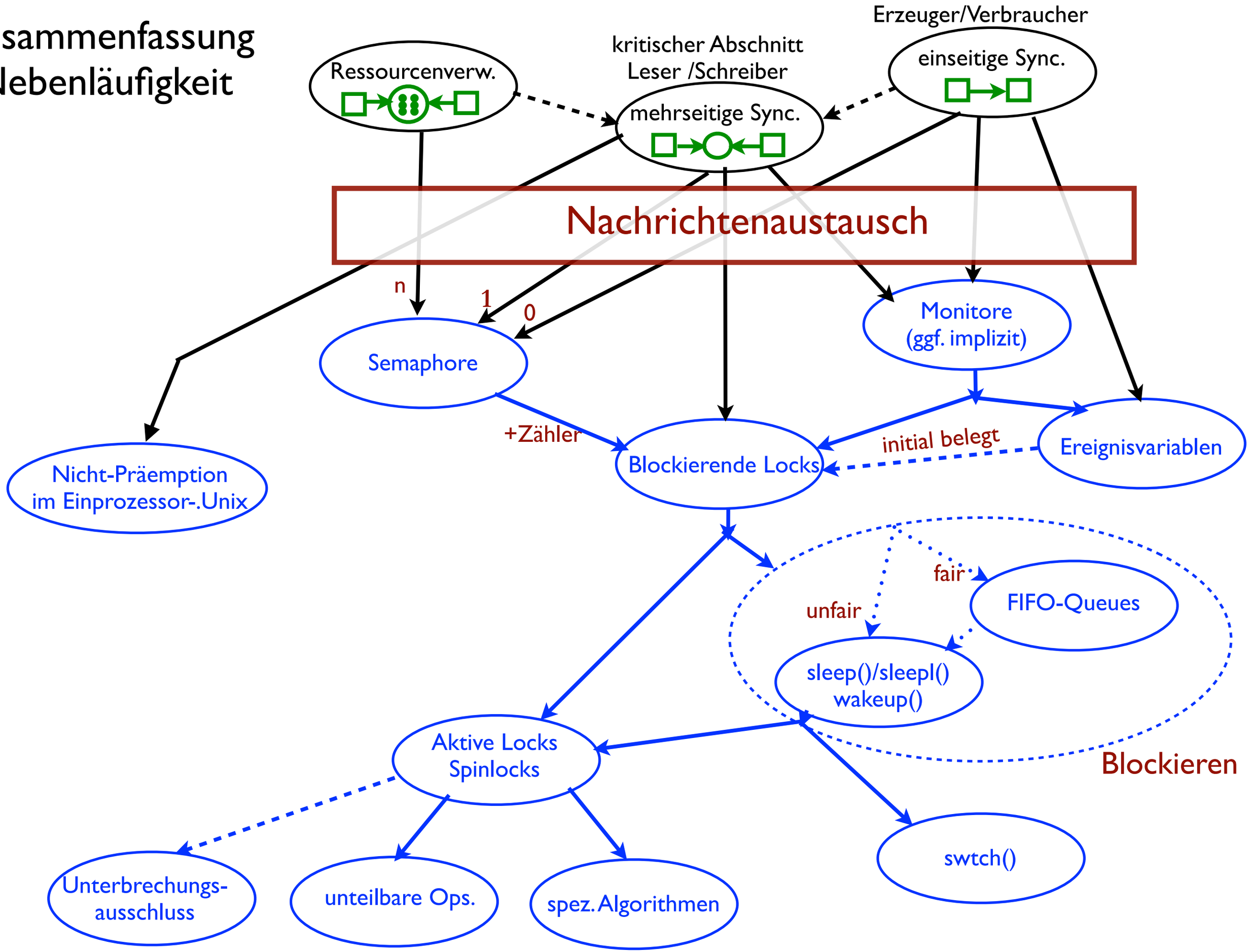
1. Grundprinzipien des Nachrichtenaustauschs
2. Communicating Sequential Processes (CSP)
3. Das Sieb des Eratosthenes in CSP

Teil 1:
Grundprinzipien des Nachrichtenaustauschs

Zusammenfassung Nebenläufigkeit



Zusammenfassung Nebenläufigkeit



Nachrichtenaustausch

- Bisher: Synchronisation/Kommunikation zwischen Prozessen/Threads über gemeinsame Variablen/Objekte (Schlossvariablen, Semaphore, ...)

  ⇒ setzt gemeinsamen Adressraum voraus
  ⇒ nicht immer gegeben

- Gegenbeispiele:
  - Unix-Prozesse (statt Threads) ohne Shared Memory
  - Verteilte Systeme/Rechnernetze

Nachrichtenaustausch

- Bisher: Synchronisation/Kommunikation zwischen Prozessen/Threads über gemeinsame Variablen/Objekte (Schlossvariablen, Semaphore, ...)

  ⇒ setzt gemeinsamen Adressraum voraus
  ⇒ nicht immer gegeben

- Gegenbeispiele:
  - Unix-Prozesse (statt Threads) ohne Shared Memory
  - Verteilte Systeme/Rechnernetze

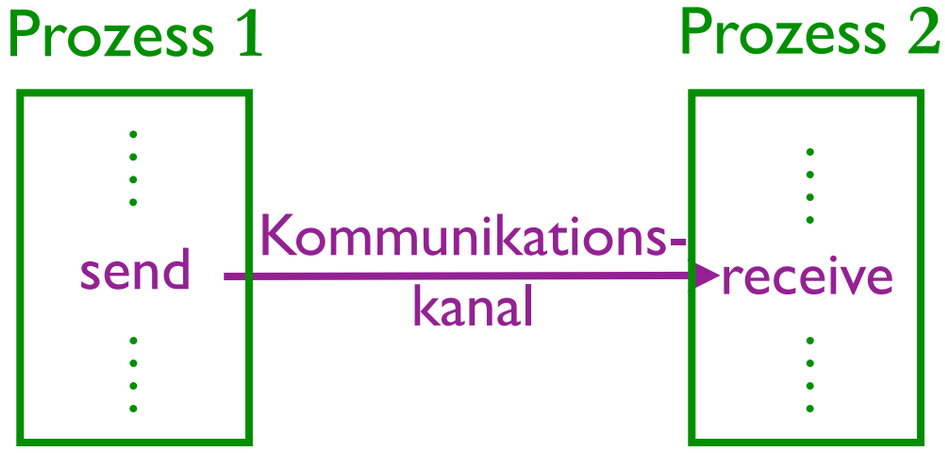
- Davon abstrahieren: Synchronisation/Kommunikation über Nachrichtenaustausch

- Bei Senden Angabe von Ziel und Nachricht

  z.B. `send (Prozess2, Nachricht)`

- Bei Empfangen u.U. Angabe von Quelle und Nachricht (bzw. Puffer, der sie aufnehmen soll)

  z.B. `receive (Prozess1, Puffer)`



- Bei Nachrichtenaustausch findet implizite Synchronisation statt:
  - einseitige Synchronisation implizit:
    Empfangen einer Nachricht erst nach ihrem Senden möglich
  - ggf. gegenseitiger Ausschluss implizit:
    durch Konzept des Kanals statt gemeinsamer Variablen gewährleistet

- Bei Nachrichtenaustausch findet implizite Synchronisation statt:
  - einseitige Synchronisation implizit:
    Empfangen einer Nachricht erst nach ihrem Senden möglich
  - ggf. gegenseitiger Ausschluss implizit:
    durch Konzept des Kanals statt gemeinsamer Variablen gewährleistet

- Jedoch: Nachrichtenaustausch kann auch auf gemeinsamen Variablen simuliert werden:
  - Kommunikationskanal ⇒ gemeinsamer Speicherbereich
  - Senden einer Nachricht ⇒ Schreiben in diesen Speicherbereich
  - Empfangen einer Nachricht ⇒ Lesen aus diesem Speicherbereich

⇒ Dann Schutz des gemeinsamen Speicherbereichs erforderlich
(z.B. durch Lock, Semaphor, Monitor)

- Bei Nachrichtenaustausch findet implizite Synchronisation statt:
  - einseitige Synchronisation implizit:
    Empfangen einer Nachricht erst nach ihrem Senden möglich
  - ggf. gegenseitiger Ausschluss implizit:
    durch Konzept des Kanals statt gemeinsamer Variablen gewährleistet

- Jedoch: Nachrichtenaustausch kann auch auf gemeinsamen Variablen simuliert werden:
  - Kommunikationskanal ⇒ gemeinsamer Speicherbereich
  - Senden einer Nachricht ⇒ Schreiben in diesen Speicherbereich
  - Empfangen einer Nachricht ⇒ Lesen aus diesem Speicherbereich

  ⇒ Dann Schutz des gemeinsamen Speicherbereichs erforderlich
  (z.B. durch Lock, Semaphor, Monitor)

- Nachrichtenaustausch universeller einsetzbar

  ⇒ im allgemeinen Fall u.U. zusätzliche Problembereiche:
  - Einrichten von Kommunikationskanälen
  - Absprachen über Nachrichtenformate
  - …

