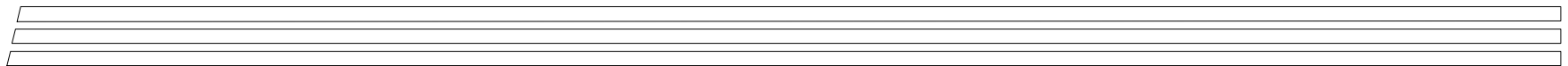


Synchronisation und Kommunikation über Nachrichten

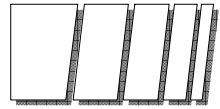
- meist bei verteiltem Speicher, kein gemeinsamer Speicher ->
 - keine globalen Variablen
 - keine zu schützenden Datenbereiche
- Kommunikation über „Kanäle“ und Nachrichtenaustausch (Message Passing)
- Modell:



- statt Schreiben/Lesen gemeinsamer Variablen
Senden /Empfangen (send/receive) von Nachrichten

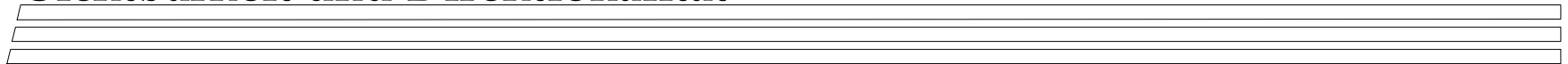


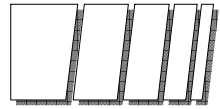
- implizite Synchronisation



Kommunikationsmodelle

- **Basiskonzepte:**
 - Prozesse und Kanäle
 - **Sende- und Empfangsprimitiven:**
 - Sende „Nachricht“ an „Empfänger“
 - Empfange „Nachricht“ von „Sender“
- **Merkmale**
 - **Bezeichnung von Quelle und Ziel der Kommunikation:**
 - Prozessbenennung (implizite Kanäle, 1:1, unidirektional) vs
 - Kanäle (mehrere Kommunikationswege zwischen gleichen Prozessen)
 - **Anzahl der Kommunikationspartner:**
 - 1:1 (Punkt-zu-Punkt), 1:n (broadcast, multicast)
 - m:n (schwarzes Brett, meist bei gemeinsamem Speicher)
 - m:1 (Briefkasten mit einem Empfänger, Multiplexer in verteilten Sys.)
 - **Synchronisation**
 - **Sichtbarkeit und Direktionalität**





Asynchrone Kommunikation

Sender braucht nicht auf Empfang der Nachricht zu warten

=> Kanalpuffer erforderlich:

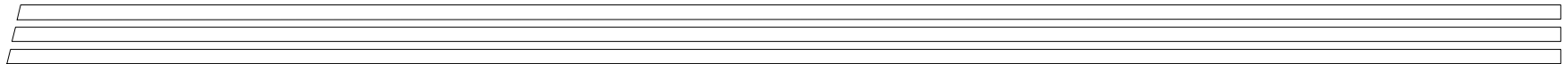
- **unbeschränkt: send blockiert nie**
- **beschränkt: Blockade bei vollem Puffer**

gepuffertes vs. ungepuffertes Senden:

- **gepuffert: Nachricht wird auf Sendepuffer in Systempuffer kopiert, bevor sie aufs Verbindungsnetzwerk geschrieben wird**
- **ungepuffert: vom Sendepuffer direkt aufs Netz schnell, aber nicht immer möglich**

nicht-blockierendes vs blockierendes Senden:

- **nicht-blockierend: Anstoss des Nachrichtenversands mit direkter Weitergabe der Kontrolle (Gefahr des Überschreibens des Sendepuffers bevor vorherige Nachricht versendet)**
- **blockierend: Warten bis Sendepuffer ausgelesen ist**

