Übungsbeispiel zu Multicast (1)

(Nach einer Idee von Reinhard Schwarz)

Sinnvolle Definition von atomarem Multicast?

- **Lokale Totale Ordnung:** Wenn zwei Prozesse P_1 und P_2 beide die Nachrichten M und N empfangen mit Gruppe(M) = Gruppe(N), dann empfängt P_1 M vor N genau dann, wenn P_2 die Nachricht M vor N empfängt.
- **Paarweise Totale Ordnung:** Wenn zwei Prozesse P_1 und P_2 beide die Nachrichten M und N empfangen, dann empfängt P_1 M vor N genau dann, wenn P_2 die Nachricht M vor N empfängt.

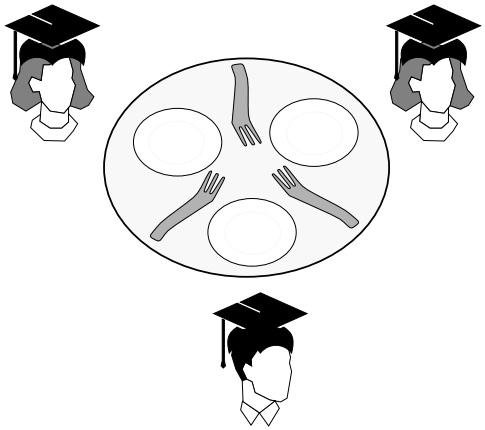
Fragen

- Wo braucht man solche Eigenschaften?
- Wann reichen die geforderten Eigenschaften, wann nicht?

Übungsbeispiel zu Multicast (2)

Beispiel: Problem der „speisenden Philosophen“

- Ein Philosoph **denkt** oder **speist**
- Zum Speisen benötigt ein Philosoph **rechte und linke Gabel.**
- Beim Denken gibt ein Philosoph beide Gabeln frei.



Wie stellt man sicher, dass die Philosophen nicht verhungern?

Übungsbeispiel zu Multicast (3)

Lösungsansatz:

Koordination der Gabelbenutzung per **paarweise atomarem Multicast**

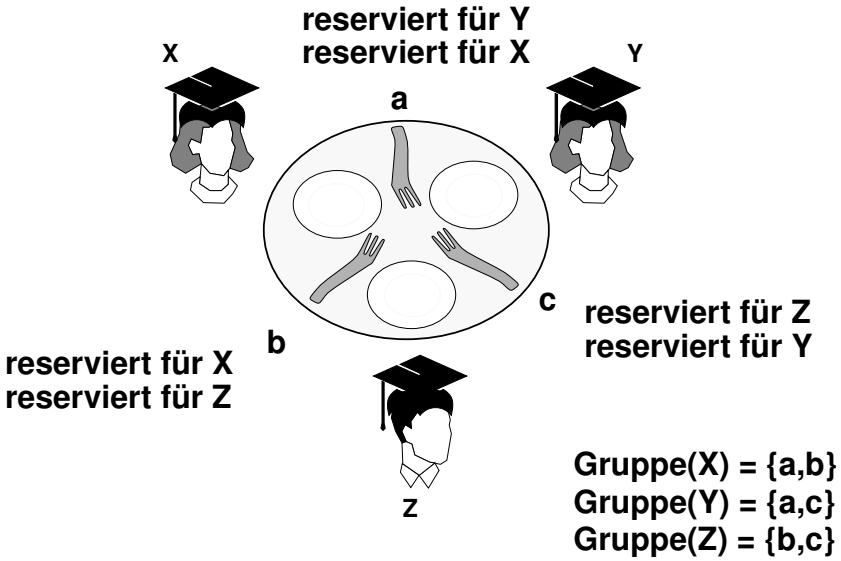
- Zum Essen sendet Philosoph X an seine benachbart liegenden Gabeln einen atomaren Multicast: *"reserviert für Philosoph X"*
- Gabel reserviert sich in der Reihenfolge der Anfragen.

→ **Durch paarweise totale Ordnung:**

- Reservierungen werden verklemmungsfrei vorgenommen, **oder?…**

Übungsbeispiel zu Multicast (4)

Typisches Szenario:



→ ***Reservierungen paarweise atomar:***
Kein Paar von Nachrichten von mehr als einer Gabel empfangen

→ ***Dennoch:*** Z wartet auf X, Z wartet auf Y, X wartet auf Z …
Deadlock!

Übungsbeispiel zu Multicast (5)

Atomarer Multicast – zweiter Versuch

wie bisher:

- Lokale Totale Ordnung: Wenn zwei Prozesse P_1 und P_2 beide die Nachrichten M und N empfangen mit Gruppe(M) = Gruppe(N), dann empfängt P_1 M vor N genau dann, wenn P_2 die Nachricht M vor N empfängt.
- Paarweise Totale Ordnung: Wenn zwei Prozesse P_1 und P_2 beide die Nachrichten M und N empfangen, dann empfängt P_1 M vor N genau dann, wenn P_2 die Nachricht M vor N empfängt.

neu:

- **Globale Totale Ordnung:** Paarweise Totale Ordnung +
  Wenn eine Nachricht *M* von einem Prozess P_1 vor Nachricht *N* ausgeliefert wurde und *N* von einem Prozess P_2 vor Nachricht *O*, so liefert kein Prozess die Nachricht *O* vor *M* aus (Transitivität der Ordnungsrelation).

Membership

- Was bedeutet "alle (korrekten) Gruppenmitglieder"?