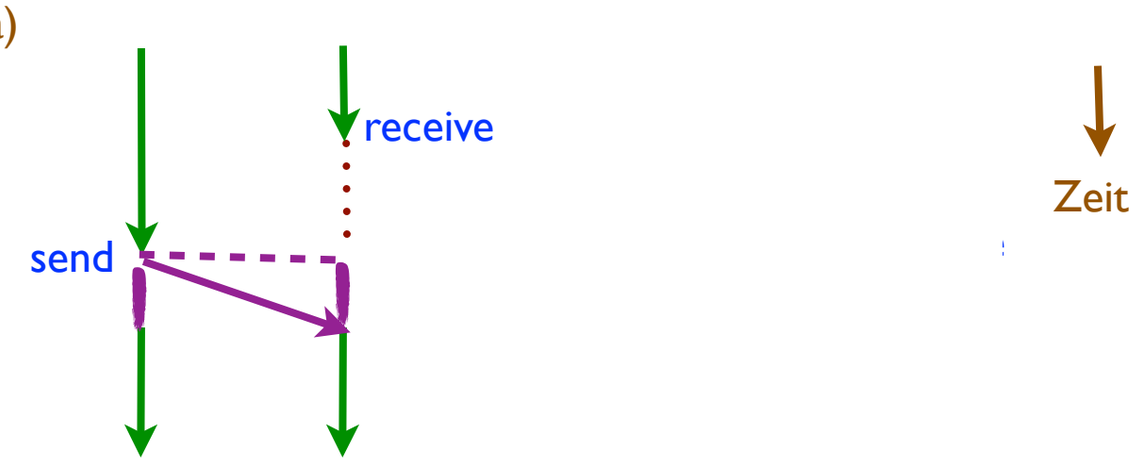
  ⇒ Übergang zu Rechnernetzen

Synchrone vs. asynchrone Kommunikation

Bedeutung der Send-/Receive-Operationen:

- Synchrone Kommunikation:
  - send()/receive() blockierend

    ⇒ Sender/Empfänger warten aufeinander

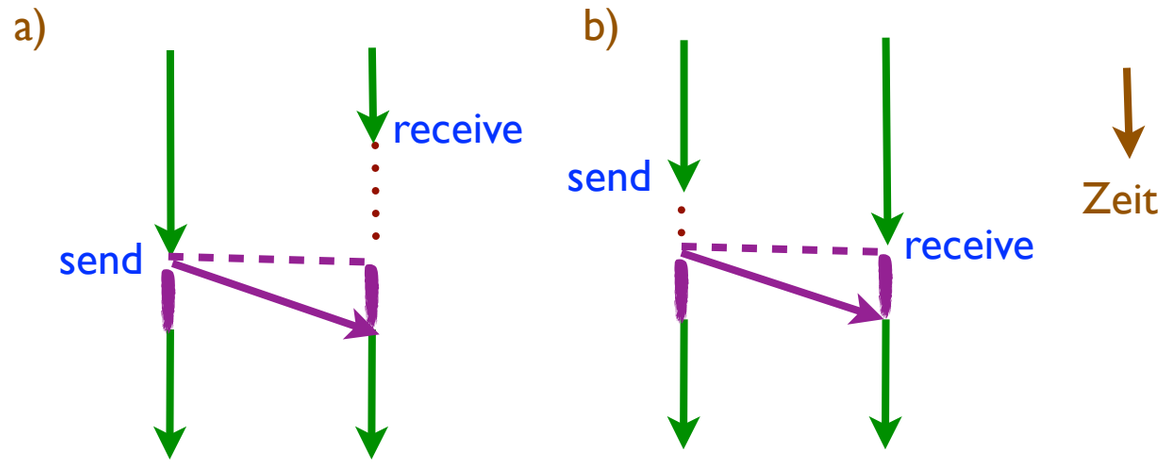


Synchrone vs. asynchrone Kommunikation

Bedeutung der Send-/Receive-Operationen:

- Synchrone Kommunikation:
  - send()/receive() blockierend

    ⇒ Sender/Empfänger warten aufeinander



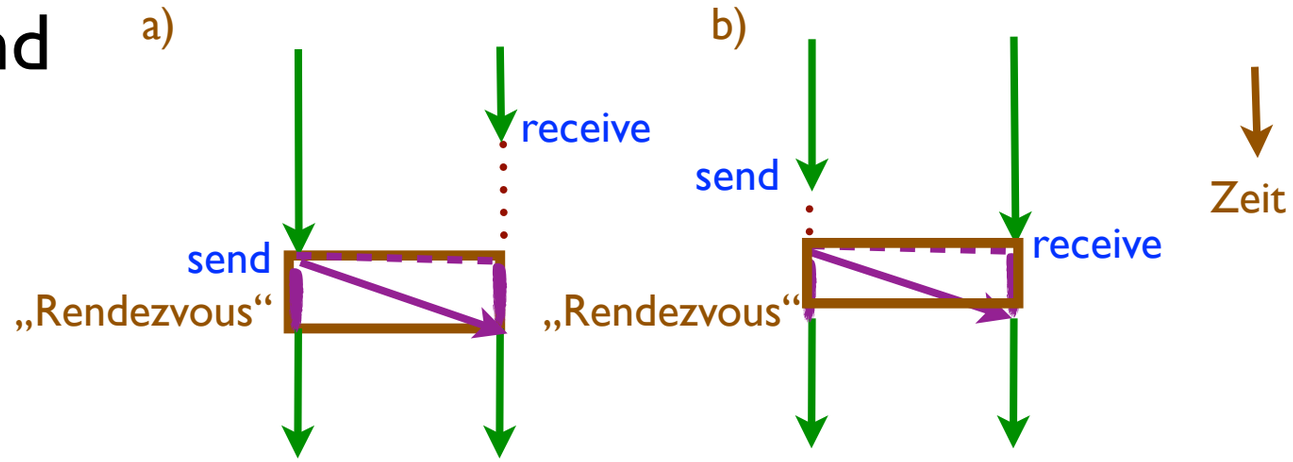
Synchrone vs. asynchrone Kommunikation

Bedeutung der Send-/Receive-Operationen:

- Synchrone Kommunikation:
  - send()/receive() blockierend

    ⇒ Sender/Empfänger warten aufeinander

    ⇒ „Rendezvous“



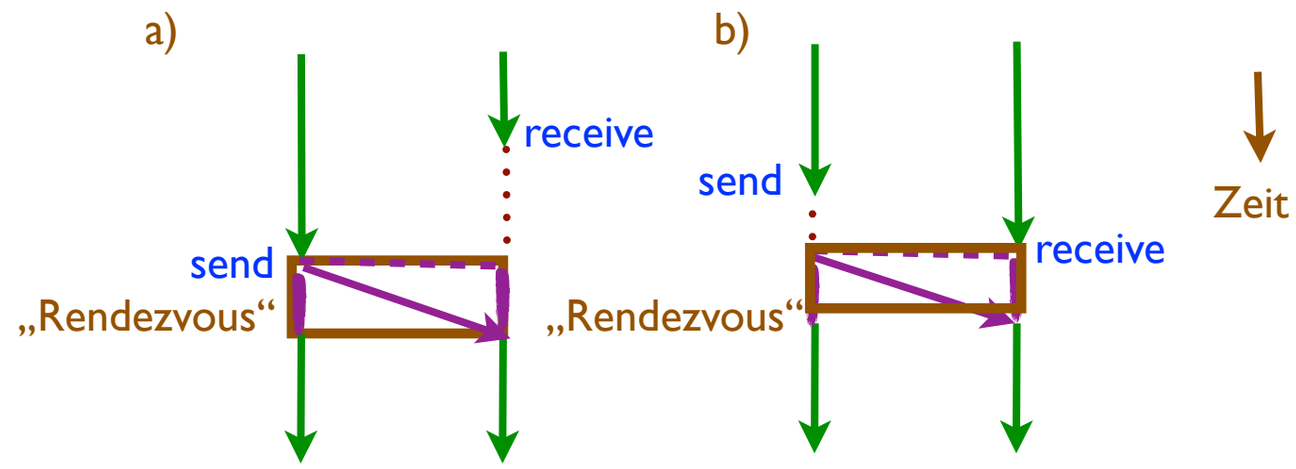
Synchrone vs. asynchrone Kommunikation

Bedeutung der Send-/Receive-Operationen:

- Synchrone Kommunikation:
  - send()/receive() blockierend

    ⇒ Sender/Empfänger warten aufeinander

    ⇒ „Rendezvous“

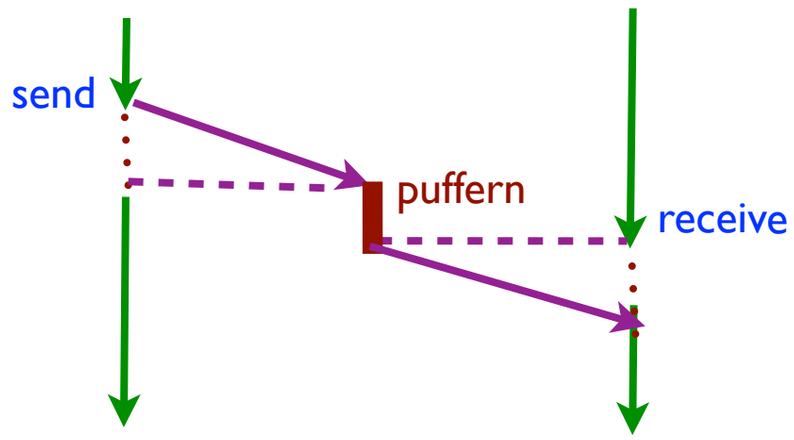


  - Eigentliche Kommunikation dauert i.d.R. Zeit (z.B. Umkopiervorgänge in Empfänger-Adressraum)
  - Synchrone Kommunikation setzt „enge Kopplung“ voraus (Sender muss wissen, wann Empfänger empfangsbereit ist)
    ⇒ in verteilten Systemen i.d.R. nicht ohne weiteres bekannt
  - Nur in Mehrprozessorumgebungen nutzbar, da beide gleichzeitig aktiv
  - Synchrone Kommunikation ist verklemmungsgefährdet, wenn nicht aufeinander abgestimmt