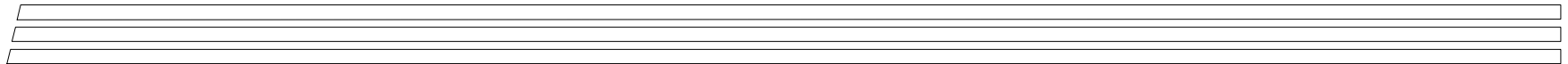


Synchrone Kommunikation

Sender und Empfänger warten auf Kommunikation

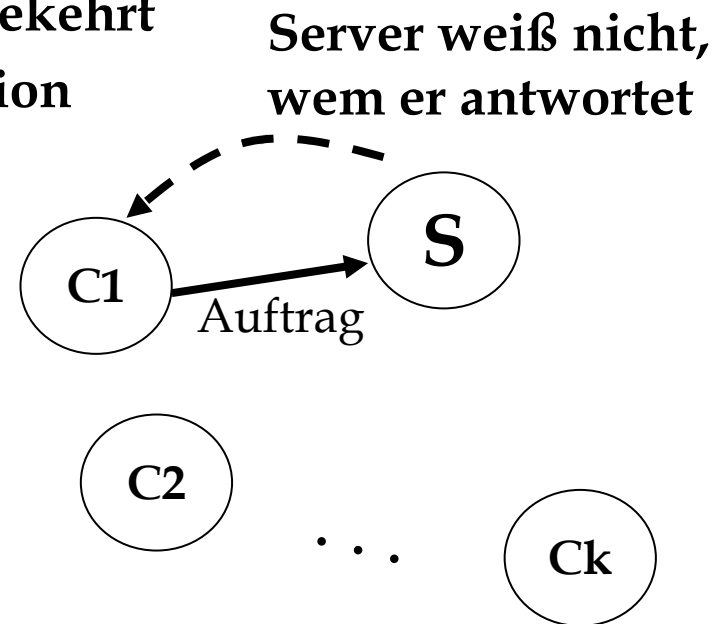
⇒ keine Zwischenpufferung der Daten notwendig

⇒ direkte Übertragung vom Sende- in Empfangsbereich



Sichtbarkeit und Direktionalität

- **symmetrisch:**
Kommunikationspartner kennen einander in gleicher Weise
=> meist datenorientierter Nachrichtenaustausch
z.B. send „Nachricht“ / „Auftrag“ to „receiver“
- **asymmetrisch:**
Sender kennt Empfänger, aber nicht umgekehrt
=> meist aktionsorientierte Kommunikation
z.B. receive „Auftrag“



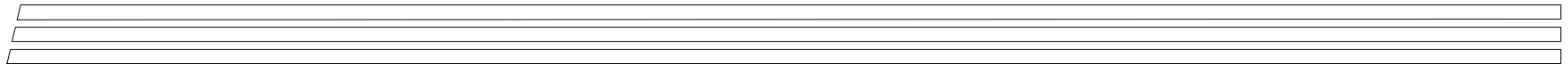
Bsp: Client/Server-Systeme
Clients müssen die Server kennen,
aber nicht die Server die Clients.

Beispielsprachen - OCCAM

- Vorläufer CSP (Communicating Sequential Processes), 1978
- Merkmale:
 - unidirektionale Punkt-zu-Punkt-Kanäle
 - symmetrische, synchrone Kommunikation
 - statisches Kanalkonzept (Festlegung aller Kanäle zur Compilezeit)
 - selektives Kommunikationskommando
 - gleichzeitiges Warten auf Eingaben verschiedener Kanäle

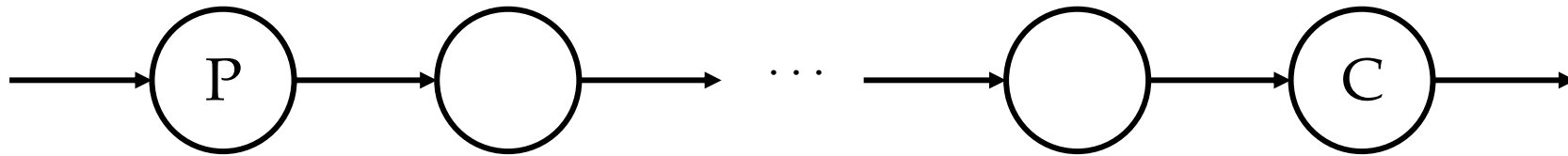
ALT

| | | |
|---------------------|---|--|
| B_1 & INPUT_GUARD_1 | } | nur Empfangsanweisungen, in CSP: Sende- & Empfangsanw. in Guards erlaubt |
| EXPR_1 | | |
| ... | | |
| B_k & INPUT_GUARD_k | } | |
| EXPR_k | | |



