Occam

Programmiersprache Occam

- William of Occam
  [Wilhelm von Ockham]

  - Franziskaner; ca. 1290-1350
  - Oxford --> Paris --> München

- Prinzip der Denkökonomie ("Occam's Razor"):
  - *Entia non sunt multiplicanda praeter necessitatem*
  - Minimalprinzip: Keine unnötigen Existenzannahmen

Sprache Occam:

Communicating Sequential Processes

- Sehr spartanisch
- Aufbauend auf CSP (Hoare, 1978)
- Entwickelt ab 1982 (David May, INMOS)
- Occam --> Occam 2 --> Occam 3

(Gleitpunktzahlen, Funktionen, Datentypen,...)

Idee:
- Programmieren durch kommunizierende Prozesse
- Kommunikation, Prozessverwaltung, Synchronisat. billig wie jede andere "primitive" Operation
- Programmiermodell realisiert durch spezielle Prozessoren (Transputer bzw. Netz von Transputern)

- *The Occam archive:* http://www.comlab.ox.ac.uk/archive/occam.html
- *Occam resources*: http://www.hensa.ac.uk/parallel/occam/documentation/
- *Occam reference manual*:
  http://www.hensa.ac.uk/parallel/occam/documentation/oc21refman.ps.gz

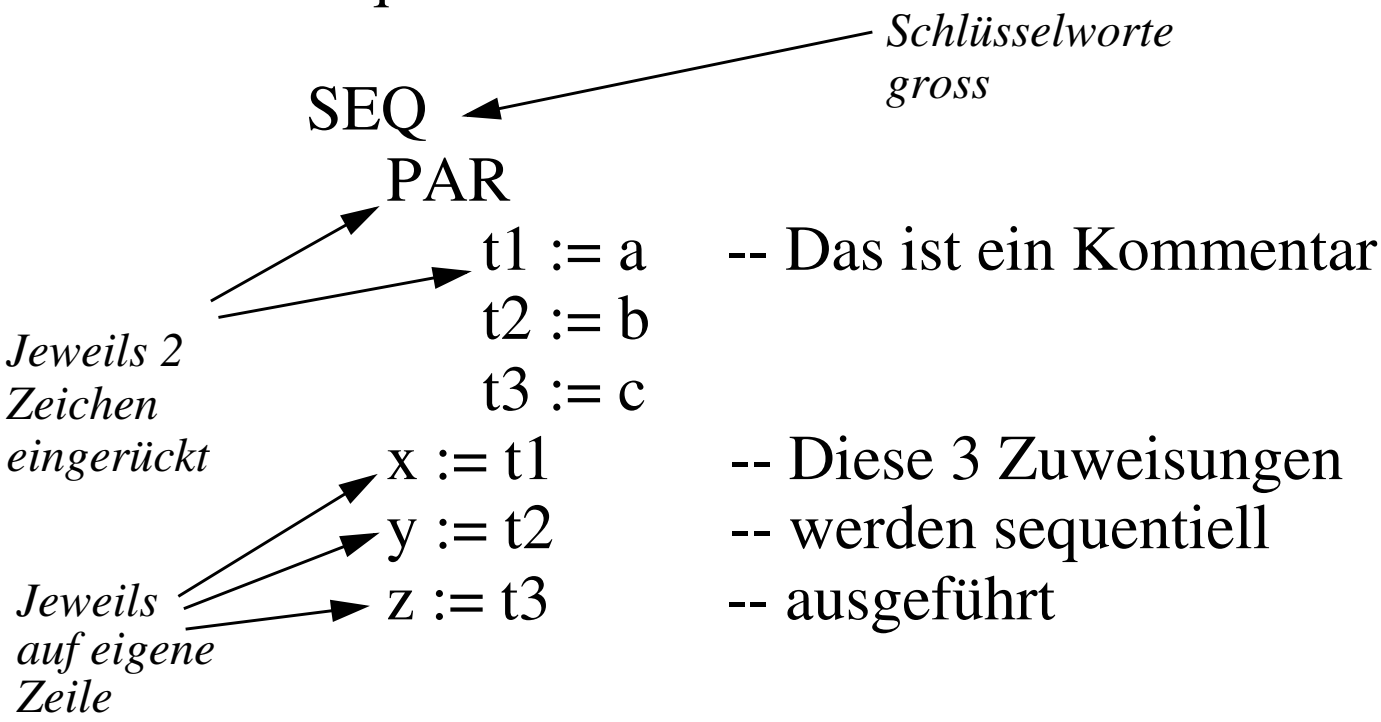
Occam: Zuweisungen

```
x := y

x,y,z := a,b,c
```

alle Ausdrücke der rechten Seite werden vor der Zuweisung ausgewertet -->

Äquivalent zu:



So geht es auch:

```
SEQ
   PAR
      ... -- Zuweisung an temporäre Variablen
   PAR
      ... -- Zuw. der temp. Var. an die linke Seite
```

```
x, y := y, x  -- vertauscht den Inhalt der Variablen!
```

Occam: Parallele Prozesse

```
PAR
    x := y+1
    z := y+2
```

Gesamtprozess beendet, wenn beide Einzelprozesse beendet

Die beiden *Prozesse* werden parallel ausgeführt (oder in beliebiger Reihenfolge)

Achtung: Kein paralleler Prozess darf eine Variable ändern, die in einem der anderen auftritt!

Bemerkung: Lesen gemeinsamer / globaler Variablen ist prinzipiell möglich. Aber Vorsicht, wenn die Prozesse auf verschiedenen Prozessoren ausgeführt werden (--> Kommunikation)!

In Occam gibt es also:

- anonyme dynamische Prozesse
- geschachtelte Prozesse (die allerdings nicht nebenläufig arbeiten)
- globale (und lokale) Variablen
- keine weiteren Synchronisationsmittel ausser *synchroner* Kommunikation über *Kanäle*

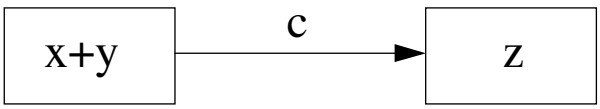
Occam: Kommunikation

- Kommunikation geschieht *synchron* über *Kanäle*!

```
PAR
    c ! x+y
    c ? z
```

hat die gleiche Wirkung wie z:=x+y

Kanal



Kanäle nur zwischen je *zwei* Prozessen, wobei einer schreibt, der andere liest.

```
  P1              P2

SEQ             SEQ
   ch1 ! a         ch1 ? x
   ch2 ? b         ch2 ! y
```

funktioniert

```
  P1              P2

SEQ             SEQ
   ch1 ! a         ch2 ! y
   ch2 ? b         ch1 ? x
```

Deadlock!

WHILE-Schleifen in Occam

```
INT a:
SEQ
    in ? a
    WHILE  a <= 32767
        SEQ
            a := a*a
            out ! a
            in ? a
```

Deklaration

Nur ein Prozess ("SEQ") nach While

Doppelpuffer in Occam

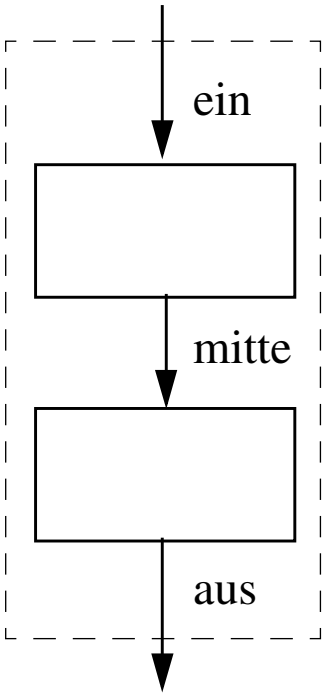
mit Kapazität 2

Pipeline-Lösung (“fall through buffer”):

Deklaration typisierter Kanäle

zwei endlos existierende parallele Prozesse

```
CHAN OF INT: ein, mitte, aus:
INT x, y:
PAR
    WHILE TRUE
        SEQ
            ein ? x
            mitte ! x
    WHILE TRUE
        SEQ
            mitte ? y
            aus ! y
```