  - *Beitritt* ("join") zu einer Gruppe während einer Übermittlung
  - *Austritt* ("leave") aus einer Gruppe während einer Übermittlung
  - *Wechsel zwischen "korrekt" und "fehlerhaft"* während einer Übermittlung

- Beachte:

  - "Zufälligkeiten" (z.B. Beitrittszeitpunkt kurz vor / nach dem Empfang einer Einzelnachricht) sollen vermieden werden (--> Nichtdeterminismus; Nicht-Reproduzierbarkeit)

- Folge:

  - Zu jedem Zeitpunkt muss *Übereinstimmung* über *Gruppenzusammensetzung* und *Zustand* aller Mitglieder erzielt werden

- Frage:

  - Wie erzielt man diese Übereinkunft?

Wechsel der Gruppenmitgliedschaft

"...während..." gibt es nicht (--> "virtuell synchron")

- Forderungen:

  - Eintritt und Austritt sollen *global atomar* erfolgen:
    - Die Gruppe muss mit allen (potentiellen) Sendern an die Gruppe über den Ein- und Austrittszeitpunkt jedes Gruppenmitglieds übereinstimmen
  - *globale Kausalität* soll gewahrt bleiben

- Realisierungsmöglichkeit:

  - konzeptuell führt jeder Prozess eine Liste mit den Namen aller Gruppenmitglieder
    - Realisierung als zentrale Liste (Fehlertoleranz und Performance?)
    - oder Realisierung als verteilte, replizierte Liste
  - massgeblich ist die zum Sendezeitpunkt gültige Mitgliederliste
  - Listenänderungen werden (virtuell) synchron durchgeführt:
    - bei zentraler Liste kein Problem
    - bei replizierten Listen: verwende *global kausalen, global atomaren Multicast*

      Schwierigkeit: *Bootstrapping-Problem* (mögliche Lösung: Service-Multicast zur dezentralen Mitgliedslistenverwaltung löst dies für sich selbst über einen zentralen Server)

Behandlung von Prozessausfällen

- Forderungen:

  - *Ausfall* eines Prozesses soll *global atomar* erfolgen:
    - Übereinstimmung über Ausfallzeitpunkt jedes Gruppenmitglieds
  - *Reintegration* nach einem vorübergehenden Ausfall soll *global atomar* erfolgen:
    - Übereinstimmung über Reintegrationszeitpunkt
  - *Globale Kausalität* soll gewahrt bleiben

- Realisierungsmöglichkeit:

  - Ausfallzeitpunkt:
    - Prozesse dürfen nur Fail-Stop-Verhalten zeigen: *"Einmal tot, immer tot"*
    - Gruppenmitglieder erklären Opfer per kausalem, atomarem Multicast übereinstimmend für tot: *"Ich sage tot – alle sagen tot!"*
    - Beachte: *"Lebendiges Begraben"* ist nicht ausschliessbar! (Irrtum eines "failure suspectors" aufgrund langsamer Nachrichten)
    - Fälschlich für tot erklärte Prozesse müssen unverzüglich Selbstmord begehen
  - Reintegration:
    - Jeder tote (bzw. für tot erklärte) Prozess kann der Gruppe nur nach dem offiziellen Verfahren ("Neuaufnahme") wieder beitreten

- Damit erfolgen Wechsel der Gruppenmitgliedschaft und Crashes in "geordneter Weise" für *alle* Teilnehmer

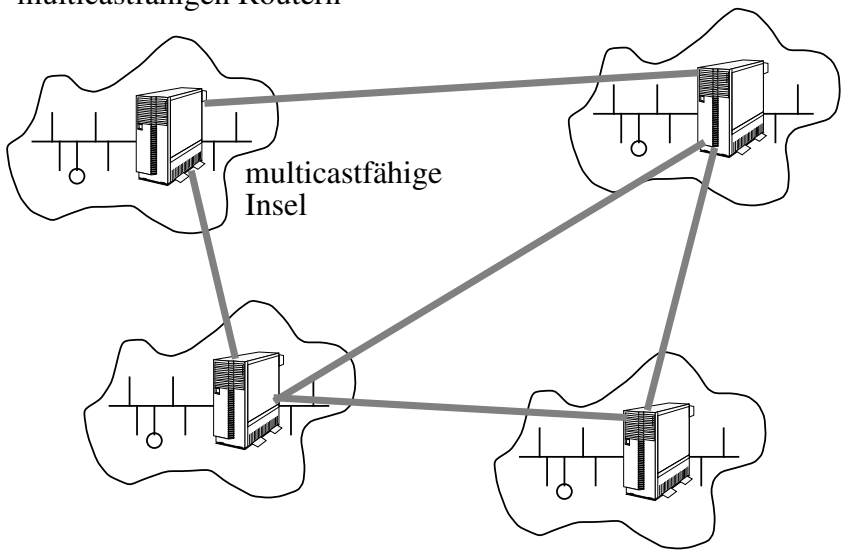
Multicast: Fazit

1. Wesentliche Neuerung gegenüber Broadcast:
   — Multicast-Gruppe
2. Gruppenüberlappungen schaffen Probleme:
   — lokale, paarweise oder globale Gültigkeit von Ordnungsbeziehungen?
3. Änderung der Gruppenmitgliedschaft ist kritisch:
   — Lösen des Membership-Problems
   — Lösen des Bootstrap-Problems
4. Das Tolerieren von Prozessausfällen ist schwierig:
   — erfordert geeignetes Fehlermodell (z.B. Fail-Stop)
   — fälschliches Toterklären nicht immer vermeidbar