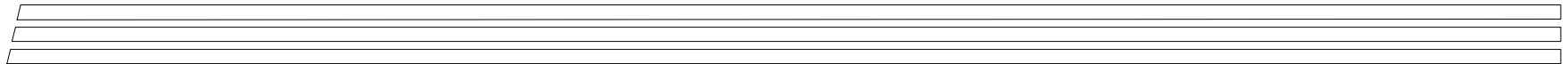
Beispiel: Puffer in Occam

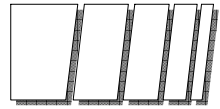
als Fließband:



```
PROC buffer (CHAN OF INT source, sink)
  WHILE true
    INT local;
    SEQ source?local
        sink!local

[n+1] CHAN OF INT stream:
PAR producer (stream[0])
  PAR index = 0 FOR n      buffer(stream[i], stream[i+1])
  consumer (stream[n+1])
```





Parallel Puffer in Occam

```
PROC buffer ([n] CHAN OF INT source, sink)
```

```
  WHILE TRUE
```

```
    INT local:
```

```
    PAR index = 0 FOR n
```

```
      source[index] ? local
```

```
      sink [index] ! local
```

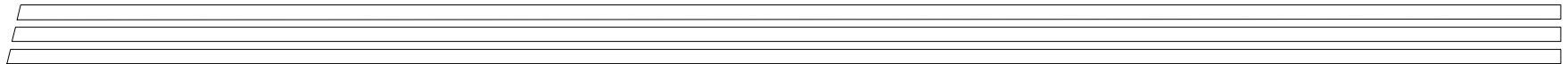
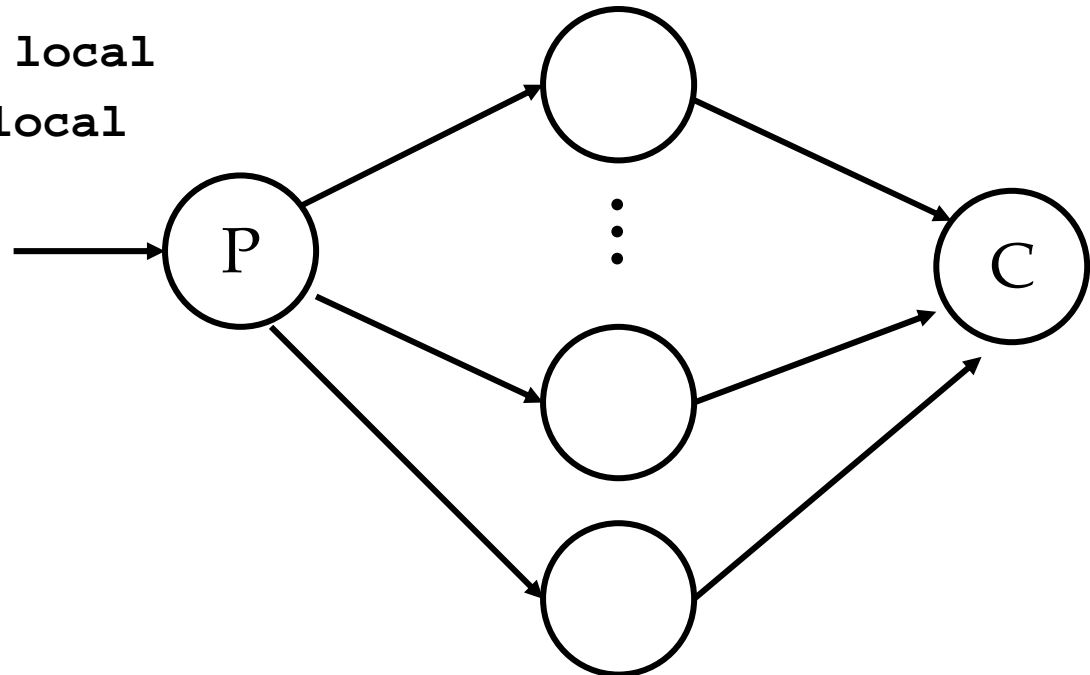
```
[n] CHAN OF INT in, out
```

```
PAR
```

```
  producer (in)
```

```
  buffer (in, out)
```

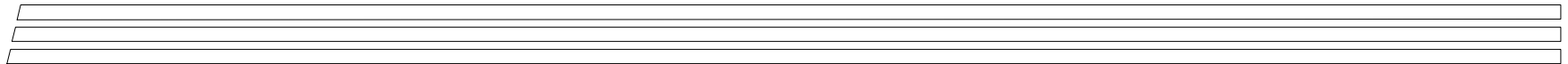
```
  consumer (out)
```

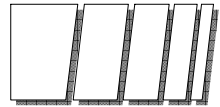




Beispielsprachen - Ada

- **Entwicklung durch DoD, 1980**
- **Merkmale:**
 - **synchrone Kommunikation**
 - **rendezvous-Konzept**
 - **asymmetrisch**
 - **n : 1**