- Asynchrone Kommunikation
  - send() nicht blockierend (receive() i.d.R. schon)

    ⇒ Senden vor Empfangsbereitschaft möglich

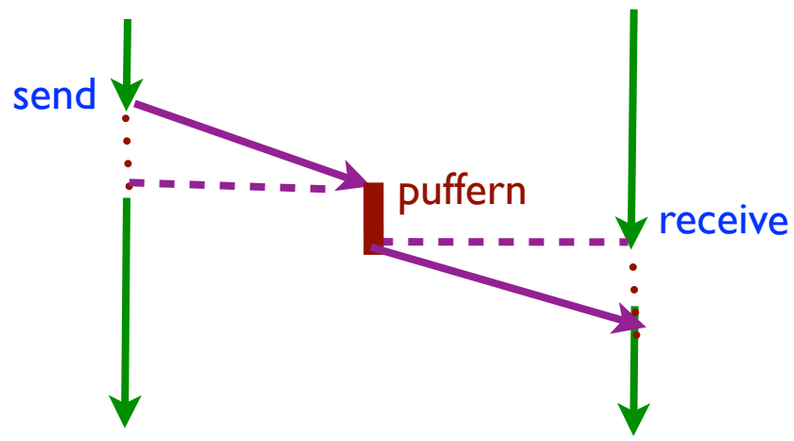
    ⇒ Erfordert Pufferung im Kommunikationskanal



- Asynchrone Kommunikation
  - send() nicht blockierend (receive() i.d.R. schon)

    ⇒ Senden vor Empfangsbereitschaft möglich

    ⇒ Erfordert Pufferung im Kommunikationskanal

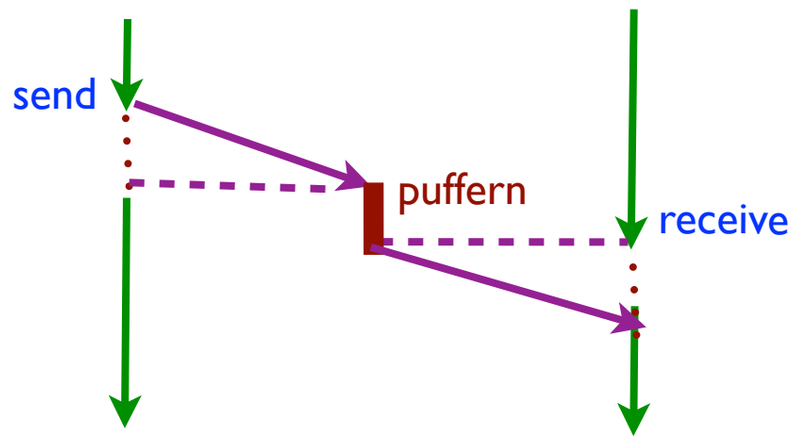


  - Auch „lose Kopplung“ der Prozesse und Einprozessorsystem möglich

- Asynchrone Kommunikation
  - send() nicht blockierend (receive() i.d.R. schon)

    ⇒ Senden vor Empfangsbereitschaft möglich

    ⇒ Erfordert Pufferung im Kommunikationskanal

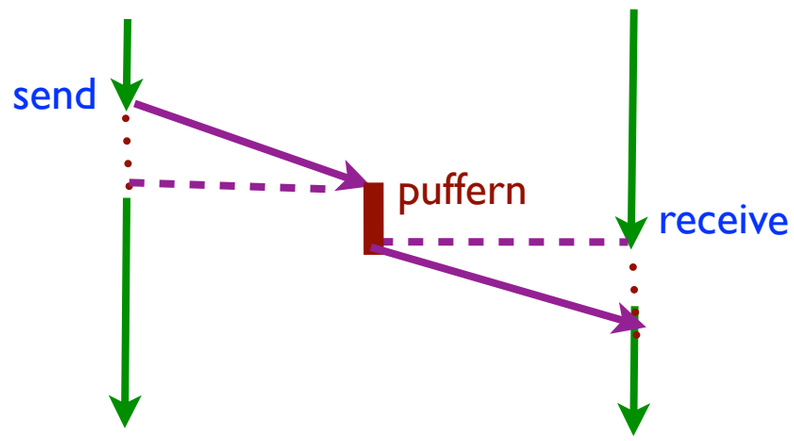


  - Auch „lose Kopplung“ der Prozesse und Einprozessorsystem möglich
  - Bei beschränktem Pufferplatz muss Sender u.U. doch blockiert werden (⇒ Pufferüberlauf)

- Asynchrone Kommunikation
  - send() nicht blockierend (receive() i.d.R. schon)

    ⇒ Senden vor Empfangsbereitschaft möglich

    ⇒ Erfordert Pufferung im Kommunikationskanal

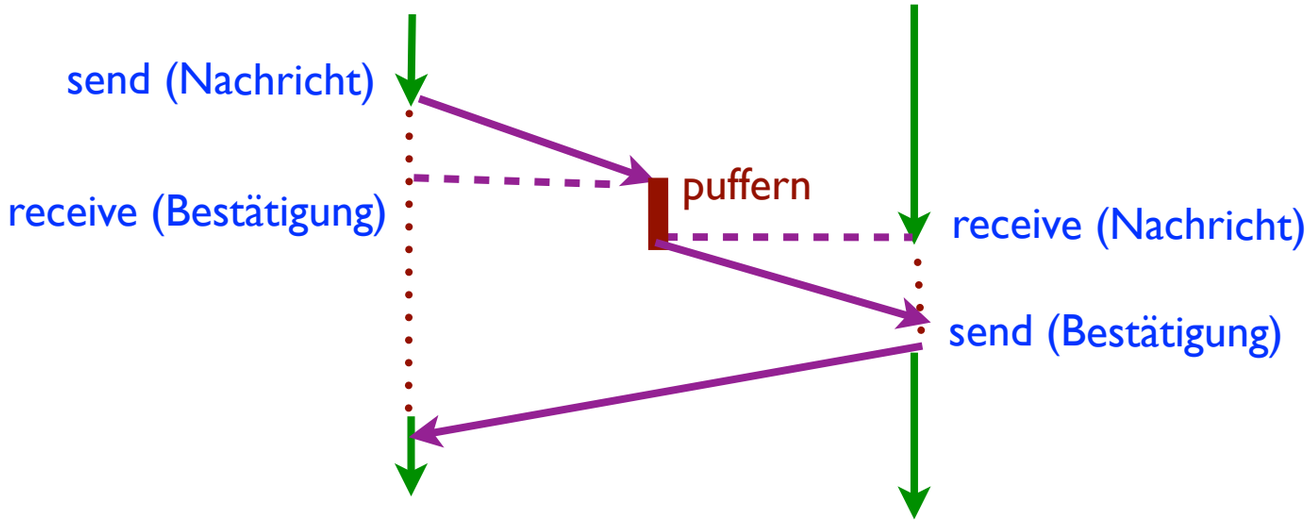


  - Auch „lose Kopplung“ der Prozesse und Einprozessorsystem möglich
  - Bei beschränktem Pufferplatz muss Sender u.U. doch blockiert werden (⇒ Pufferüberlauf)
  - Alternative:
    - send() kehrt bei Pufferüberlauf mit Fehlermeldung zurück
    - Entsprechend: receive() kehrt bei leerem Puffer mit Fehlermeldung zurück

Dualität der synchronen/asynchronen Kommunikation

- Synchrone Kommunikation kann durch asynchrone Operationen vorgenommen werden

⇒ Sender wartet nach Senden auf Bestätigung

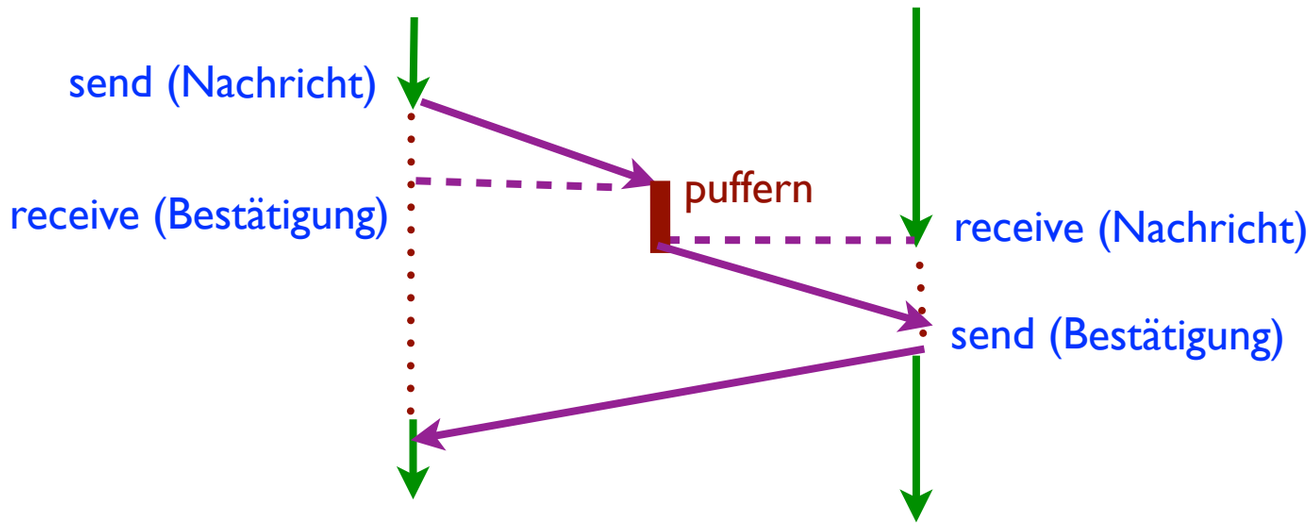


  - Reine Empfangsbestätigung vs. „Antwort“
  - Gebräuchliches Verhalten in Rechnernetzen