- Anwendungen in der Praxis (Multi- und Broadcast):
  - gegenwärtig noch etwas zögerlich / rudimentär
  - Anwendungsgebiete: ausfallsichere Systeme, Telekom-Anwendungen, kooperatives Arbeiten...
  - es gibt Systeme, die Multicast-Kommunikationsprimitive enthalten
  - Sinnhaftigkeit und Notwendigkeit von Multicasts für grössere verteilte Anwendungen wird zunehmend erkannt
  - unsichere Multicasts im Internet zur (i.a. unidirektionalen) Verbreitung von Multimediadaten üblich (MBone)

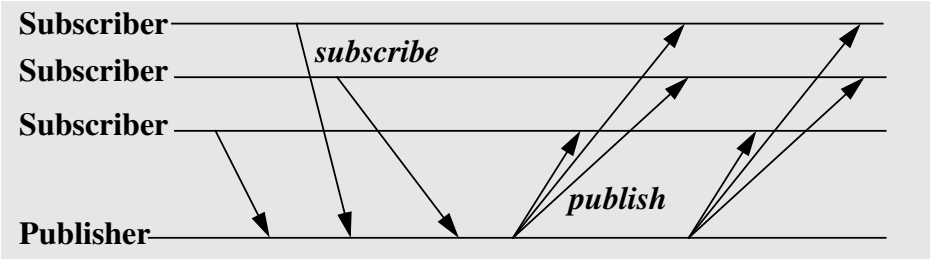
MBone (Multicast Backbone)

- Multicaststruktur im Internet zur (i.a. unidirektionalen) Verbreitung von Audio- und Videodaten
  - IP-basiert; Class-D-Multicastadressen (224.0.0.0 bis 239.255.255.255)
  - unsicher (möglicher IP-Paketverlust)
- Übertragung entlang eines Spannbaums
  - Nachrichten werden nicht mehrfach über die gleiche Strecke übertragen (nur ein Mal für alle Knoten im Unterbaum)
  - Kanten, die nicht mehr benötigt werden (z.B. bei Austritt aus der Gruppe) werden vom Spannbaum entfernt
- Router sollten multicastfähig sein
  - multicastfähige Inseln des Internet werden durch “Tunnel” miteinander verbunden
  - hierzu existiert ein MBone-Overlaynetz zwischen den multicastfähigen Routern



Push bzw. Publish & Subscribe

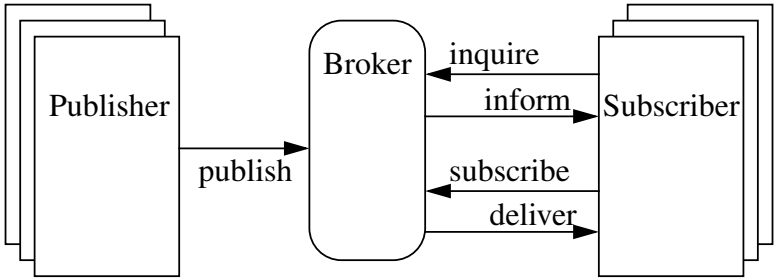
- Im Unterschied zum klassischen “Request / Reply-” bzw. “Pull-Paradigma”
  - wo Clients die gewünschte Information aktiv anfordern müssen
  - ein Client aber nicht weiss, ob bzw. wann sich eine Information geändert hat
  - dadurch periodische Nachfrage beim Server notwendig (“polling”)
- Subscriber (= Client) meldet sich für den Empfang der gewünschten Information an
  - z.B. “Abonnement” eines Informationskanals (“channel”)
  - i.a. existieren mehrere verschiedene Kanäle
  - u.U. auch dynamische, virtuelle Kanäle (--> “subject-based addressing”)



- Subscriber erhält automatisch (aktualisierte) Information, sobald diese zur Verfügung steht
  - “callback” des Subscribers (= Client) durch den Publisher (= Server)
  - push: “event driven” <--> pull: “demand driven”
  - Beispiel für push-Paradigma: Publikation aktueller Börsenkurse
- Für “publish” typischerweise Multicast einsetzen
  - “subscribe” entspricht offenbar einem “join” einer Multicast-Gruppe
  - Zeitverzögerung, Stärke der Multicast-Semantik und Grad an Fehlertoleranz wird oft als “Quality of Service” bezeichnet

Broker als Informationsvermittler

- Publisher und Subscriber müssen nicht direkt miteinander in Kontakt stehen
- Dazwischengeschalteter Broker verwaltet und vermittelt ggf. die Informationskanäle
  - i.a. mehrerer verschiedener Publisher
  - noch stärkere Entkopplung von Sender und Empfänger

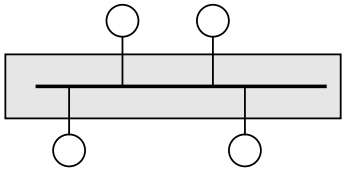


- Subscriber erfahren vom Broker, welche Kanäle abonniert werden können
  - Broker kann auch noch andere Dienste realisieren
- Subscriber melden sich beim Broker statt beim Publisher an
- Typische Rollenverteilung:
  - Publisher als *Produzent* von Information
  - Subscriber als *Konsument*

Ereigniskanäle für Software-Komponenten

- Stark entkoppelte Kommunikation bei komponenten-basierter Software-Architektur
  - Komponenten haben getrennte Lebenszyklen ← "hot pluggable" --> "spontaneous networking"
  - Entkoppelung fördert bessere Wiederverwendbarkeit und Wartbarkeit
  - anonym: Sender / Empfänger erfahren nichts über die Identität des anderen → oder sogar die Existenz
  - Auslösen von Ereignissen bei Sendern
  - Reagieren auf Ereignisse bei Empfängern
  - dazwischenliegende "third party objects" können Ereignisse speichern, filtern, umlenken...