Alternierende Puffer:

```
CHAN OF INT ein, aus:
INT x,y:
SEQ
    ein ? x
    WHILE TRUE
        SEQ
            PAR
                ein ? y
                aus ! x
            PAR
                ein ? x
                aus ! y
```

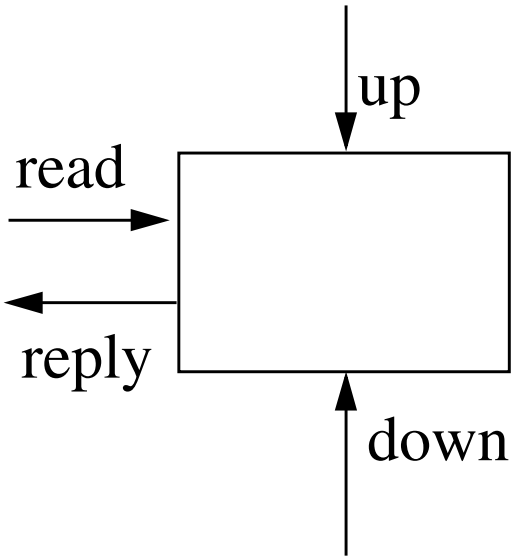
- Haben beide Lösungen die gleiche Semantik? (FIFO, keine unnötige Blockierung von Konsument oder Produzent...) Beweis?
- Vor-/ Nachteile der Implementierungen?
- Welche Realisierung ist besser?
- Lösungen skalierbar für mehr als zwei Pufferplätze?

Occam: ALT-Konstrukt

- Gleichzeitiges Warten auf Empfang bzgl. mehrerer Kanäle
- Höchstens eine der Alternativen wird gewählt

Beispiel:

```
ALT
    up ? increment
        x := x + increment
    down ? decrement
        x := x - decrement
    read ? request
        reply ! x
```

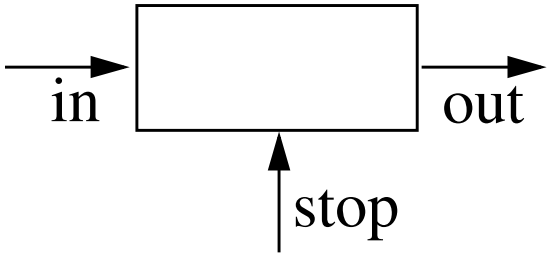


Beispiel zeigt, wie der wechselseitig ausgeschlossene Zugriff auf eine Variablen geregelt werden kann (*Monitor-Konzept*)

Oft sinnvoll: Umfassende WHILE-Schleife

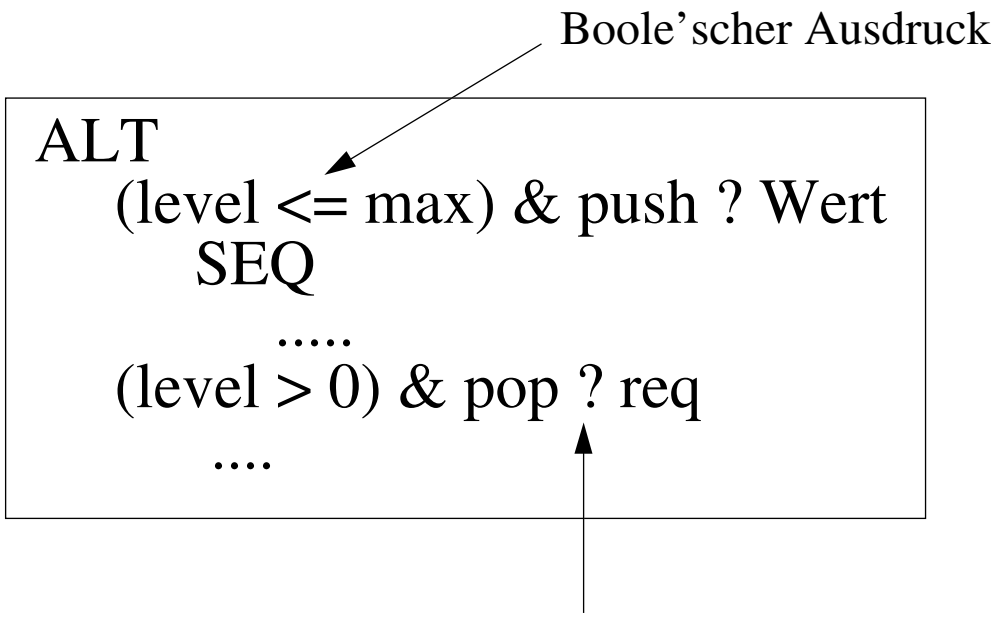
```
WHILE going
  ALT
    in ? z
      out ! z
    stop ? z
      going := FALSE
```

"Ventil":



Occam: Guards

Beispiel "Stack":



- Spricht etwas dagegen, in einem ALT-Konstrukt auch ein '!' ("output guard") anstelle eines '?' ("input guard") zuzulassen?

  (alternatives Empfangen / Senden, je nachdem, was gerade - abhängig von den Kommunikationspartnern - geht)

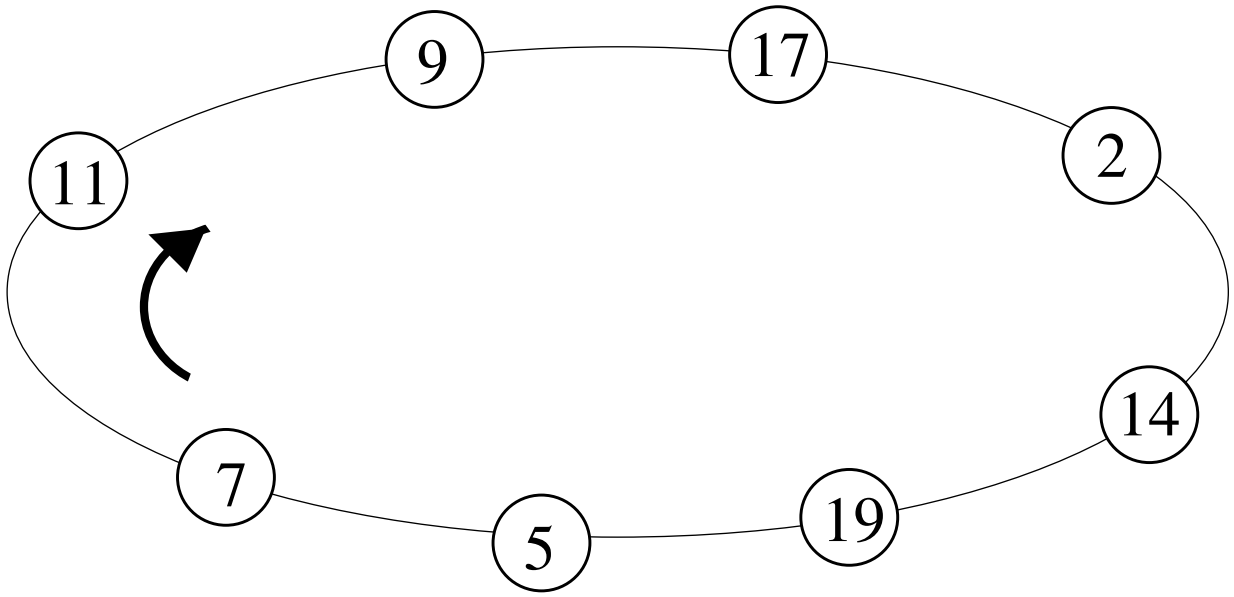
Election-Algorithmus in Occam

Dezentrale “Wahl” eines eindeutigen “Masters”

- Jeder Prozess mit Identität *p* hat lokale Variable *m*

- *m* ist initial *p*; am Ende enthält *m* die Identität des Masters

- Voraussetzung hier: Ring, wobei alle Identitäten der beteiligten Prozesse verschieden sind