Beispiel: Bounded Buffer in Ada

Task Spezifikation

- Deklaration des Namens und der Prozeduren
- für alle Prozesse sichtbar

task buffer is

```
entry Append (I: in Integer);  
entry Take (I: out Integer)
```

end buffer;

Task Implementation

- Definition der Prozeduren

task body buffer is

```
N: constant Integer := 100;  
B: array (0..N-1) of Integer;  
anfang, ende : Integer := 0;  
anzahl: Integer := 0
```

```
begin
```

```
loop
```

```
select
```

```
when anzahl < N =>
```

```
accept Append(I:in Integer)
```

```
do B[ende]:= I; end Append;
```

```
anzahl := anzahl+1;
```

```
ende := (ende+1) mod N
```

```
or
```

```
when anzahl > 0 =>
```

```
accept Take(I:out Integer)
```

```
do I := B(anfang); end Take;
```

```
anzahl := anzahl-1;
```

```
anfang := (anfang+1) mod N;
```

```
end select;
```

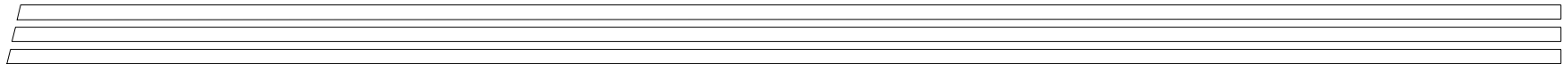
```
end loop;
```

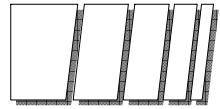
```
end buffer;
```

Beispielsprachen - SR/MPD

- **Modellierung verschiedener Konzepte**
- **Kernkonzept: Operationen**
- **Aufruf einer Operation:**
 - **asynchrones send**
 - **synchrones call**
- **Ausführung einer Operation:**
 - **mittels proc -> eigener Prozess**
 - **mittels in -> rendezvous -> bestehender Prozess**

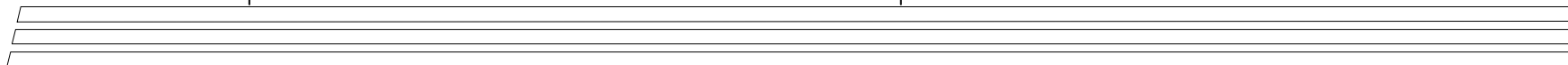
=> 4 verschiedene Kombinationen





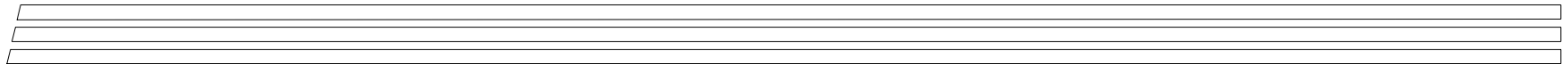
Aufruf und Ausführung von Operationen

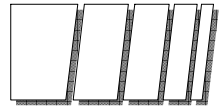
| | Aufruf | | | | |
|------------|---|------|--|--|--|
| Ausführung | | call | | send | |
| proc | <p>RPC remote procedure call Prozedur- aufruf</p> | | | <p>fork dyna- mische Prozess- erzeugung</p> | |
| in | <p>rendez- vous synchrone Nachrichten- übertragung</p> | | | <p>asynchrone Nachrichten- übertragung</p> | |



Operationen in MPD

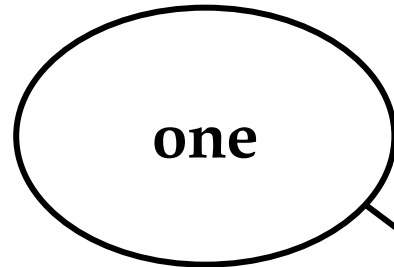
- **Deklaration:** `op <name> (<params>) { <invocation> }`
 mögliche Invocations: {send}, {call}, {send,call}
- **Implementierung (Deklaration von Prozess- und Prozedurrümpfen)**
 `proc <name> (<params>) {`
 `<body> }`
- **Spezialfälle (Schlüsselwörter)**
 procedure: Operation mit call-Restriktion
 process: Operation mit send-Restriktion
- **Kanäle sind Operationen ohne Implementierung.**
 Obige Deklaration kann als m:n Kanal betrachtet werden.
 `send <name> (<params>)` = nicht-blockierendes Senden
 `receive <name> (<variable>)` = Nachrichtenempfang