- Ereigniskanal als "Softwarebus"
  - agiert als Zwischeninstanz und verknüpft die Komponenten
  - registriert Interessenten
  - Dispatching eingehender Ereignisse
  - ggf. Pufferung von Ereignisen



- Probleme
  - Ereignisse können "jederzeit" ausgelöst werden, von Empfängern aber i.a. nicht jederzeit entgegengenommen werden
  - falls Komponenten nicht lokal, sondern verteilt auf mehreren Rechnern liegen, die "üblichen" Probleme: verzögerte Meldung, ggf. verlorene Ereignisse, Multicastsemantik...

- Beispiele
  - Microsoft-Komponentenarchitektur DCOM / ActiveX / OLE
  - "Distributed Events" bei JavaBeans im Rahmen von Jini (event generator bzw. remote event listener; beliebige Objekte als "Parameter" möglich)
  - event service von CORBA (sprach- und plattformunabhängig.; typisierte und untypisierte Kanäle; Schnittstellen zur Administration von Kanälen; Semantik nicht genauer spezifiziert [z.B. Pufferung des Kanals])

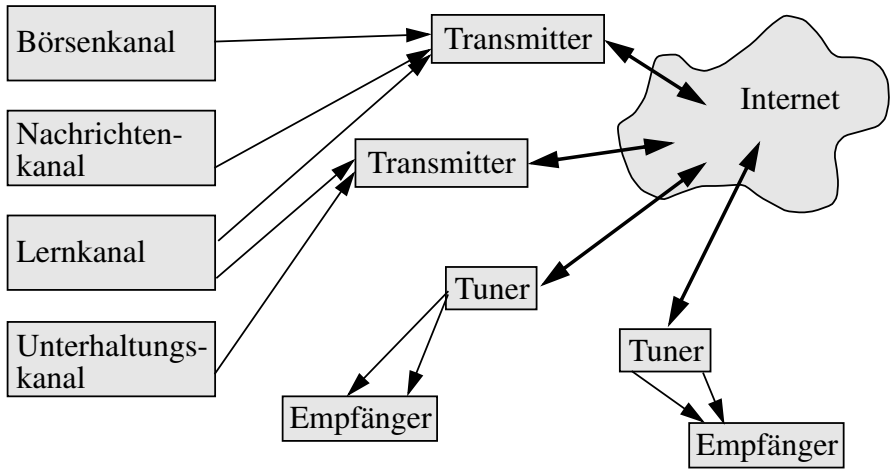
Push-Prinzip: Beispielprodukte

- "Information Bus" von TIBCO
  - publish / subscribe im Rahmen von Middleware
  - Motivation: Propagieren von "business events" (z.B. Finanzsektor)
  - Einsatzgebiete z.B. data monitoring oder real time decision support
  - "call back" einer Funktion beim Eintreffen einer Nachricht

- Pointcast (www.pointcast.com)
  - "broadcasting personalized news and information directly to your computer"

- CDF (Channel Definition Format) von Microsoft

- Castanet von Marimba (www.marimba.com)
  - ursprünglicher Zweck: "automatically distribute and maintain software applications and content within a company or across the Internet"
  - Gruppen von Kanälen; sichere (signierte, zertifizierte) Kanäle



- Identifizierung der Transmitter über eine URL
- Tuner erhält WWW-Seite vom angesprochenen Transmitter
- WWW-Seite enthält alle Kanäle eines Transmitters in Form von Links
- push wird simuliert durch periodisches polling (Subscriber kann Aktualisierungsrate festlegen)

Tupelräume

- Gemeinsam genutzter (“virtuell globaler”) Speicher
- Blackboard- oder Marktplatz-Modell
  - Daten können von beliebigen Teilnehmern eingefügt, gelesen und entfernt werden
  - relativ starke Entkopplung der Teilnehmer
- Tupel = geordnete Menge typisierter Datenwerte
- Entworfen von D. Gelernter (Yale University) für die Sprache Linda (ca. 1985)
- Operationen:
  - out (t): Einfügen eines Tupels t in den Tupelraum
  - in (t): Lesen und Löschen von t im Tupelraum
  - read (t): Lesen von t im Tupelraum
- Inhaltsadressiert (“Assoziativspeicher”)
  - Vorgabe eines Zugriffsmusters (bzw. “Suchmaske”) beim Lesen, damit Ermittlung der restlichen Datenwerte eines Tupels (“wild cards”)
  - Beispiel: int i,j; in(“Buchung”, ?i, ?j) liefert *ein* “passendes” Tupel
  - analog zu einigen Datenbankabfragesprachen (z.B. QbE)
- Synchrone und asynchrone Leseoperationen
  - ’ in’ und ‘read’ blockieren, bis ein passendes Tupel vorhanden ist
  - ‘inp’ und ‘readp’ blockieren nicht, sondern liefern als Prädikat (Daten vorhanden?) ‘wahr’ oder ‘falsch’ zurück

Tupelräume (2)

- Damit natürlich auch übliche Kommunikationsmuster realisierbar, z.B. Client-Server:

```
/* Client */
...
out(“Anfrage” client_Id, Parameterliste);
in(“Antwort”, client_Id, ?Ergebnisliste);
...
/* Server*/
...
while (true)
{  in(“Anfrage”, ?client_Id, ?Parameterliste);
   ...
   out(“Antwort”, client_Id, Ergebnisliste);
}
```