Dualität der synchronen/asynchronen Kommunikation

- Synchrone Kommunikation kann durch asynchrone Operationen vorgenommen werden

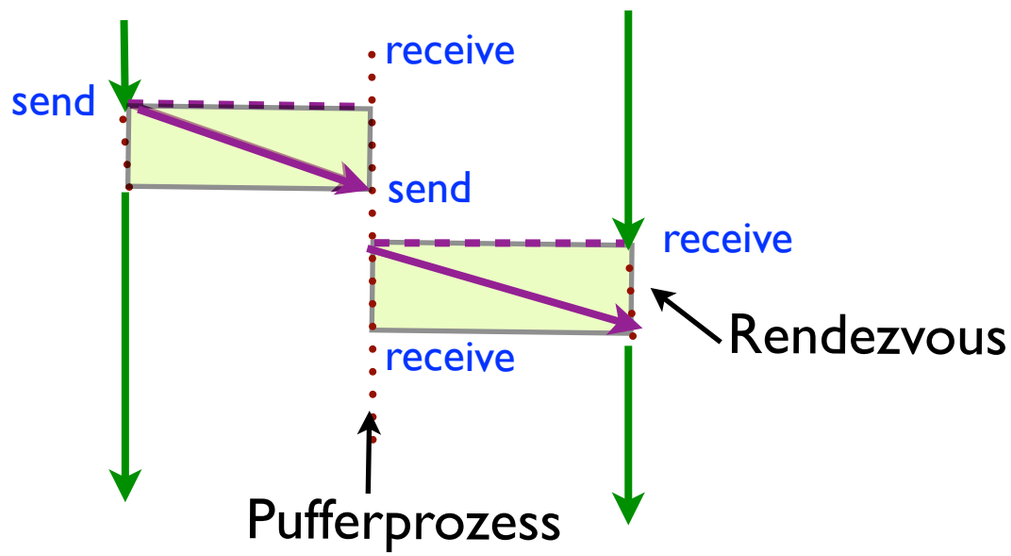
⇒ Sender wartet nach Senden auf Bestätigung



  - Reine Empfangsbestätigung vs. „Antwort“
  - Gebräuchliches Verhalten in Rechnernetzen

- Asynchrone Kommunikation kann durch synchrone Operationen vorgenommen werden

⇒ Einführung eines Pufferprozesses



Modellierung der Kommunikationsbeziehung über Semaphore

- Innerhalb eines Systems möglich ⇒ ähnlich zu einseitiger Synchronisation
- Achtung vereinfacht: send()/receive() als unteilbar angenommen (unter Nutzung von Puffer)

- Asynchrone Kommunikation:

```
Sema s1(0);

Prozess1                        Prozess2
...                             ...
send(Nachricht);                s1.P();
s1.V();                         receive(Nachricht);
```

Modellierung der Kommunikationsbeziehung über Semaphore

- Innerhalb eines Systems möglich ⇒ ähnlich zu einseitiger Synchronisation
- Achtung vereinfacht: send()/receive() als unteilbar angenommen (unter Nutzung von Puffer)

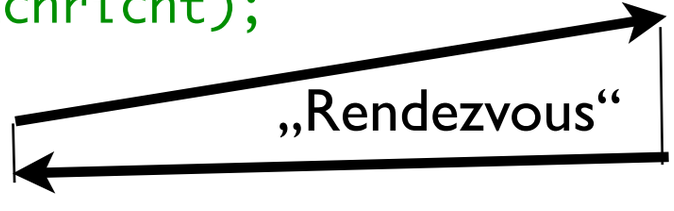
- **Asynchrone Kommunikation:**

```
Sema s1(0);

Prozess1                        Prozess2
...                             ...
send(Nachricht);                s1.P();
s1.V();                         receive(Nachricht);
```

- **Synchrone Kommunikation:**

  ⇒ nicht genau modellierbar, aber ähnliche Semantik zu Bestätigungsnachricht

```
Sema s1(0);
Sema s2(0);

Prozess1                        Prozess2
...                             ...
send(Nachricht);                s1.P();
s1.V();                         receive(Nachricht);
s2.P();                         s2.V();
```

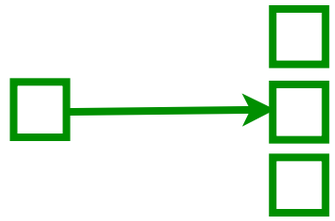


Kleine Aufgabe

Unter welchen Umständen ist das Umsteigen zwischen Straßenbahnen am Hauptbahnhof eher als *synchron* bzw. eher als *asynchron* zu bezeichnen?

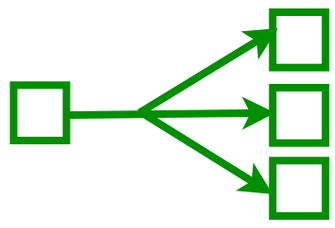
Bemerkungen:

- Ein Prozess kann u.U. mit mehreren anderen kommunizieren
  - 1:1-Beziehungen (Unicast)



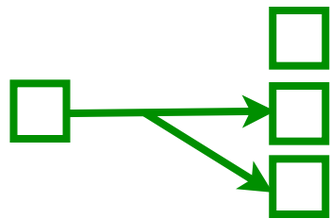
  - 1:n-Beziehungen (Broadcast)

    ⇒ eine Nachricht „an alle“

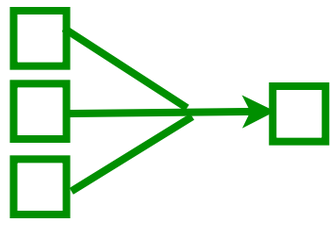


  - 1:k-Beziehungen (Multicast)

    ⇒ eine Nachricht
    „an mehrere angegebene“



  - n:1-Beziehungen



  - n:m-Beziehungen

- Setzt Benennung des/der Partner(s) voraus
  - Prozessbezeichner

    ⇒ genaue Auflistung erforderlich

  - Besser: Kanalbezeichner

    ⇒ verschiedene Kanäle unterscheidbar

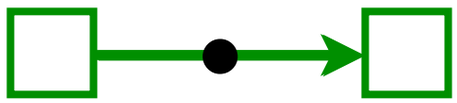
    ⇒ Partner können dennoch anonym bleiben

  - Falls nur ein Sender oder ein Empfänger auf Kanal: Port-Bezeichner

    ⇒ dennoch Entkopplung von Prozess-ID

    (dynamische Zuordnung möglich)

- Ansiedlung des Pufferbereichs bei asynchroner Kommunikation:
  - im Kanal (z.B. Unix Pipes ⇒ Kern)



  - beim Empfänger (klassische „Mailbox“)

    ⇒ Sender legen Nachrichten dort ab (z.B. Aufträge)

  - beim Sender („Ausgangskorb“)

    ⇒ Empfänger holen Nachrichten dort ab

Fragen – Teil 1

- Was versteht man unter *synchronem* bzw. *asynchronem* Nachrichtenaustausch? Inwiefern sind diese beiden Kommunikationsformen aufeinander abbildbar?
- Wie kann man die Synchronisationseigenschaften von synchronem bzw. asynchronem Nachrichtenaustausch mit Semaphoren modellieren?
- Wozu verwendet man *Kanäle* bzw. *Ports*? Was ist das?