- Lösung: Message-extinction-Prinzip:

  - Der Prozess mit der *kleinsten* Identität p soll gewinnen

  - Jeder Prozess gibt neue “Approximation” des globalen Minimums an Nachbarn weiter

  - Schlechtere empfangene “Approximationen” werden verschluckt

IF-Anweisung

```
IF
   a >= b
      max := a
   a <= b
      max := b
```

- betont willkürliche Auswahl im Falle a=b

- Beliebige Anzahl von Zweigen

- Erster ’true-Zweig’ wird genommen

- STOP falls kein ’true-Zweig’ gefunden

  ↑ "elementarer" Prozess, der nie beendet wird

```
IF
   x < 0
      x := -x
   TRUE
      SKIP
```

’else-Fall’

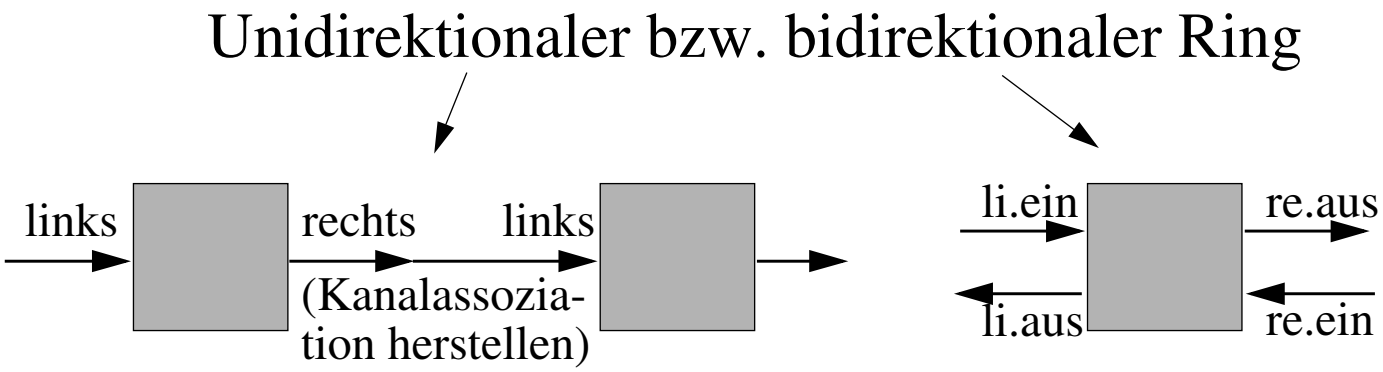
Prozess, der nichts tut

- Frage: Was ist die Semantik von:

```
WHILE TRUE
   SKIP
```

- Ist dies äquivalent zu STOP ?

Lösungsansatz Election-Algorithmus?




```
...
m := p
WHILE  aktiv
    SEQ
        links ? x
        IF
            (x < m)
                SEQ
                    m := x
                    rechts ! m
            ...
```

p hat in jedem Prozess einen anderen Wert

```
ALT
    li.ein ? x
        SKIP
    re.ein ? x
        SKIP
```

So geht's auch:

```
PAR
    m := x
    rechts! x
```

```
PAR
    li.aus ! m
    re.aus ! m
```

Führt das nicht zu einem Deadlock, wenn zwei benachbarte Prozesse sich gleichzeitig etwas gegenseitig zusenden wollen?

==> So geht es nicht, es kommt nichts in Gang! (Alle warten)

- Andere Idee: Zuerst senden!

```
m := ...
rechts ! m
WHILE  aktiv
...
```

So geht es auch nicht: Deadlock!

Frage: Ginge es bei asynchroner Kommunikation?

Das Output-Guard-Problem

zur Lösung des unidirektionalen Election-Problems

- Sollte man vielleicht “gleichzeitig” oder “alternativ” senden *und* empfangen? Wie ist es mit folgenden Ansätzen?

```
PAR
    links ? x
    rechts ! m
```

Das gefällt uns nicht:
Man kann doch nicht immer zusätzlich senden, wenn man empfangen will (bzw. umgekehrt)!
Aber: funktioniert es prinzipiell?

```
ALT
    links ? x
      ...
    rechts ! m
      ...
```

Alternatives senden und “output guards” sind in Occam verboten! Wieso?

- Problem der "gemischten Kommunikationswächter"
  - Symmetrie muss “irgendwie” automatisch gebrochen werden, und zwar in dezentraler Weise!

    ==> “output-guard-Problem”

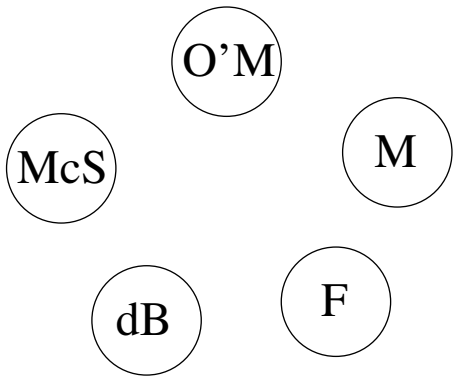
- Wie wäre folgender Ansatz?

```
m := ...
PAR
    rechts ! m
    WHILE  aktiv
      SEQ
        links ? x
        ...
```

anfangs gleichzeitig senden und den eigentlichen Prozess ausführen

- würden hier nicht zwei parallele Prozesse auf dem gleichen Kanal senden?
- anders aber ähnlich?

Samuel Beckett hat bereits1944 das Koordinationsproblem beim “alternativen” Senden / Empfangen beschrieben!

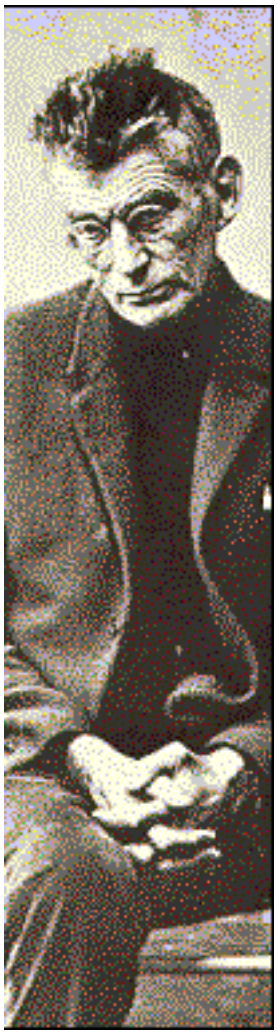


Samuel Beckett (1906-1989)**: Watt**

...Dann begannen sie, einander anzublicken, und es verging eine ganze Weile, ehe es ihnen gelang. Nicht, dass sie einander lange angeblickt hätten, nein, so dumm waren sie nicht. Aber wenn, falls fünf Personen einander anblicken, theoretisch dazu nur zwanzig Blicke erforderlich sind, da jeder viermal blickt, so reicht diese Zahl praktisch selten aus, wegen der vielen Blicke, die sich verirren.

Zum Beispiel, Mr. Fitzwein blickt Mr. Magershon zu seinen Rechten an. Aber Mr. Magershon blickte gerade nicht Mr. Fitzwein zu seiner Linken an, sondern Mr. O’Meldon zu seiner Rechten. Aber Mr. O’Meldon blickte nicht Mr. Magershon zu seiner Linken an, sondern, seinen Kopf nach vorn streckend, Mr. MacStern, vier Sitze links von ihm, am anderen Ende des Tisches. Aber Mr. MacStern blickte nicht, seinen Kopf nach vorn streckend, Mr. O’Meldon, vier Sitze rechts von ihm, am anderen Ende des Tisches an, sondern sitzt kerzengerade und blickt Mr. de Baker zu seiner Rechten an. Aber Mr. de Baker blickt nicht gerade Mr. MacStern zu seiner Linken an, sondern Mr. Fitzwein zu seiner Rechten...