Beachte: Zuordnung des “richtigen” Clients über die client_Id

- Erweiterungen des Modells
  - *Persistenz* (Tupel bleiben nach Programmende erhalten, z.B. auf Platte)
  - *Transaktionseigenschaft* (wichtig, wenn mehrere Prozesse parallel auf den Tupelraum bzw. gleiche Tupel zugreifen)
- Problem: effiziente, skalierbare Implementierung?
  - *zentrale Lösung*: Engpass
  - *replizierter Tupelraum* (jeder Rechner enthält vollständige Kopie des Tupelraums; schnelle Zugriffe, jedoch hoher Synchronisationsaufwand)
  - *verteilter Tupelraum* (jeder Rechner hat einen Teil des Tupelraums; ‘out’-Operationen können z.B. lokal ausgeführt werden, ‘in’ ggf. mit Broadcast)
  - *fehlertolerante Lösung*? (z.B. Replikation mit kausal atomarem Broadcast)
- Problem: globaler Speicher ist der strukturierten Programmierung und der Verifikation abträglich
  - unüberschaubare potentielle Seiteneffekte
  - ggf. mehrere unabhängige Tupelräume

JavaSpaces

- Für Java, orientiert sich am Linda-Vorbild
  - gespeichert werden Objekte --> “Verhalten” neben den Daten
  - Tupel entspricht Gruppen von Objekten
- Teil der Jini-Infrastruktur für verteilte Java-Anwendungen
  - Kommunikation zwischen entfernten Objekten
  - Transport von Programmcode (“Verhalten”) vom Sender zum Empfänger
  - gemeinsame Nutzung von Objekten
- Operationen
  - *write*: mehrfache Anwendung erzeugt verschiedene Kopien
  - *read*
  - *readifexists*: blockiert (im Gegensatz zu read) nicht; liefert ggf. ‘null’
  - *takeifexists*
  - *notify*: Benachrichtigung (mittels eines Ereignisses), wenn ein passendes Objekt in den JavaSpace geschrieben wird
- Nutzen (neben Kommunikation)
  - atomarer Zugriff auf Objektgruppen
  - sicherer (“reliable”) verteilter Speicher
  - persistente Datenhaltung für Objekte

  aber keine Festlegung, ob eine Implementierung fehlertolerant ist und einen Crash überlebt
- *Implementierung*: könnte z.B. auf einer relationalen oder objektorientierten Datenbank beruhen
- *Semantik*: Reihenfolge der Wirkung von Operationen verschiedener Threads ist nicht festgelegt
  - selbst wenn ein write vor einem read beendet wird, muss read nicht notwendigerweise das lesen, was write geschrieben hat

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:

Schlüsselworte gross

```
SEQ
  PAR
    t1 := a      -- Das ist ein Kommentar
    t2 := b
    t3 := c
  x := t1        -- Diese 3 Zuweisungen
  y := t2        -- werden sequentiell
  z := t3        -- ausgeführt
```

Jeweils 2 Zeichen eingerückt

Jeweils auf eigene Zeile

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*

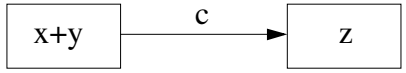
Occam: Kommunikation

- 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

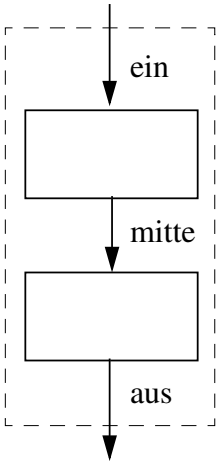
mit Kapazität 2

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
```

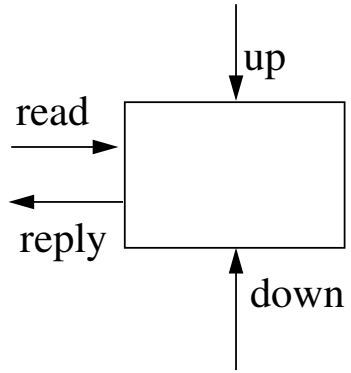
- Haben beide Lösungen die gleiche Semantik? (FIFO, keine unnötige Blockierung von Konsument oder Produzent...) Beweis?
- Vor-/ Nachteile der Implementierungen?
- Welche Realisierung ist besser?
- Lösungen skalierbar für mehr als zwei Pufferplätze?

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
```

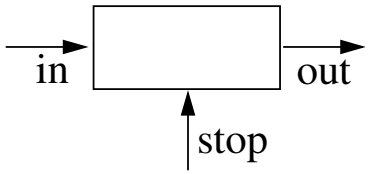


Beispiel zeigt, wie der wechselseitig ausgeschlossene Zugriff auf eine Variablen geregelt werden kann (*Monitor-Konzept*)

Oft sinnvoll: Umfassende WHILE-Schleife

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

"Ventil":



Occam: Guards

Beispiel "Stack":

Boole'scher Ausdruck

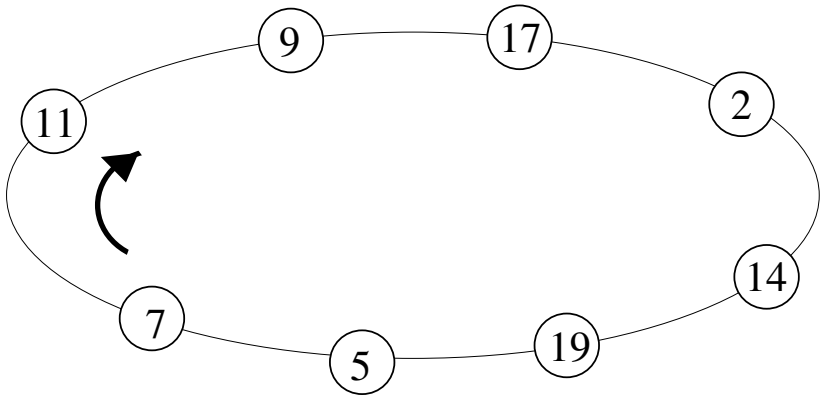
```
ALT
    (level <= max) & push ? Wert
        SEQ
            .....
    (level > 0) & pop ? req
        ....
```

- Spricht etwas dagegen, hier auch ein '!' ("output guard") anstelle eines '?' ("input guard") zuzulassen?