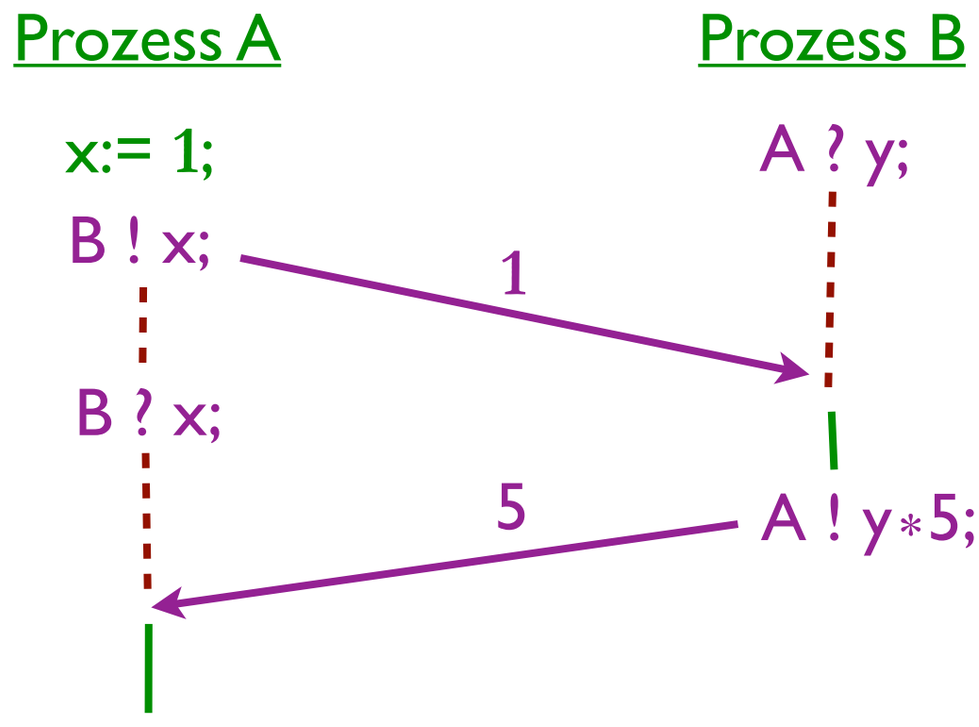
Teil 2:
Communicating Sequential Processes (CSP)

CSP (Communicating Sequential Processes)
Hoare 1978

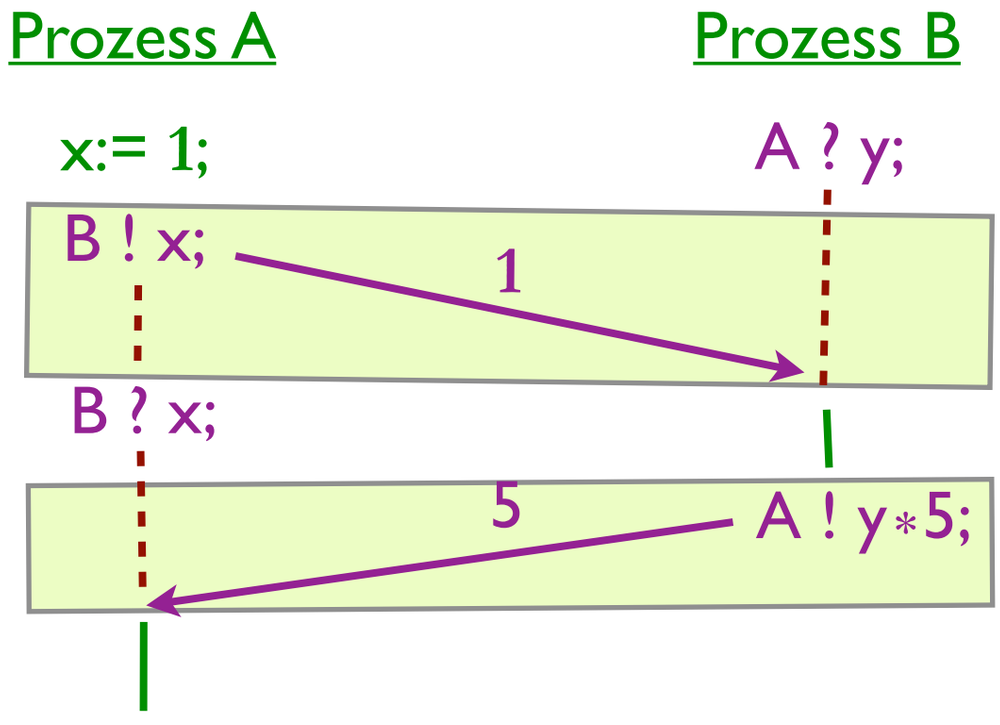
- Methode zum Beschreiben von Algorithmen mit synchronem Nachrichtenaustausch
- Programmiersprache mit „Lücken“
  (wurde später z.B. in OCCAM ergänzt)
- Wesentliche Konzepte dann in Erlang aufgegriffen
- Im folgenden Originalsyntax/-semantik verwendet

  ⇒ gibt auch Varianten/Weiterentwicklungen

- Kommunikationsanweisungen
  - synchron
  - Ausgabeanweisung: `recipient ! message` (≙ send(recipient, message))
    - An angegebenen Empfangsprozess (nicht Kanal/Port) soll Nachricht message gesendet werden
  - Eingabeanweisung: `sender ? variable` (≙ receive(sender, variable))
    - Von angegebenem Sendeprozess soll Nachricht empfangen und in variable abgelegt werden



- Kommunikationsanweisungen
  - synchron
  - Ausgabeanweisung: `recipient ! message` (≙ send(recipient, message))
    - An angegebenen Empfangsprozess (nicht Kanal/Port) soll Nachricht message gesendet werden
  - Eingabeanweisung: `sender ? variable` (≙ receive(sender, variable))
    - Von angegebenem Sendeprozess soll Nachricht empfangen und in variable abgelegt werden



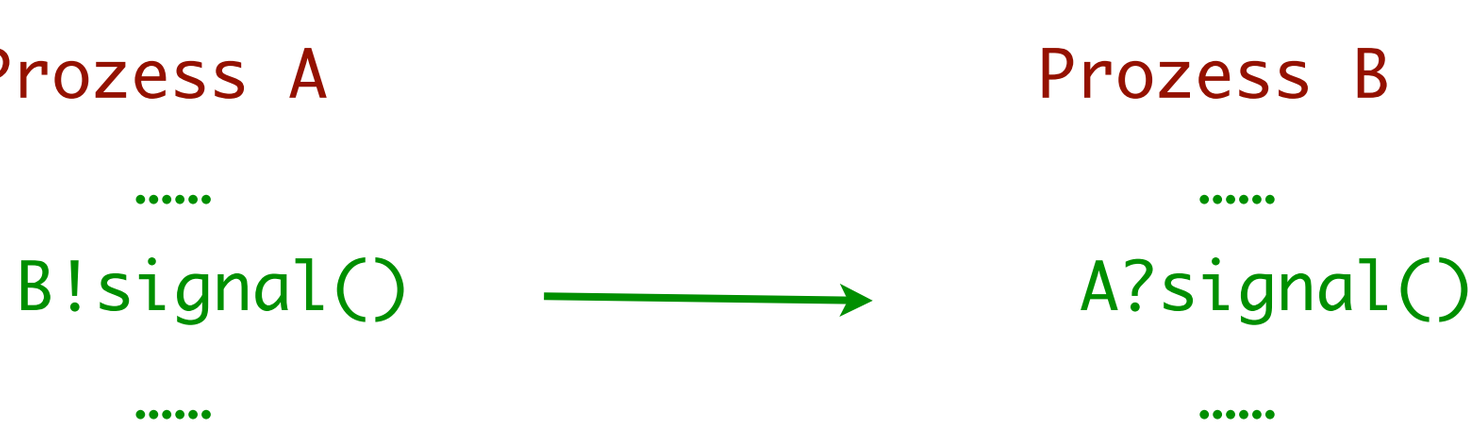
2 Rendezvous

- Nachrichten/Variablen können strukturiert sein:

  ```
  a:= auftrag(datum,nummer,artikel)
  ```

- Strukturierte Variablen können komponentenlos sein

  ⇒ dienen im Nachrichtenaustausch zur
  Signalisierung von Ereignissen



- Nachrichten/Variablen können strukturiert sein:

  ```
  a:= auftrag(datum,nummer,artikel)
  ```

- Strukturierte Variablen können komponentenlos sein

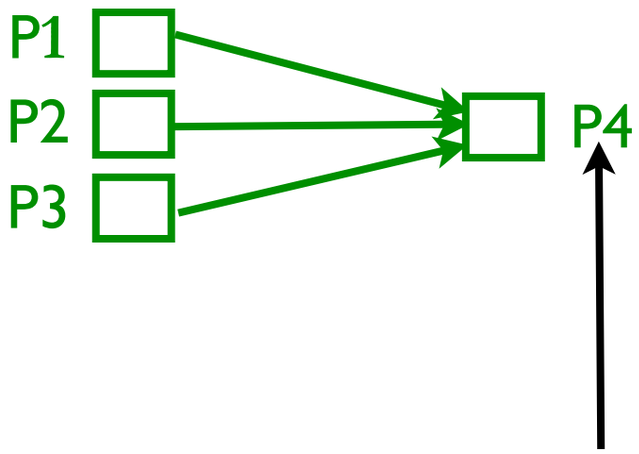
  ⇒ dienen im Nachrichtenaustausch zur
  Signalisierung von Ereignissen

```
Prozess A                          Prozess B
… erzeugen …                       ……
B!signal()        ──────────►      A?signal()
……                                 … verbrauchen …
```

Zur Erinnerung: Ereignisvariablen

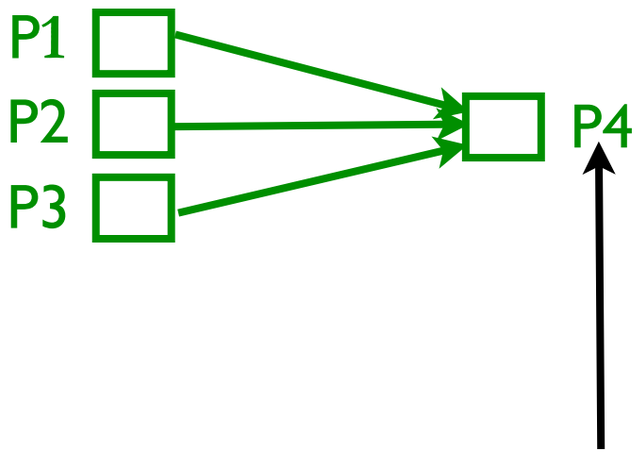
```
                Condition c;
… erzeugen …                       c.wait();
c.signal();                        … verbrauchen …
```

- Alternativ-Anweisung
  - sehr wichtiges Konzept von CSP
  - ermöglicht es z.B., wahlweise auf mehrere mögliche Nachrichten zu warten (n:1-Kommunikation)



⇒ welcher Prozess wird (zuerst) senden?