...Und das ist noch nicht alles. Denn viele, viele Blicke können noch geworfen werden, und viel, viel Zeit kann noch verloren gehen, ehe jedes Auge das Auge sieht, das es sucht, und in jeden Geist die Kraft, der Trost und die Zuversicht fliessen, die nötig sind, um zur Tagesordnung zurückzukehren. Und all dies aus Mangel an Methode...



Samuel Beckett

...Und eine der besten Methoden... ist vielleicht die, dass Nummern an die Mitglieder des Komitees gegeben werden, eins, zwei, drei, vier, fünf, sechs, sieben, und so weiter, ebensoviele Nummern, wie es Mitglieder des Komitees gibt, so dass jedes Mitglied des Komitees seine Nummer hat und kein Mitglied des Komitees nummernlos ist...

Wenn dann der Augenblick kommt... sollen alle Mitglieder ausser Nummer eins gemeinsam Nummer eins anblicken, und Nummer eins soll sie alle der Reihe nach anblicken... Dann sollen...

.
.
.

Alternatives Senden, Output-Guards

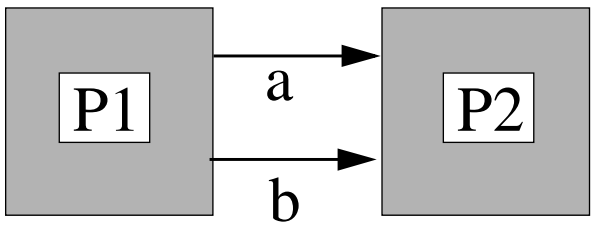
... ist in Occam nicht möglich!

Wieso? - Semantik zunächst nicht klar
(Fairness, livelocks, Determinismus...)
- Implementierung schwierig / ineffizient

Es war lange unklar, ob bzw. wie man solche Programme in solche ohne diese Konstrukte transformieren kann!

- Was könnte man mit '!' in ALT noch tun?

Was bewirken folgende zwei Prozesse?



P1:

```
ALT
    a ! x
        Wert := 0
    b ! x
        Wert := 1
```

P2: (Sei W ∈ {0,1})

```
ALT
    W=0  &  a ? y
        SKIP
    W=1  &  b ? y
        SKIP
```

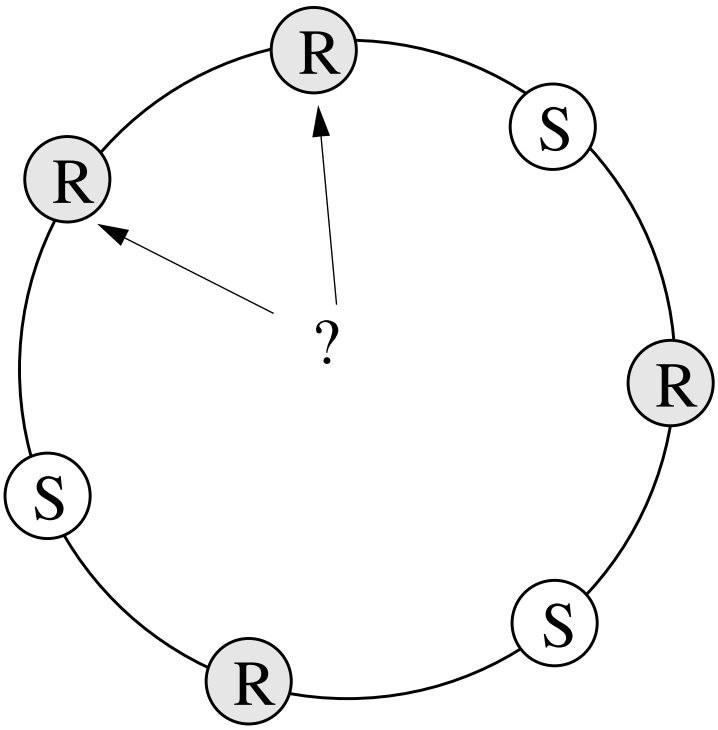
- Lässt sich iterieren! (Bitweise Übertragung)
- Geht nicht bei asynchroner Kommunikation!
- Anwendung bei Telefon mit defektem Mikrofon (P2)...
- Vorsicht mit Beweisen, dass etwas *nicht* geht!

Election - Lösungsversuch mit Occam

Idee: Zwei Typen von Prozessen

- Typ S: Versendet zunächst
- Typ R: Empfängt zunächst

Lösung ist allerdings syntaktisch nicht mehr ganz symmetrisch!



Idee: S-, R-Prozesse abwechselnd plazieren ("Schachbrettmuster")

```
m := p
links ? x
IF
(x < m)
    m := x
TRUE
    SKIP
rechts ! m
WHILE  aktiv ...
```

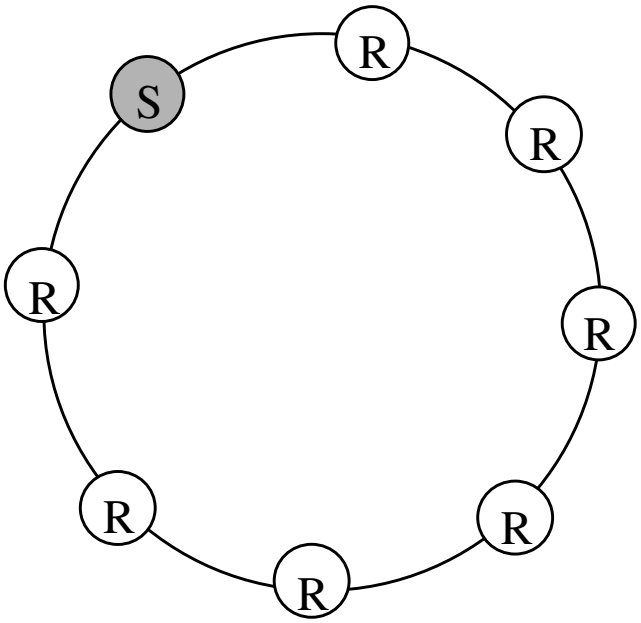
Nur bei R-Prozessen

Garantiert, dass jeder Prozess mindestens ein Mal sendet!

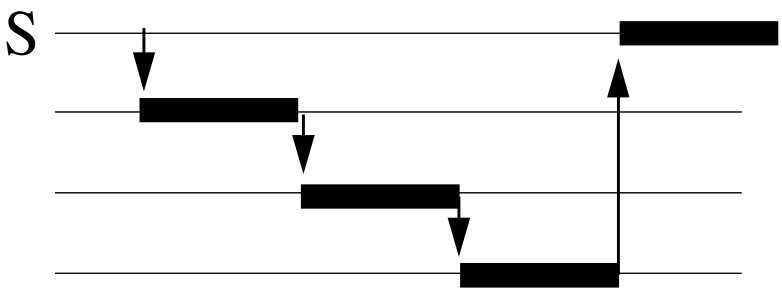
Frage: Welches zahlenmässige Verhältnis von S-/R-Prozessen ist günstig?