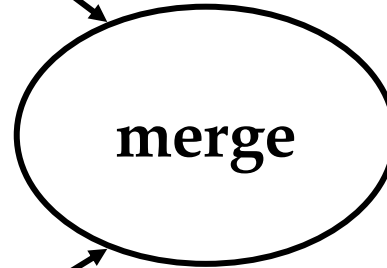


Beispiel: Mischen geordneter Folgen

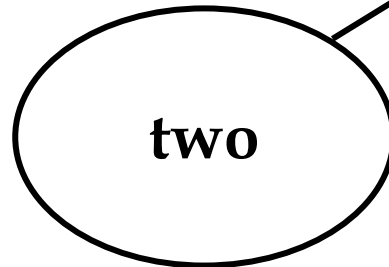
...
send strm1 (v1)
...
send strm1(EOS)
...



strm1



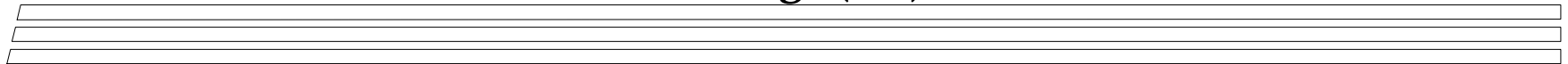
strm2

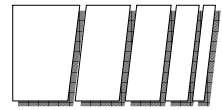


...
send strm2 (v2)
...
send strm2(EOS)
...

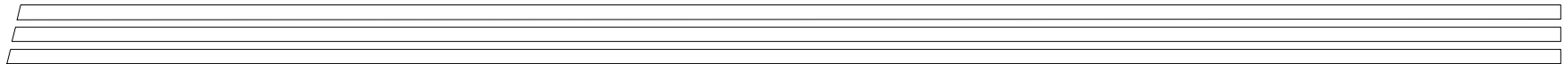
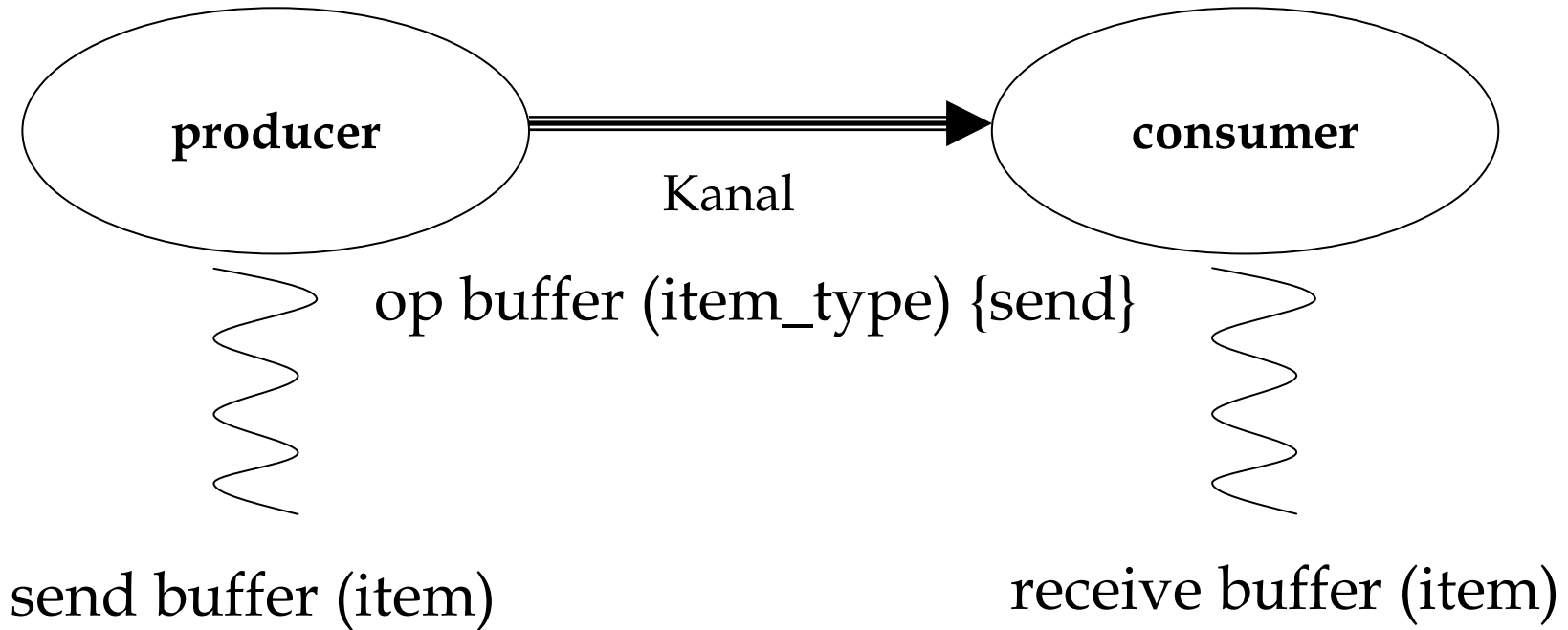
```
receive strm1(X1)
receive strm2(X2)
while (X1 < EOS or
      X2 < EOS)
{ if (X1 <= X2)
  {write (X1)
   receive strm1(X1) }
  else
  {write (X2)
   receive strm2(X2) }
  fi
}
```