- Alternativ-Anweisung
  - sehr wichtiges Konzept von CSP
  - ermöglicht es z.B., wahlweise auf mehrere mögliche Nachrichten zu warten (n:1-Kommunikation)



⇒ welcher Prozess wird (zuerst) senden?

- basiert auf Konzept der bewachten Kommandos (guarded commands, Dijkstra 1975)

```
[ guard1 → anweisung1
[] guard2 → anweisung2
[] guard3 → anweisung3
]
```

guard: Bedingung
[]: „oder wenn“

- Beispiel: Auf mehrere Eingaben warten:

```
[ P1 ? variable1 → anweisung1
[] P2 ? variable2 → anweisung2
[] P3 ? variable3 → anweisung3
]
```

- Empfangsbedingung trifft zu, wenn Nachricht empfangen wurde

- Beispiel: Auf mehrere Eingaben warten:

```
[ P1 ? variable1 → anweisung1
[] P2 ? variable2 → anweisung2
[] P3 ? variable3 → anweisung3
]
```

Beispiel: Nachricht von P1 und P2
⇒ anweisung1 oder anweisung2

- Empfangsbedingung trifft zu, wenn Nachricht empfangen wurde
- Treffen mehrere Bedingungen zu, ist Auswahl nicht-deterministisch:

- Beispiel: Auf mehrere Eingaben warten:

```
[ P1 ? variable1 → anweisung1
[] P2 ? variable2 → anweisung2
[] P3 ? variable3 → anweisung3
]
```

Beispiel: Nachricht von P1 und P2
⇒ anweisung1 oder anweisung2

- Empfangsbedingung trifft zu, wenn Nachricht empfangen wurde
- Treffen mehrere Bedingungen zu, ist Auswahl nicht-deterministisch:
  - Beispiel: Maximalwert

```
[ a<=b → max:=b
[] a>=b → max:=a
]
```

- Beispiel: Absolutwert

```
[ x<=0 → x:= -x
[] x>=0 → skip
]
```

skip: Leere Anweisung

- Beispiel: Auf mehrere Eingaben warten:

```
[ P1 ? variable1 → anweisung1
[] P2 ? variable2 → anweisung2
[] P3 ? variable3 → anweisung3
]
```

Beispiel: Nachricht von P1 und P2
⇒ anweisung1 oder anweisung2

- Empfangsbedingung trifft zu, wenn Nachricht empfangen wurde
- Treffen mehrere Bedingungen zu, ist Auswahl nicht-deterministisch:
  - Beispiel: Maximalwert

```
[ a<=b → max:=b
[] a>=b → max:=a
]
```

  - Beispiel: Absolutwert

```
[ x<=0 → x:= -x
[] x>=0 → skip
]
```

skip: Leere Anweisung

- Kurzschreibweise: Indizierung

```
[ (i:1..n) guard(i) → anweisung(i) ]
```

- While-Schleife (z.B. Kopieren von Nachrichten):

```
*[ P1 ? nachricht →  P2 ! nachricht ]
```

* Wiederholung

- Prozesse in CSP
  - Werden in CSP als nebenläufige Blöcke notiert
  - Jeder Prozess hat eindeutigen Namen
  - Abkürzende Schreibweise: Indizierung von gleichartigen Prozessen

```
[ druckserver:: anweisungen
|| druckclient(i:1..N):: anweisungen
]
```

Alternativanweisung:

```
[ . . . → . . .
[] . . . → . . .
]
```

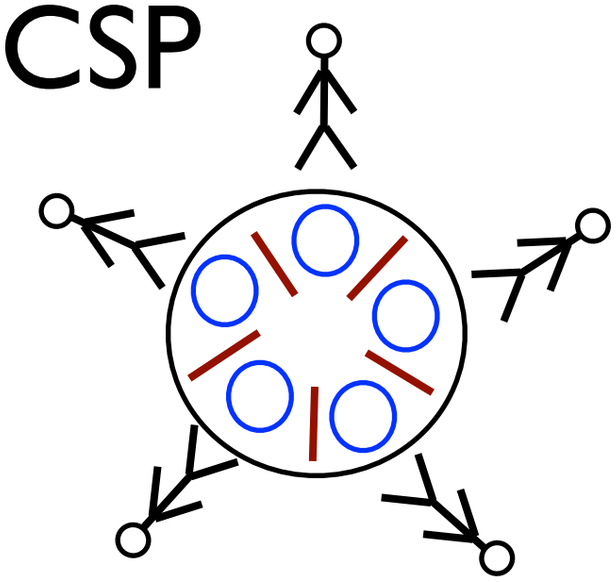
- Prozesse in CSP
  - Werden in CSP als nebenläufige Blöcke notiert
  - Jeder Prozess hat eindeutigen Namen
  - Abkürzende Schreibweise: Indizierung von gleichartigen Prozessen

```
[ druckserver:: anweisungen
|| druckclient(i:1..N):: anweisungen
]
```

  - Prozess terminiert, wenn
    - alle Anweisungen abgearbeitet sind
    - wenn Kommunikationsanweisung scheitert (Partnerprozesse sind bereits terminiert)
  - Entsprechend: Schleife/guarded command terminiert/scheitert, wenn Bedingungen(en) nicht mehr erfüllt werden können

Die speisenden Philosophen in CSP

```
[ philosoph(i:0..4):: PHIL]
```
(denken und speisen)



Die speisenden Philosophen in CSP

`[ philosoph(i:0..4):: PHIL]` (denken und speisen)

Wdh.: Mögliche Semaphor-Lösung

```
Sema stick[5];   //Array von 5 Semaphoren, mit 1 initialisiert
Sema chair(4);   //mit 4 initialisiertes Semaphor

philosoph(int i) {
  while (1) {
    ... denken ...
    chair.P();
    stick[i].P();
    stick[(i+1)%5].P();
    ... speisen ...
    stick[i].V();
    stick[(i+1)%5].V();
    chair.V();
  };
}
```

Die speisenden Philosophen in CSP

```
[ philosoph(i:0..4):: PHIL    (denken und speisen)
|| stick(i:0..4):: STICK      (nur einer kann Stäbchen haben)
|| table:: TABLE              (nur 4 Philosophen am Tisch)
]
```

⇒ spezielle „Synchronisationsprozesse“

Die speisenden Philosophen in CSP

```
[ philosoph(i:0..4):: PHIL    (denken und speisen)
|| stick(i:0..4):: STICK      (nur einer kann Stäbchen haben)
|| table:: TABLE              (nur 4 Philosophen am Tisch)
]
```

⇒ spezielle „Synchronisationsprozesse“

```
PHIL=
*[ true → ... denken ...
          table ! setzen();
          stick(i) ! aufnehmen();
          stick((i+1)mod5) ! aufnehmen();
          ... speisen ...
          stick(i) ! niederlegen();
          stick((i+1)mod5) ! niederlegen();
          table ! aufstehen();
]
```

Zur Erinnerung: Synchroner Nachrichtenaustausch
⇒ sonst würde diese Lösung nicht funktionieren

Die speisenden Philosophen in CSP

```
[ philosoph(i:0..4):: PHIL
|| stick(i:0..4):: STICK
|| table:: TABLE
]
```

⇒ spezielle „Synchronisationsprozesse“

```
PHIL=
*[ true → ... denken ...
      table ! setzen();
      stick(i) ! aufnehmen();
      stick((i+1)mod5) ! aufnehmen();
      ... speisen ...
      stick(i) ! niederlegen();
      stick((i+1)mod5) ! niederlegen();
      table ! aufstehen();
]
```

```
STICK=
*[ philosoph(i) ? aufnehmen()
         → philosoph(i) ? niederlegen()
 [] philosoph((i-1)mod5) ? aufnehmen()
         → philosoph((i-1)mod5) ? niederlegen()
]
```

Die speisenden Philosophen in CSP

```
[ philosoph(i:0..4):: PHIL
|| stick(i:0..4):: STICK
|| table:: TABLE
]
```

⇒ spezielle „Synchronisationsprozesse“

```
PHIL=
*[ true → ... denken ...
      table ! setzen();
      stick(i) ! aufnehmen();
      stick((i+1)mod5) ! aufnehmen();
       ... speisen ...
      stick(i) ! niederlegen();
      stick((i+1)mod5) ! niederlegen();
      table ! aufstehen();
]
```

```
STICK=
*[ philosoph(i) ? aufnehmen()
          → philosoph(i) ? niederlegen()
 [] philosoph((i-1)mod5) ? aufnehmen()
          → philosoph((i-1)mod5) ? niederlegen()
]
```

```
TABLE=
gäste: integer;
gäste:= 0;
*[ (i:0..4) gäste<4; philosoph(i) ? setzen()
           → gäste:= gäste+1
 [] (i:0..4) philosoph(i) ? aufstehen()
           → gäste:= gäste-1
]
```

Fragen – Teil 2

- Was ist ein *guarded command*? Warum kann die Verwendung eines solchen Konzepts gerade in Zusammenhang mit Nachrichtenaustauschvorgängen interessant sein?