Variation der Anzahl der S/R-Prozesse

- Annahme: Senden am *Ende* einer Aktivitätsphase



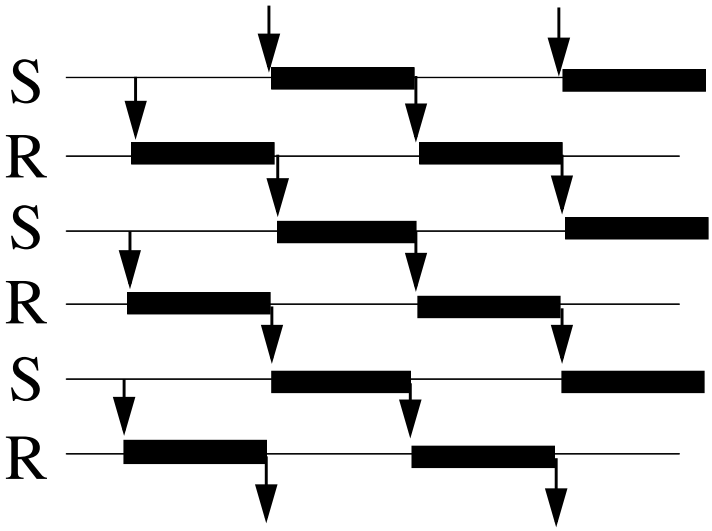
1) *Nur ein einziger Sende-Prozess*

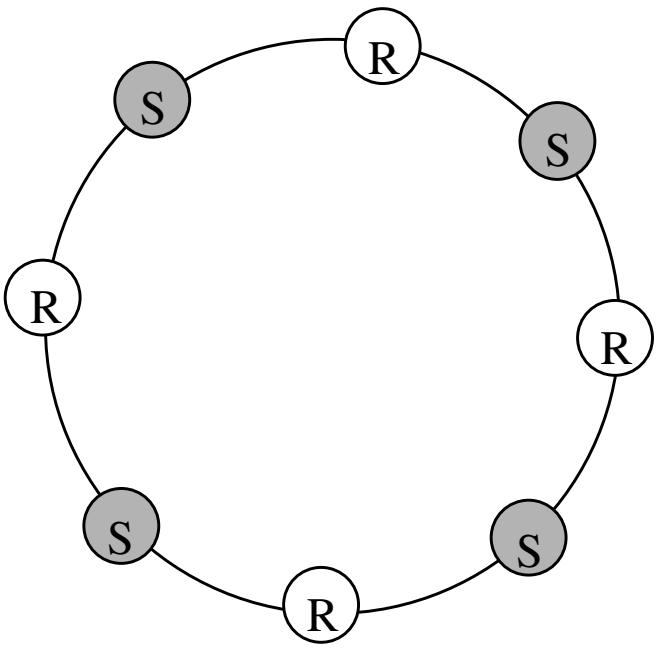


Beobachtung: keine Parallelität (höchstens ein Prozess arbeitet)

---

2) *Abwechselnd S/R-Prozesse ("Schachbrettmuster")*

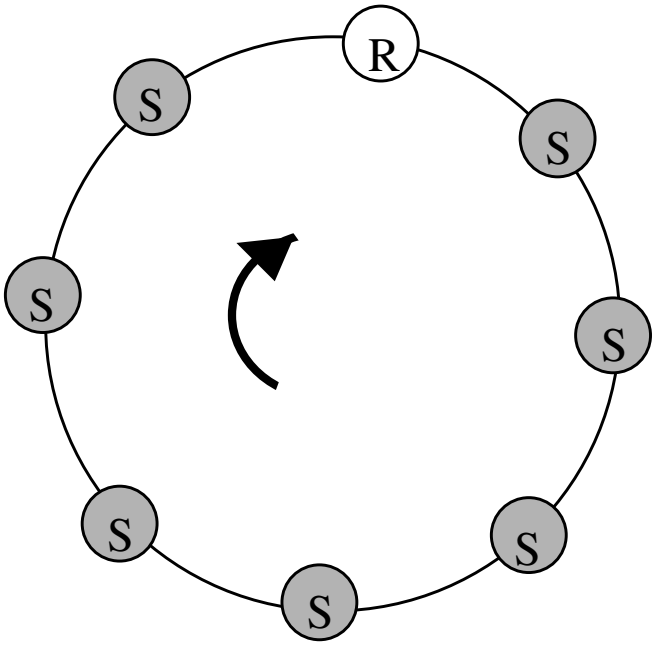




Beobachtung: etwa die Hälfte aller Prozesse ist gleichzeitig aktiv

Geht es nicht besser?

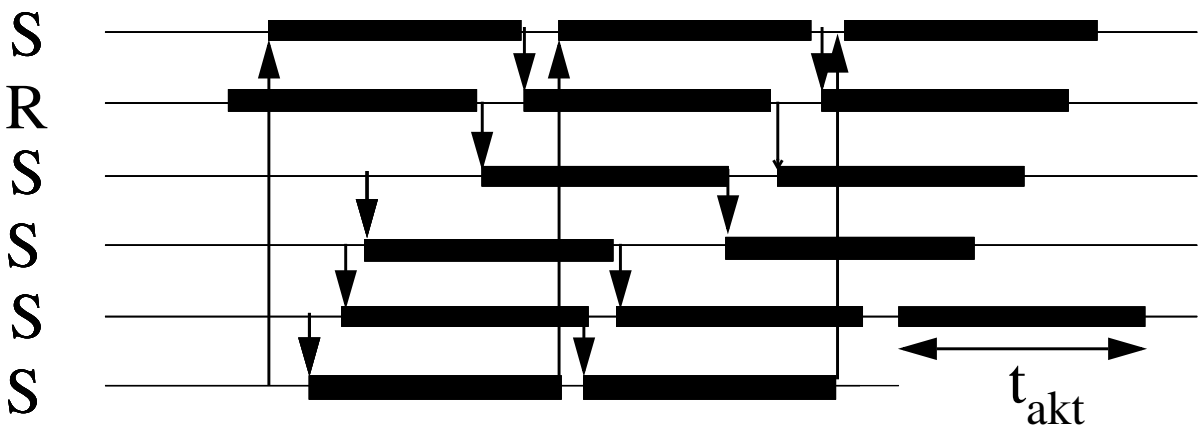
3) *Nur ein einziger R-Prozess*



Beobachtung:

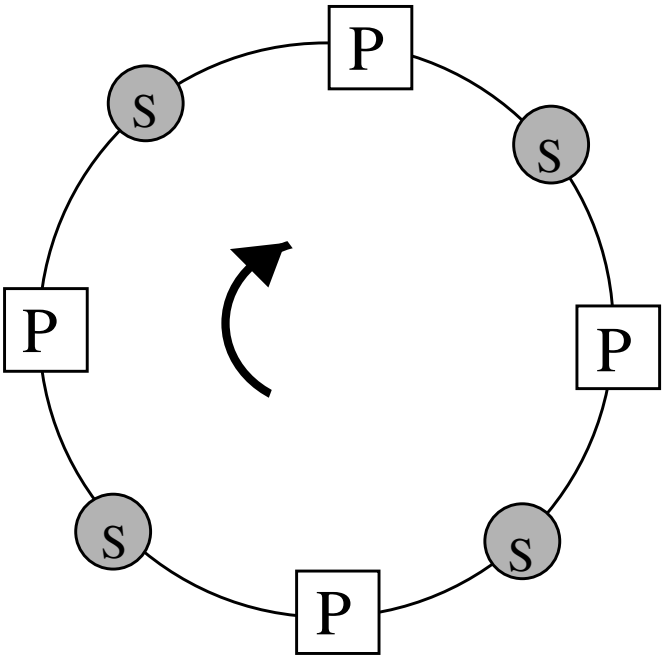
Empfangsbereitschaft läuft schnell (t_{trans}) in umgekehrter Richtung!

Bei langen Aktivitäts-phasen ($t_{akt} > n\ t_{trans}$): fast immer n-1 Pro-zesse aktiv!



- Geht es noch besser?
  - sogar syntaktisch symmetrisch?

Election mit Puffern

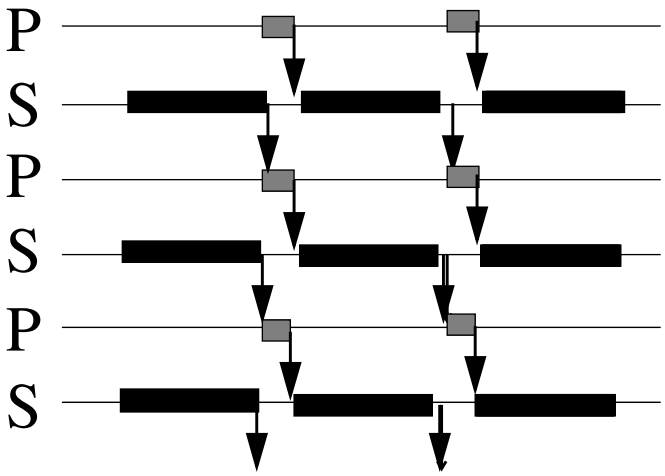


- Wieviele Puffer?
  - zwischen je 2 Prozesse?
  - genügt ein einziger?