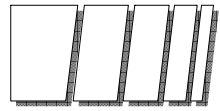
op strm 1(int), strm2 (int)
const EOS = high(int)





Beispiel: Erzeuger-Verbraucher





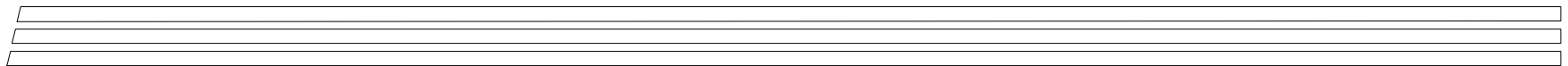
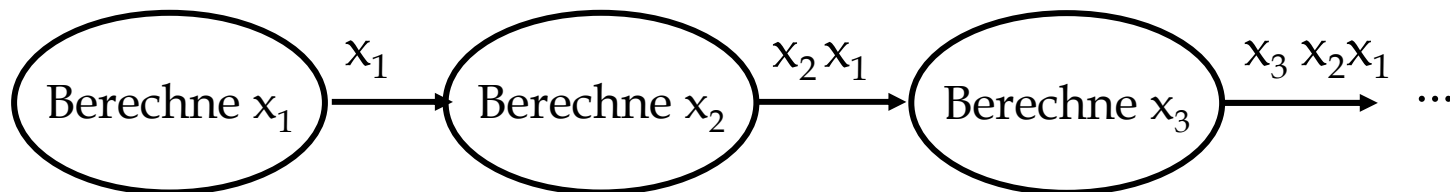
Fallstudie: Lösung von Dreiecksgleichungssystemen

- $\underline{A} \underline{x} = \underline{b}$ mit unterer Dreiecksmatrix

$$\underline{A} = \begin{pmatrix} a_{11} & 0 & \dots & 0 \\ a_{21} & a_{22} & \dots & 0 \\ \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & 0 \\ a_{n1} & \dots & \dots & a_{nn} \end{pmatrix} \quad \underline{x} = (x_1, \dots, x_n), \quad \underline{b} = (b_1, \dots, b_n)$$

- Lösung: $x_1 = b_1 / a_{11}; x_2 = (b_2 - a_{21} x_1) / a_{22}; \dots;$
 $x_n = (b_n - \sum_{j=1}^{n-1} a_{jn} x_j) / a_{nn}$

- Ansatz: Pipeline-Algorithmus



Programm *backsubstitution.mpd*

```
resource backsubstitution ()
const int n = 5;    real A[n,n]; real b[n];    real x[n]
op pipechan [n+1] (real)
process pipe [i=1 to n] {
    real sum = 0; real xvalue
    for [j = 1 to i-1] {
        receive pipechan[i](xvalue)
        send pipechan[i+1](xvalue)
        sum = sum + A[i,j] * xvalue
    }
    x[i] = (b[i] - sum)/A[i,i]
    send pipechan[i+1](x[i])
}
process writer () {
    int x
    receive pipechan[n+1](x)
    while (x != 0) { write(x); receive pipechan[n+1](x) }
}
end
```