Teil 3:
Das Sieb des Eratosthenes in CSP

Das Sieb des Eratosthenes

- Bekannter Algorithmus zum Bestimmen von Primzahlen in aufsteigender Reihenfolge
- Ausgehend von bereits gefundener Primzahl p werden ihre Vielfache „ausgesiebt“ (beginnend bei p^2).

  2 3 4 5 6 7 8 9 10 11 12 13 ...

Das Sieb des Eratosthenes

- Bekannter Algorithmus zum Bestimmen von Primzahlen in aufsteigender Reihenfolge
- Ausgehend von bereits gefundener Primzahl p werden ihre Vielfache „ausgesiebt“ (beginnend bei p^2).

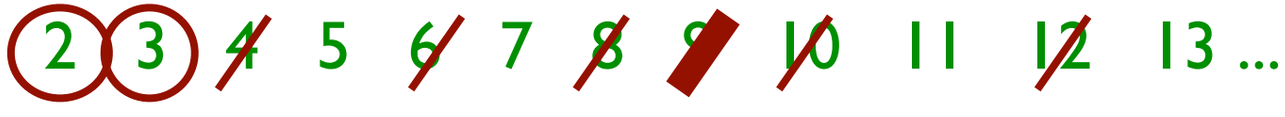
2 3 ~~4~~ 5 ~~6~~ 7 ~~8~~ 9 ~~10~~ 11 ~~12~~ 13 ...

- Optimierung: 2 als Sonderfall betrachten

⇒ nur ungerade Zahlen untersuchen

Das Sieb des Eratosthenes

- Bekannter Algorithmus zum Bestimmen von Primzahlen in aufsteigender Reihenfolge
- Ausgehend von bereits gefundener Primzahl p werden ihre Vielfache „ausgesiebt“ (beginnend bei p^2).



- Optimierung: 2 als Sonderfall betrachten

⇒ nur ungerade Zahlen untersuchen

Das Sieb des Eratosthenes

- Bekannter Algorithmus zum Bestimmen von Primzahlen in aufsteigender Reihenfolge
- Ausgehend von bereits gefundener Primzahl p werden ihre Vielfache „ausgesiebt“ (beginnend bei p^2).

2 3 ~~4~~ 5 ~~6~~ 7 ~~8~~ ~~9~~ ~~10~~ 11 ~~12~~ 13 ...

- Optimierung: 2 als Sonderfall betrachten

⇒ nur ungerade Zahlen untersuchen

Das Sieb des Eratosthenes

- Bekannter Algorithmus zum Bestimmen von Primzahlen in aufsteigender Reihenfolge
- Ausgehend von bereits gefundener Primzahl p werden ihre Vielfache „ausgesiebt“ (beginnend bei p^2).

2 3 ~~4~~ 5 ~~6~~ 7 ~~8~~ ~~9~~ ~~10~~ 11 ~~12~~ 13 ...

- Optimierung: 2 als Sonderfall betrachten

  ⇒ nur ungerade Zahlen untersuchen

- Endekriterium: $p \geq \sqrt{MaxNum}$ (für Primzahlen bis MaxNum-1)

  ⇒ keine Vielfachen mehr aussiebbar

  ⇒ verbleibende Zahlen müssen Primzahlen sein

- 2 Lösungen betrachten:
  - sequentielle Lösung (in C++)
  - nebenläufige Lösung (in CSP)

Das Sieb des Eratosthenes (sequentielle Variante)

```
#include <iostream.h>
const int max_wurzel= 100;
const int max_nummer= max_wurzel * max_wurzel;
bool sieb[max_nummer];     // sieb[i]= true ⇒ i könnte Primzahl sein

sieb_initialisieren() {
  int i;
  for (i=0; i<max_nummer; i++)
    sieb[i]= i&1;          // nur ungerade Zahlen
  sieb[2]= true;           //Sonderfall 2
}

aussieben() {
  for (int p=3; p<max_wurzel; p+=2) {
    if (sieb[p]) {         // p ist Primzahl?
      for (int mp= p*p; mp<max_nummer; mp+=2*p) {
        sieb[mp]= false;   // Vielfache von p aussieben
      }
    }
  }
}

sieb_ausgeben() {
  for (int p=2; p<max_nummer; p++)
    if (sieb[p])
      cout << p << endl;
}

main() {
  sieb_initialisieren();
  aussieben();
  sieb_ausgeben();
}
```

Das Sieb des Eratosthenes (sequentielle Variante)

```
#include <iostream.h>
const int max_wurzel= 100;
const int max_nummer= max_wurzel * max_wurzel;
bool sieb[max_nummer];      // sieb[i]= true ⇒ i könnte Primzahl sein

sieb_initialisieren() {
  int i;
  for (i=0; i<max_nummer; i++)
    sieb[i]= i&1;           // nur ungerade Zahlen
  sieb[2]= true;            //Sonderfall 2
}

aussieben() {
  for (int p=3; p<max_wurzel; p+=2) {
    if (sieb[p]) {          // p ist Primzahl?
      for (int mp= p*p; mp<max_nummer; mp+=2*p) {
        sieb[mp]= false;    // Vielfache von p aussieben
      }
    }
  }
}

sieb_ausgeben() {
  for (int p=2; p<max_nummer; p++)
    if (sieb[p])
      cout << p << endl;
}

main() {
  sieb_initialisieren();
  aussieben();
  sieb_ausgeben();
}
```

```
  10110
& 00001
-------
  00000
```

Das Sieb des Eratosthenes (sequentielle Variante)

```
#include <iostream.h>
const int max_wurzel= 100;
const int max_nummer= max_wurzel * max_wurzel;
bool sieb[max_nummer];     // sieb[i]= true ⇒ i könnte Primzahl sein

sieb_initialisieren() {
  int i;
  for (i=0; i<max_nummer; i++)
    sieb[i]= i&1;          // nur ungerade Zahlen
  sieb[2]= true;           //Sonderfall 2
}
```

```
  aussieben() {
    for (int p=3; p<max_wurzel; p+=2) {
      if (sieb[p]) {          // p ist Primzahl?
        for (int mp= p*p; mp<max_nummer; mp+=2*p) {
          sieb[mp]= false;   // Vielfache von p aussieben
        }
      }
    }
  }
```

```
main() {
  sieb_initialisieren();
  aussieben();
  sieb_ausgeben();
}
```

Das Sieb des Eratosthenes (sequentielle Variante)

```
#include <iostream.h>
const int max_wurzel= 100;
const int max_nummer= max_wurzel * max_wurzel;
bool sieb[max_nummer];     // sieb[i]= true ⇒ i könnte Primzahl sein

sieb_initialisieren() {
  int i;
  for (i=0; i<max_nummer; i++)
    sieb[i]= i&1;          // nur ungerade Zahlen
  sieb[2]= true;           //Sonderfall 2
}

aussieben() {
  for (int p=3; p<max_wurzel; p+=2) {
    if (sieb[p]) {         // p ist Primzahl?
      for (int mp= p*p; mp<max_nummer; mp+=2*p) {
        sieb[mp]= false;   // Vielfache von p aussieben
      }
    }
  }
}

sieb_ausgeben() {
  for (int p=2; p<max_nummer; p++)
    if (sieb[p])
      cout << p << endl;
}

main() {
  sieb_initialisieren();
  aussieben();
  sieb_ausgeben();
}
```

Das Sieb des Eratosthenes in CSP

- Erhöhung möglicher Nebenläufigkeit
- Vorsehen mehrerer Prozesse („Siebe“)

  ⇒ Kaskadierung

- Jeder kümmert sich um Vielfache einer Primzahl
- Kommunikation zwischen „Sieben“ über Nachrichtenaustausch (in CSP: synchroner Nachrichtenaustausch)