- Kapazität der Puffer?

- Overhead, da Puffer als eigene Prozesse realisiert werden müssen?

- Effekt: Simulation von *asynchroner* Kommunikation



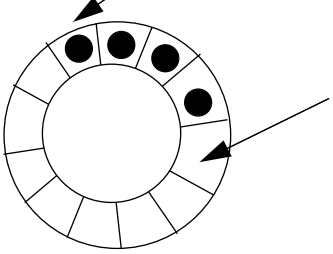
Abzüglich der Puffer-Verzögerungszeit arbeiten alle Arbeitsprozesse gleichzeitig!

- Welche Realisierung von Puffern ist hier günstig?
  (Man überdenke eine Lösung mit einem Pipeline-Puffer der Kapazität 1!)

```
WHILE TRUE
  SEQ
    input ? x
    output ! x
```

Beachte: Wir müssten bei der Lösung des Election-Problems auch noch die Terminierungserkennung einbauen! (Gewinner erhält seine eigene Zahl zurück.)

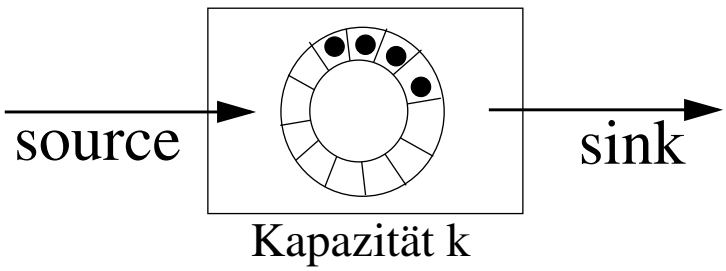
Ein Beispiel: Beschränkter zyklischer FIFO-Puffer in Occam



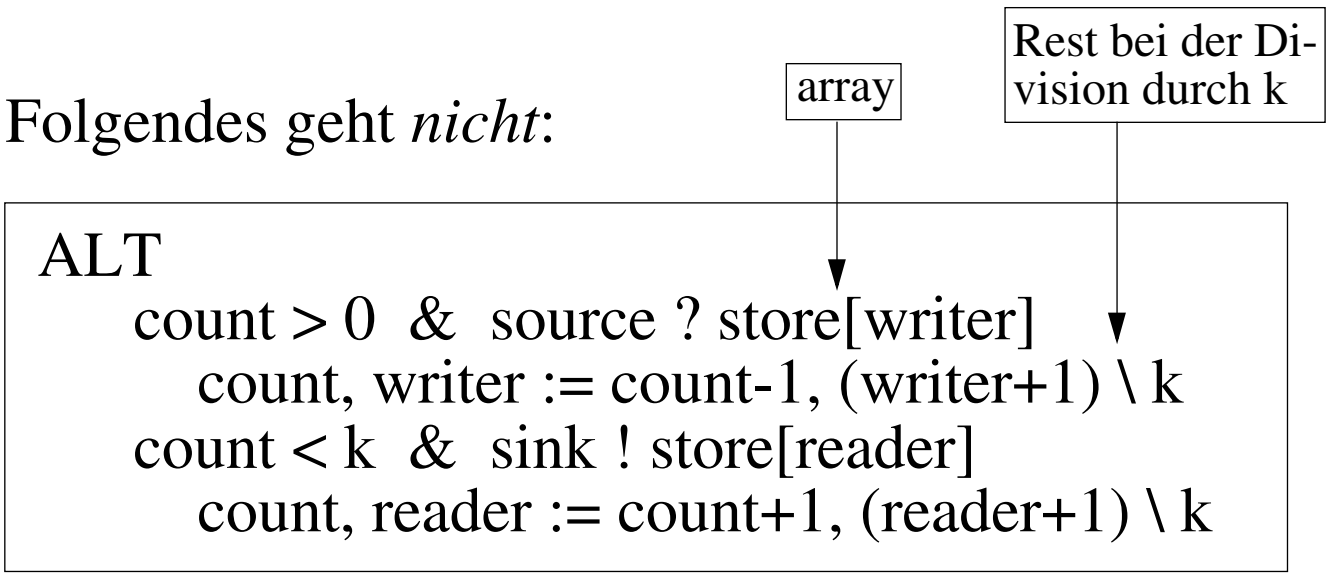
reader (vorderstes Element)

writer (nächster freier Platz)

count (Anzahl der freien Plätze)



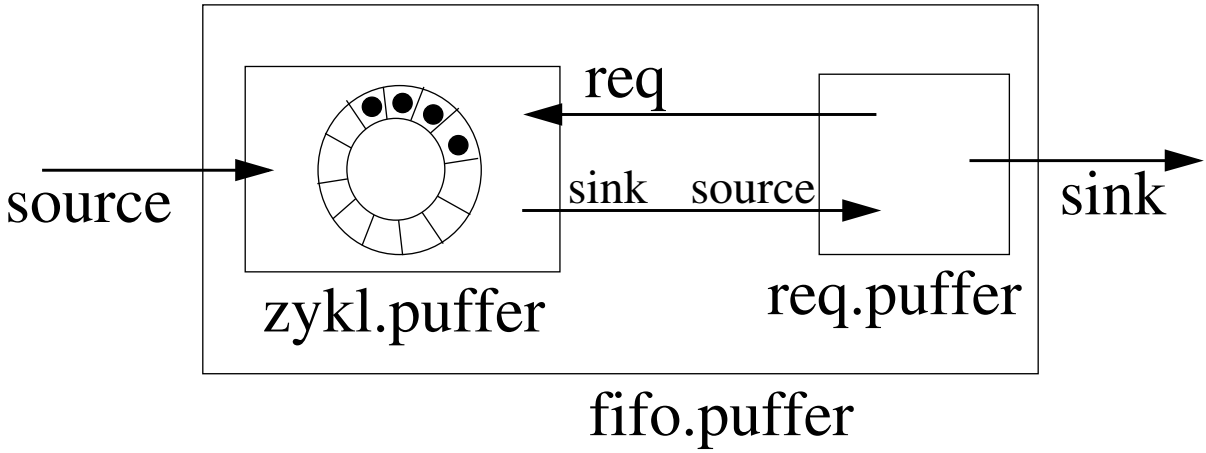
Folgendes geht *nicht*:



- Es gibt keine “output-guards” in Occam!
  - vielleicht automatisch in eines ohne output-guards transformieren?
- Lösung wie gehabt: Inversion der Kommunikation

Die Puffer-Architektur

- Idee: Inversion der Kommunikationsbeziehung für die Benutzer des Puffers transparent machen!