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

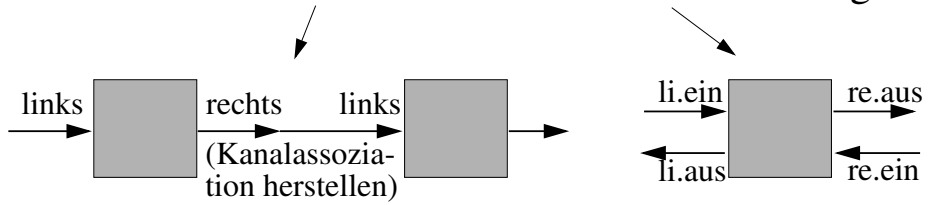
- Frage: Was ist die Semantik von:

```
WHILE TRUE
   SKIP
```

- Ist dies äquivalent zu STOP ?

Lösungsansatz Election-Algorithmus?

Unidirektionaler bzw. bidirektionaler Ring




```
...
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
```

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

So geht's auch:

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

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

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

- 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”

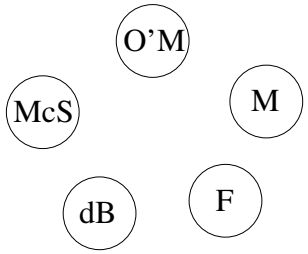
- Wie wäre folgender Ansatz?

```
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!

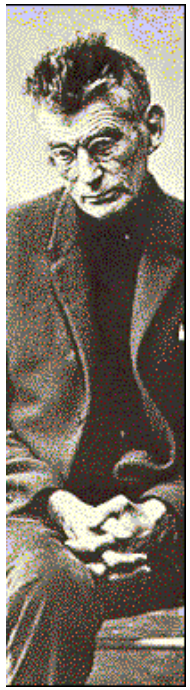


Samuel Beckett (1906-1989)**: Watt**

...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

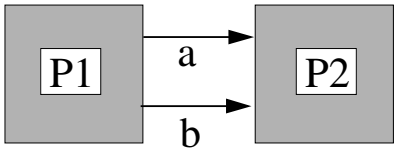
... ist in Occam nicht möglich!

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})

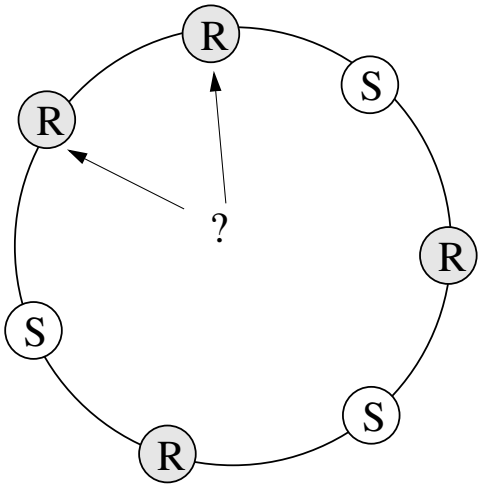
```
ALT
   W=0  &  a ? y
      SKIP
   W=1  &  b ? y
      SKIP
```

- 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 ...
```

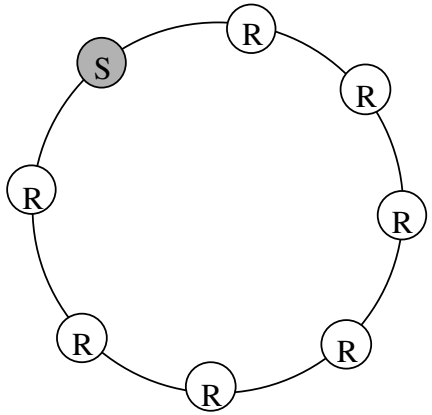
Nur bei R-Prozessen

Garantiert, dass jeder Prozess mindestens ein Mal sendet!

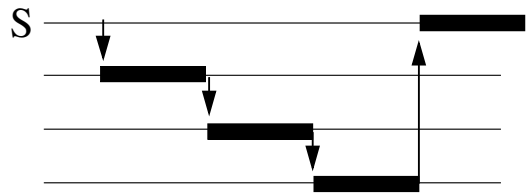
Frage: Welches zahlenmässige Verhältnis von S-/R-Prozessen ist günstig?