  ⇒ Mitteilung der nächsten zu untersuchenden Zahl

  an das nächste Sieb

```
 Sieb0          Sieb1
(Vielfache 2)  (Vielfache 3)

 Drucken 2
     3 ------> Drucken 3
               mp= 3²= 9
```

```
 Sieb0           Sieb1           Sieb2
(Vielfache 2)  (Vielfache 3)  (Vielfache 5)

 Drucken 2
     3 ------> Drucken 3
               mp= 3²= 9

     5 ------> 5 < 9  ------> Drucken 5
                              mp= 5²= 25
```

```
 Sieb0          Sieb1          Sieb2          Sieb3
(Vielfache 2)  (Vielfache 3)  (Vielfache 5)  (Vielfache 7)

 Drucken 2
     3 ------> Drucken 3
               mp= 3²= 9

     5 ------> 5 < 9  ------> Drucken 5
                              mp= 5²= 25

     7 ------> 7 < 9  ------> 7 < 25 ------> Drucken 7
                                             mp= 7²= 49
```

```
 Sieb0           Sieb1           Sieb2           Sieb3
(Vielfache 2)  (Vielfache 3)  (Vielfache 5)  (Vielfache 7)

 Drucken 2
     3 ------> Drucken 3
               mp= 3²= 9

     5 ------> 5 < 9  ------> Drucken 5
                              mp= 5²= 25

     7 ------> 7 < 9  ------> 7 < 25 ------> Drucken 7
                                             mp= 7²= 49

 9 ------> 9 = 9
           aussieben
```

```
 Sieb0          Sieb1          Sieb2          Sieb3          Sieb4
(Vielfache 2)  (Vielfache 3)  (Vielfache 5)  (Vielfache 7)   (Vielfache 11)

 Drucken 2
     3 ------> Drucken 3
               mp= 3²= 9

     5 ------> 5 < 9  ------> Drucken 5
                              mp= 5²= 25

     7 ------> 7 < 9  ------> 7 < 25 ------> Drucken 7
                                             mp= 7²= 49

  9 ------> 9 = 9
            aussieben

 11 ------> 11 > 9
            mp = 9 + 2*3 = 15
            11 < 15 -----> 11 < 25 -----> 11 < 49  ------> Drucken 11
                                                           mp= 11²= 121
```

```
 Sieb0          Sieb1          Sieb2          Sieb3          Sieb4
(Vielfache 2)  (Vielfache 3)  (Vielfache 5)  (Vielfache 7)   (Vielfache 11)

 Drucken 2
     3 ------> Drucken 3
               mp= 3²= 9

     5 ------> 5 < 9  ------> Drucken 5
                              mp= 5²= 25

     7 ------> 7 < 9  ------> 7 < 25 ------> Drucken 7
                                             mp= 7²= 49

  9 ------> 9 = 9
            aussieben

 11 ------> 11 > 9
            mp = 9 + 2*3 = 15
            11 < 15 -----> 11 < 25 -----> 11 < 49  ------> Drucken 11
                                                           mp= 11²= 121

 13 ------> 13 < 15 -----> 13 < 25 -----> 13 < 49  ------> 13 < 121 ----->
```

```
 Sieb0          Sieb1          Sieb2          Sieb3          Sieb4
(Vielfache 2)  (Vielfache 3)  (Vielfache 5)  (Vielfache 7)   (Vielfache 11)

 Drucken 2
     3 ------> Drucken 3
               mp= 3²= 9

     5 ------> 5 < 9  ------> Drucken 5
                              mp= 5²= 25

     7 ------> 7 < 9  ------> 7 < 25 ------> Drucken 7
                                             mp= 7²= 49

     9 ------> 9 = 9
               aussieben

    11 ------> 11 > 9
               mp = 9 + 2*3 = 15
               11 < 15 -----> 11 < 25 -----> 11 < 49  ------> Drucken 11
                                                              mp= 11²= 121

    13 ------> 13 < 15 -----> 13 < 25 -----> 13 < 49  ------> 13 < 121 ----->

    15 ------> 15 = 15
               aussieben
```

Das Sieb des Eratosthenes in CSP

```
sieb(0)::
  n: integer;
  drucken ! 2;
  n:= 3;
 *[ n<10000 → sieb(1) ! n; n:= n+2 ]

sieb(i:1..49)::
  p, mp: integer;
  sieb(i-1) ? p;
  drucken ! p;
  mp:= p*p;
 *[ m: integer;
     sieb(i-1) ? m; →
   *[ m>mp → mp:= mp + 2*p ]
    [ m=mp → skip
    [] m<mp → sieb(i+1) ! m
    ]
 ]

sieb(50)::
  n: integer;
 *[ sieb(49) ? n →  drucken ! n ]

drucken::
  n: integer;
 *[ (i:0..50) sieb(i) ? n →  …  drucken … ]
```

Das Sieb des Eratosthenes in CSP

```
sieb(0)::
   n: integer;
   drucken ! 2;
   n:= 3;
 *[ n<10000 → sieb(1) ! n; n:= n+2 ]


sieb(i:1..49)::
  p, mp: integer;
  sieb(i-1) ? p;
  drucken ! p;
  mp:= p*p;
 *[m: integer;
     sieb(i-1) ? m; →
   *[ m>mp → mp:= mp + 2*p]
    [ m=mp → skip
    [] m<mp → sieb(i+1) ! m
    ]
 ]

sieb(50)::
  n: integer;
 *[ sieb(49) ? n →  drucken ! n ]

drucken::
  n: integer;
 *[(i:0..50) sieb(i) ? n →  …  drucken … ]
```

Das Sieb des Eratosthenes in CSP

```
sieb(0)::
  n: integer;
  drucken ! 2;
  n:= 3;
 *[ n<10000 → sieb(1) ! n; n:= n+2 ]
```

```
sieb(i:1..49)::
   p, mp: integer;
   sieb(i-1) ? p;
   drucken ! p;
   mp:= p*p;
  *[ m: integer;
       sieb(i-1) ? m; →
     *[ m>mp → mp:= mp + 2*p ]
      [ m=mp → skip
      [] m<mp → sieb(i+1) ! m
      ]
  ]
```

```
sieb(50)::
  n: integer;
 *[ sieb(49) ? n →  drucken ! n ]

drucken::
  n: integer;
 *[(i:0..50) sieb(i) ? n →  …  drucken … ]
```

Das Sieb des Eratosthenes in CSP

```
sieb(0)::
  n: integer;
  drucken ! 2;
  n:= 3;
 *[ n<10000 → sieb(1) ! n; n:= n+2 ]

sieb(i:1..49)::
  p, mp: integer;
  sieb(i-1) ? p;
  drucken ! p;
  mp:= p*p;
 *[m: integer;
    sieb(i-1) ? m; →
   *[ m>mp → mp:= mp + 2*p]
   [ m=mp → skip
   [] m<mp → sieb(i+1) ! m
   ]
 ]

sieb(50)::
  n: integer;
 *[ sieb(49) ? n →  drucken ! n ]

drucken::
  n: integer;
 *[(i:0..50) sieb(i) ? n →  …  drucken … ]
```

Fazit:

- Sehr einfaches Beispiel eines nebenläufigen Algorithmus
- Allerdings:
  - Kommunikationsaufwand überwiegt Rechenaufwand bei weitem
  - Problem des synchronen Nachrichtenaustauschs in CSP

⇒ Gegenseitiges Warten

Nur auf Mehrprozessorsystemen umsetzbar...

```
 Sieb0          Sieb1          Sieb2          Sieb3          Sieb4
(Vielfache 2)  (Vielfache 3)  (Vielfache 5)  (Vielfache 7)  (Vielfache 11)

 Drucken 2
      3 ------> Drucken 3
                mp= 3²= 9

      5 ------> 5 < 9  ------> Drucken 5
                               mp= 5²= 25

      7 ------> 7 < 9  ------> 7 < 25 ------> Drucken 7
                                              mp= 7²= 49

      9 ------> 9 = 9
                aussieben

     11 ------> 11 > 9
                mp = 9 + 2*3 = 15
                11 < 15 -----> 11 < 25 -----> 11 < 49  ------> Drucken 11
                                                               mp= 11²= 121

     13 ------> 13 < 15 -----> 13 < 25 -----> 13 < 49  ------> 13 < 121 ----->

     15 ------> 15 = 15
                aussieben
```

Fazit:

- Sehr einfaches Beispiel eines nebenläufigen Algorithmus
- Allerdings:
  - Kommunikationsaufwand überwiegt Rechenaufwand bei weitem
  - Problem des synchronen Nachrichtenaustauschs in CSP

    ⇒ Gegenseitiges Warten

    Nur auf Mehrprozessorsystemen umsetzbar...

- Bei asynchronem Nachrichtenaustausch mit mehrelementigem Puffer weitergehende Unabhängigkeit zwischen den Prozessen

  ⇒ Dennoch: Beispiel insgesamt zu einfach für großen Nutzen

Exkurs: Kommunikation in Erlang

- Asynchroner Nachrichtenaustausch
- Senden [syntaktisch wie CSP-Ausgabeanweisung]

```
Process ! Message
```

- Empfangen [ähnlich zu CSP-Alternativ-Anweisung/Unix-select()]

```
receive
  MessagePattern1 ->
    Actions1;
  MessagePattern2 ->
    Actions2;
   …
  after Time ->
    TimeOutActions
end
```

CSP-Beispiel: Auf mehrere Eingaben warten:

```
[ P1 ? variable1 → anweisung1
[] P2 ? variable2 → anweisung2
[] P3 ? variable3 → anweisung3
]
```

Fragen – Teil 3

- Welche Nachteile hat die CSP-Lösung des „Sieb des Eratosthenes“?

Zusammenfassung

- Nachrichtenaustausch
  - synchron (*Rendezvous*)
  - asynchron (Puffer)
  - Unicast, Multicast, Broadcast
- CSP (Communicating Sequential Processes)
  - Kommunikationsanweisungen (synchron)
  - Alternativ-Anweisung (*guarded commands*)
  - Speisende Philosophen in CSP
  - Sieb des Eratosthenes in CSP
- Kurzüberblick Erlang

Nachrichtenaustausch – Fragen

1. Was versteht man unter *synchronem* bzw. *asynchronem* Nachrichtenaustausch? Inwiefern sind diese beiden Kommunikationsformen aufeinander abbildbar?
2. Wie kann man die Synchronisationseigenschaften von synchronem bzw. asynchronem Nachrichtenaustausch mit Semaphoren modellieren?
3. Wozu verwendet man *Kanäle* bzw. *Ports*? Was ist das?
4. Was ist ein *guarded command*? Warum kann die Verwendung eines solchen Konzepts gerade in Zusammenhang mit Nachrichtenaustauschvorgängen interessant sein?
5. Welche Nachteile hat die CSP-Lösung des „Sieb des Eratosthenes“?