Prozedur

als Parameter

```
PROC  zykl.puffer (CHAN  OF  INT  source, sink, req)
    INT  reader, writer, count, x:
    [k] INT store:
    SEQ
        count, reader, writer := k, 0, 0
        WHILE  TRUE
        ALT
            count > 0  &  source ? store[writer]
               count, writer := count-1, (writer+1) \ k
            count < k  &  req ? x
               SEQ
                   sink ! store[reader]
                   count, reader := count+1, (reader+1) \ k
:
```

Deklaration des arrays

Variable x wird nicht verwendet

Passive Rolle (“Server”)

Puffer-Architektur (2)

Aktive Rolle (“Client”):

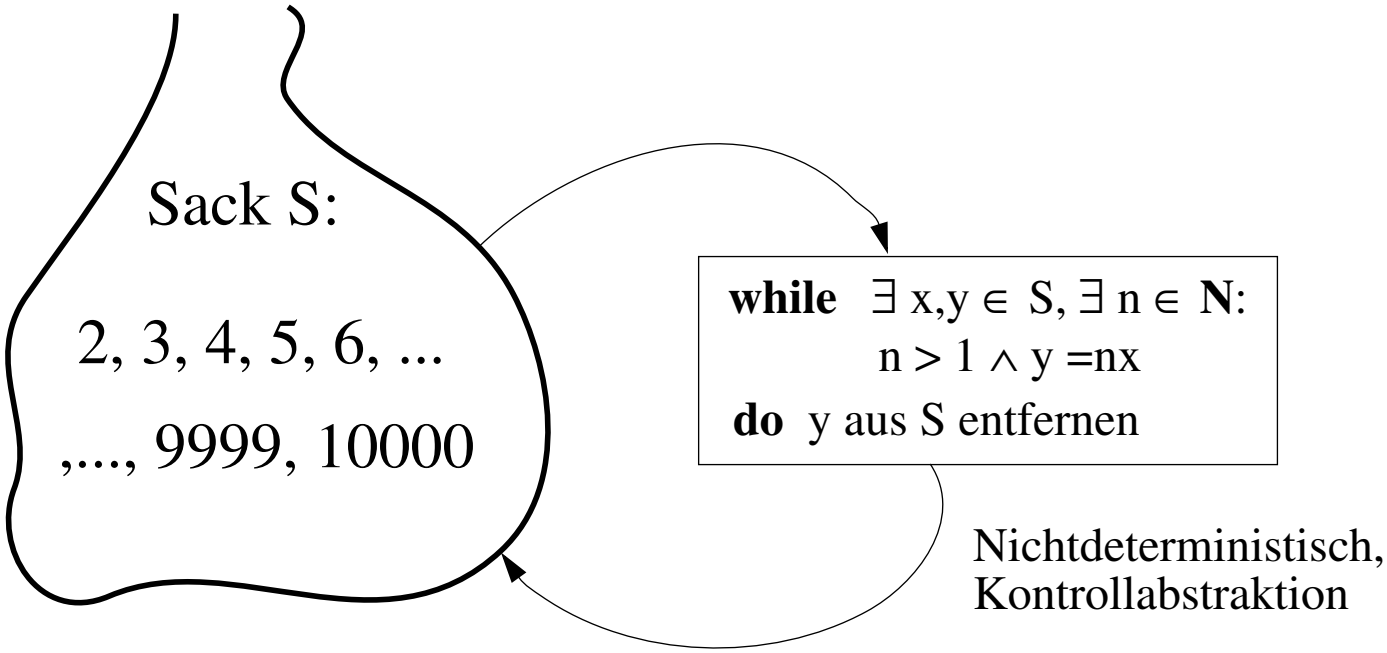
```
PROC req.puffer (CHAN OF INT source, sink, req)
  WHILE TRUE
    INT x:
    SEQ
      req ! 0
      source ? x
      sink ! x
:
```

Wieso sollte man besser *nicht parallel hierzu* schon den nächsten request an den zykl. Puffer senden?

```
PROC fifo.puffer (CHAN OF INT source, sink)
  CHAN OF INT req, intern:
  PAR
    zykl.puffer(source, intern, req)
    req.puffer(intern, sink, req)
:
```

- Vergleich von Pipeline-Puffer und dieser Lösung?
  - Aufwand
  - Verzögerungszeit (Abh. von der Kapazität k?)
  - Durchsatz
  - Anwendbarkeit (wann blockiert der Puffer seine Anwender?)

Naive Primzahlberechnung als Beispiel verteilter Berechnungen mit Occam



- *Implementierung*:

geschachtelte Schleife:

$\forall x:$

$\quad \forall y>x:$

$\quad\quad y/x \in \mathbf{N} \rightarrow y$ streichen

- *Realisierung* z.B. mittels Bit-array ("Indikatorleiste"):

2	3	~~4~~	5	~~6~~	7	8	~~9~~	~~10~~	11	~~12~~	13	~~14~~	~~15~~	16	17	~~18~~	19	~~20~~
1	1	0	1	0	1	0	0	0	1	0	1	0	0	0	1	0	1	0

--> *Sieb des Eratosthenes*

- Aber wie dies parallel bzw. verteilt lösen?

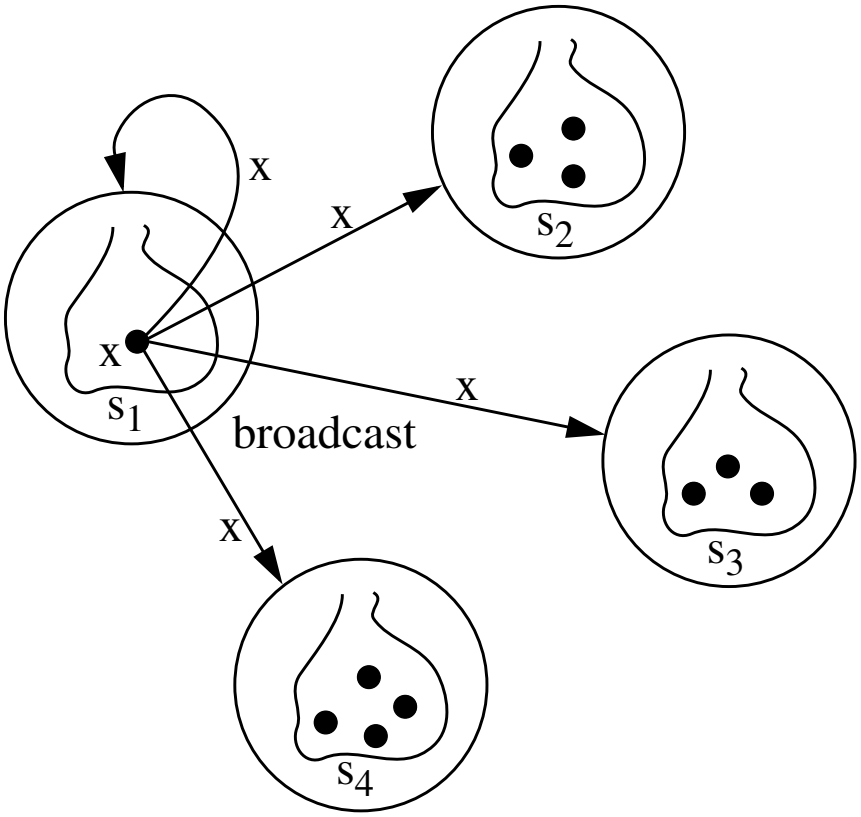
Sieb des Eratosthenes

Aus: “Nebenläufige Programme” von R. G. Herrtwich
und G. Hommel (Springer-Verlag)

Primzahlberechnung verteilt?

Idee: Sack S aufteilen in kleinere Säcke s_i: $S = \bigcup s_i$

Frage: Partition? ($i \neq j \Rightarrow s_i \cap s_j = \emptyset$)



- Nach und nach: Sende alle Werte aus s_i an alle anderen
- Bei Ankunft von x: Entferne alle Vielfachen von x aus dem lokalen Sack
- Nach Ende der Berechnung ggf. alle Säcke vereinigen
- Strategie (?): Vor dem Senden von x warten, ob nicht x noch (wg. eintreffender Nachrichten) gestrichen wird
- Zunächst vielleicht so viel wie möglich lokal streichen
- Wie S aufteilen? (Extremfall: einelementige Säcke)
- Speedup möglich?
- Wie das Ende der verteilten Berechnung feststellen?