Variation der Anzahl der S/R-Prozesse

- Annahme: Senden am *Ende* einer Aktivitätsphase

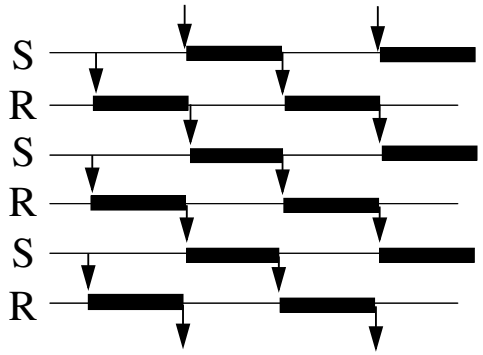


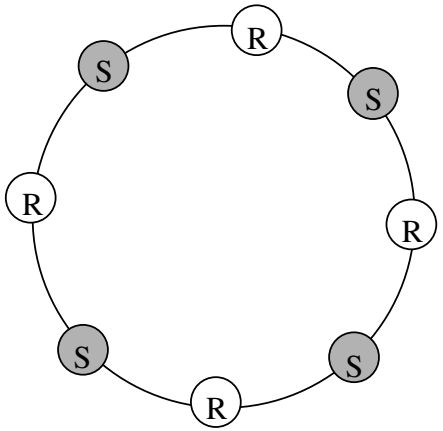
1) *Nur ein einziger Sende-Prozess*



Beobachtung: keine Parallelität (höchstens ein Prozess arbeitet)

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

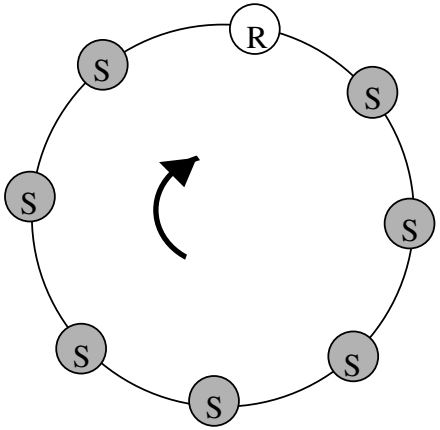




Beobachtung: etwa die Hälfte aller Prozesse ist gleichzeitig aktiv

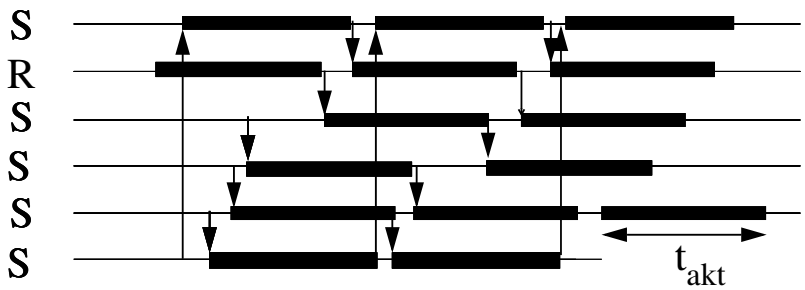
Geht es nicht besser?

3) *Nur ein einziger R-Prozess*



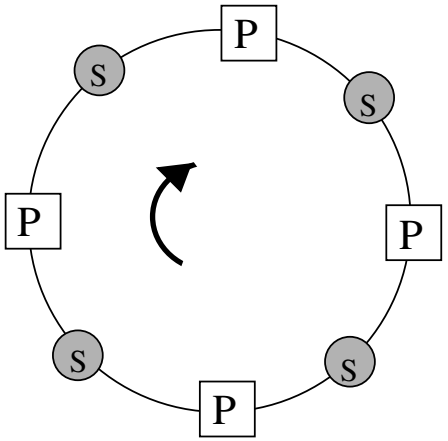
Beobachtung:
Empfangsbereitschaft läuft schnell (t_{trans}) in umgekehrter Richtung!

Bei langen Aktivitätsphasen ($t_{akt} > n\ t_{trans}$): fast immer n-1 Prozesse aktiv!



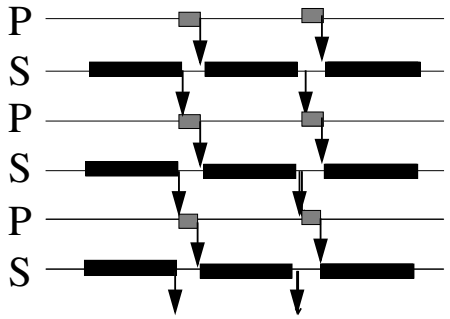
- Geht es noch besser?
  - sogar syntaktisch symmetrisch?

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

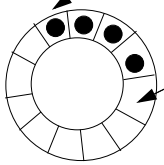


Abzüglich der Puffer-Verzögerungszeit arbeiten alle Arbeitsprozesse gleichzeitig!

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

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

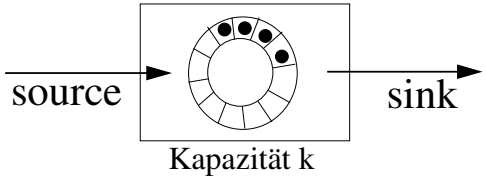
Ein Beispiel: Beschränkter zyklischer FIFO-Puffer in Occam



reader (vorderstes Element)

writer (nächster freier Platz)

count (Anzahl der freien Plätze)



Folgendes geht *nicht*:

array

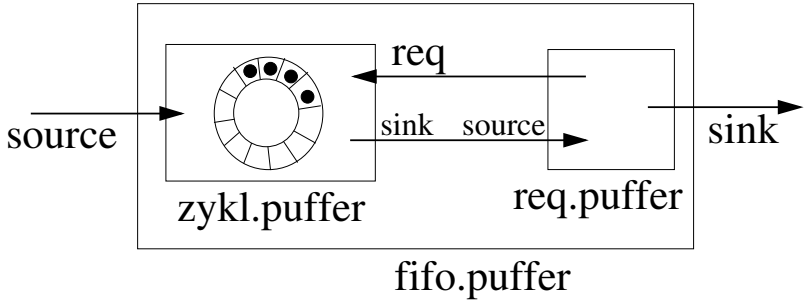
Rest bei der Division durch k

```
ALT
    count > 0  &  source ? store[writer]
       count, writer := count-1, (writer+1) \ k
    count < k  &  sink ! store[reader]
       count, reader := count+1, (reader+1) \ k
```