Die Primzahl-Pipeline

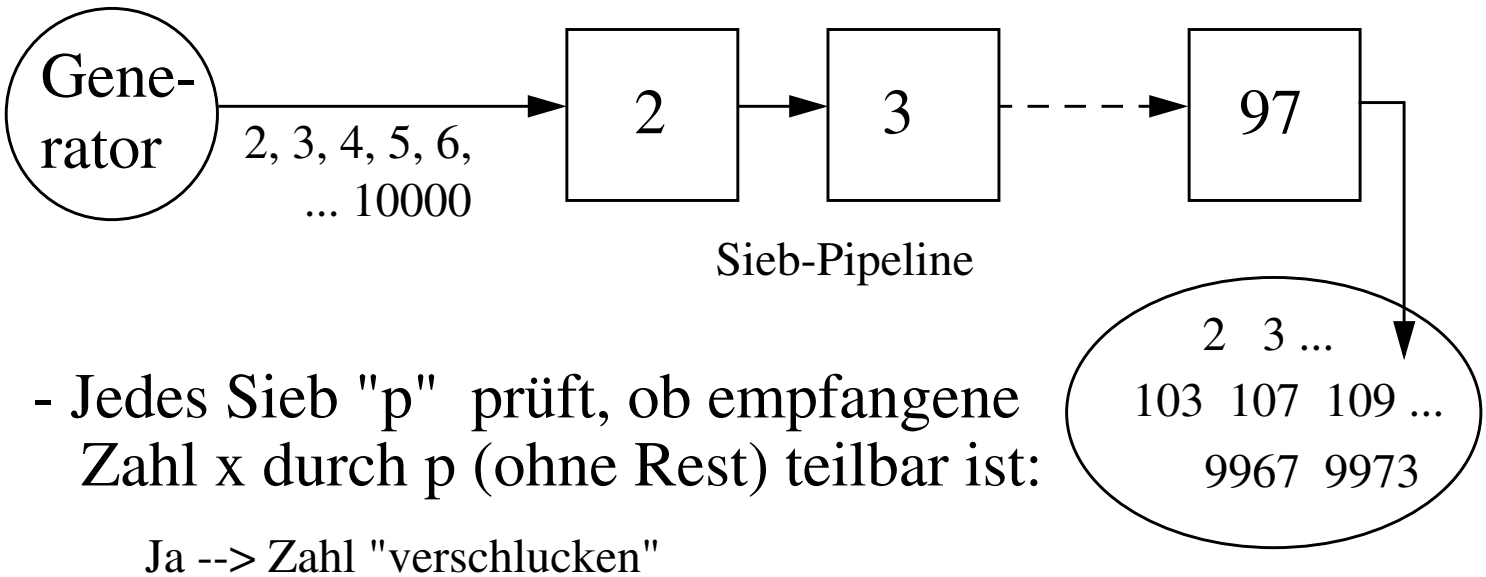
Aufgabe: Berechne alle Primzahlen bis 10000

Bsp.: Ist 127 Primzahl?

1) durch 2, 3, 4, 5, 6, 7, ..., 126 teilbar ? Nein --> prim

2) durch 2, 3, 4, 5, 6,. 7, ..., 13 teilbar? ($13^2 > 127$)

3) durch 2, 3, 5, 7, 11, 13 teilbar?
(--> Primzahlen $\leq \sqrt{127}$ genügen als “Testfaktoren”!)



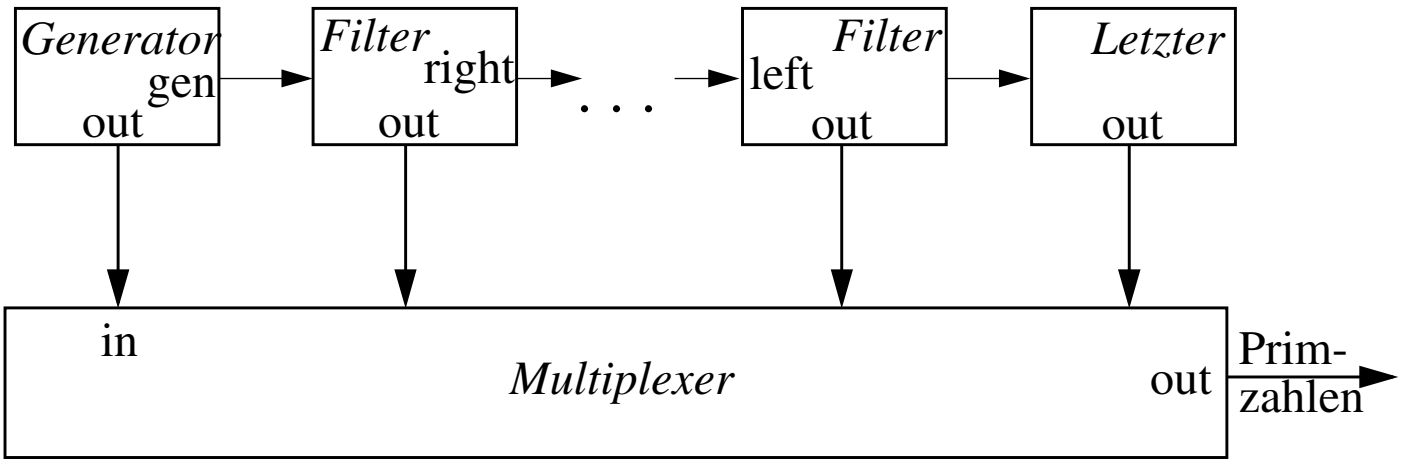
- Jedes Sieb "p" prüft, ob empfangene Zahl x durch p (ohne Rest) teilbar ist:

  Ja --> Zahl "verschlucken"

  Nein --> Zahl weiterreichen

- Pipeline ggf. dynamisch aufbauen: Hinten kommen nur Primzahlen heraus, die zur Verlängerung der Pipeline benutzt werden können

Primzahl-Pipeline




```
PROC  Generator (CHAN OF INT  gen, out)
    INT i:
    SEQ
        out ! 2
        i := 3
        WHILE  i <= 10000
            SEQ
                gen ! i
                i := i+2
        gen ! 0                     -- Schluss-Signal
:
```

Der Filter-Prozess

```
PROC  Filter (CHAN  OF  INT  left, right, out)
    INT  p, mp, x:
    SEQ
        left ? p          -- sei p <> 0
        out ! p
        mp := p*p     -- immer ungerade
        x := p
        WHILE  x<> 0
            SEQ
                left ? x
                WHILE  x > mp
                    mp := (p+p)+mp
        IF
            x < mp
                right ! x
            x = mp
                SKIP
:
```

Beachte: Es finden keine (zeitaufwendigen) Probedivisionen oder Multiplikationen statt!

Wird das Schlusssignal '0' durch alle Siebe propagiert?

-- jetzt x ≤ mp

-- verschlucken

Mit Puffer:

Ohne Puffer könnten Prozesse vorne in der Pipeline durch weiter hinten liegende gebremst werden!

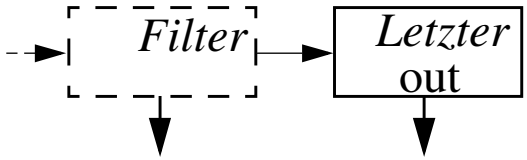
```
PAR
    IF
        ... ! ...
    left ? y
x := y
```

gleichzeitig schreiben und lesen

... sowie einige weitere offensichtliche Anpassungen

Die beiden restlichen Prozesse

```
PROC Letzter (CHAN OF INT left, out)
  INT x:
  SEQ
    x := 1
    WHILE x <> 0
      SEQ
        left ? x
        out ! x
```




```
PROC Multiplexer ([]CHAN OF INT in,
                  CHAN OF INT out)
INT p, i:
SEQ
  i := 0
  WHILE i <= Anzahl.Filter + 1
    SEQ
      in[i] ? p
      out ! p
      i := i+1
  WHILE p <> 0
    SEQ
      in[Anzahl.Filter + 1] ? p
      out ! p
```

Es müssen nun die Prozesse noch gegründet und konfiguriert werden (d.h. über die richtigen Kanäle miteinander verbunden werden und auf die vorhandenen Prozessoren geeignet verteilt werden; ggf. unter Beachtung von Lastausgleich).

Denkübung: Wie sollten 100 Filter auf 7 Prozessoren verteilt werden?

Übung (lehrreich!!): Dies z.B. in Java oder C realisieren, verteilt auf einigen Rechnern implementieren und den Speedup maximieren!