- 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"):

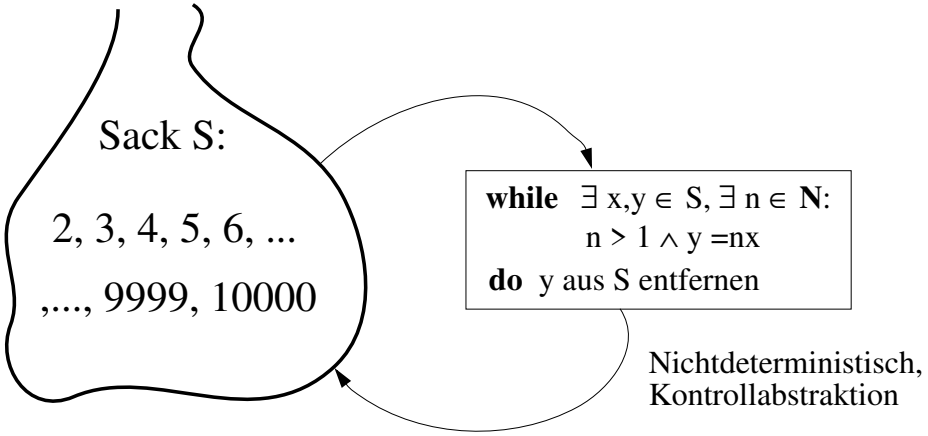
```
PROC req.puffer (CHAN OF INT source, sink, req)
  WHILE TRUE
    INT x:
    SEQ
      req ! 0
      source ? x
      sink ! x
:
```

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:

$\forall x$:

  $\forall y>x$:

    $y/x \in \mathbf{N}$ --> 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?

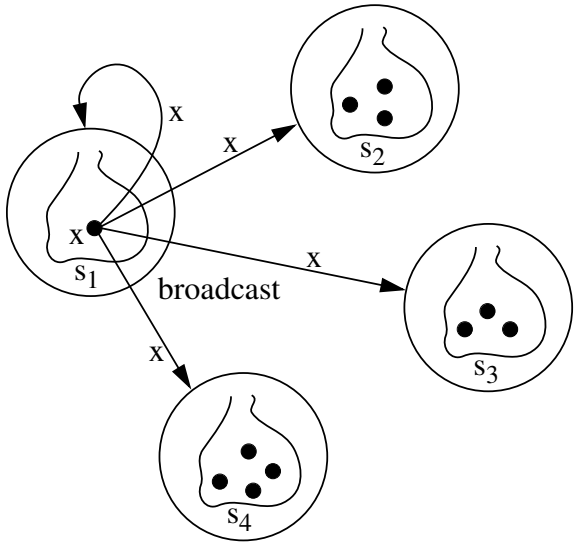
Sieb des Eratosthenes

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

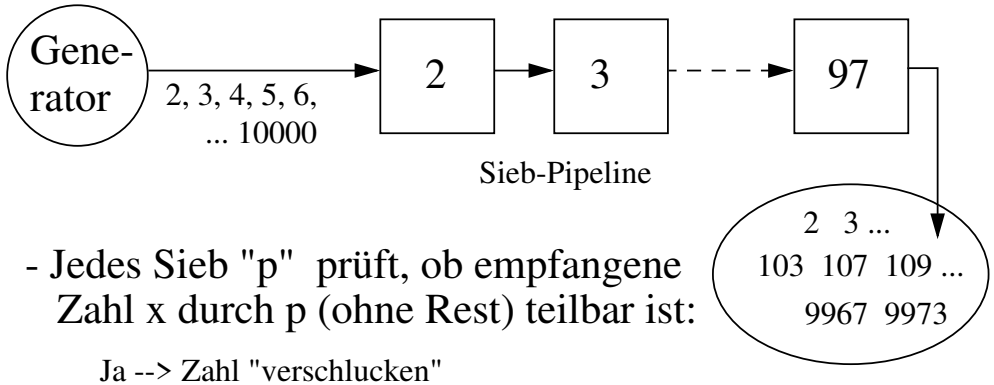
Aufgabe: Berechne alle Primzahlen bis 10000

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"!)



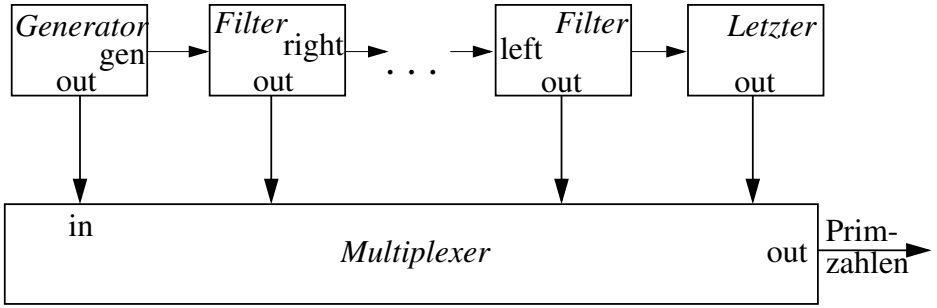
- Jedes Sieb "p" prüft, ob empfangene Zahl x durch p (ohne Rest) teilbar ist:

  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!

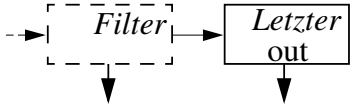
```
PAR
   IF
      ... ! ...
   left ? y
x := y
```

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 Prozessore verteilt werden?

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