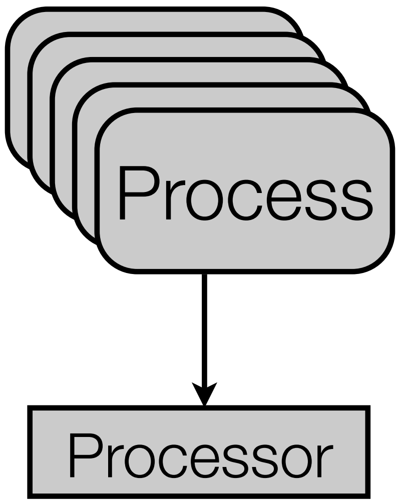
Parallel Programming Concepts

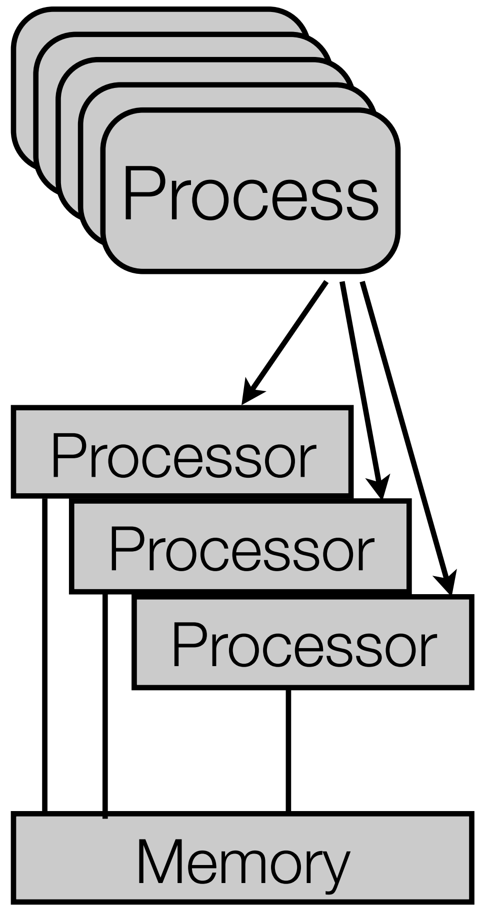
Theory of Concurrency - Multicomputer

Peter Tröger

Von Neumann Model



- Pipelining
- Super-scalar
- VLIW
- Branch prediction
- ...



- Processor executes a sequence of instructions, which specify
    - Arithmetic operation
    - Memory to be read / written
    - Address of next instruction
- Software layering tackles complexity of instruction stream
- Parallelism adds coordination problem between multiple instruction streams being executed

Terminology

- **Concurrency**
  - Supported to have two or more actions *in progress* at the same time
  - Classical operating system responsibility
    (resource sharing for better utilization of CPU, memory, network, ...)
  - Demands **scheduling** and **synchronization**
- **Parallelism**
  - Supported to have two or more actions executing *simultaneously*
  - Demands **parallel hardware**, **concurrency support**, (and **communication**)
  - Programming model relates to chosen hardware / communication approach
- Examples: Windows 3.1, threads, signal handlers, shared memory

History

- *1963: Co-Routines concept by Melvin Conway*
  - Foundation for message-based concurrency concepts
- *Late 1970‘s*
  - Parallel computing moved from shared memory towards multicomputers
  - Dijkstra / Hoare / Hansen worked on different according abstractions
- *1975, Concept of „recursive non-deterministic processes“ by Dijkstra*
  - Generator concept, foundation for Hoare‘s work on *Communicating Sequential Processes (CSP)*
- *1978, Distributed Processes: A Concurrent Programming Concept, B. Hansen*
  - Synchronized procedure called by one process and executed by another
  - Foundation for RPC variations in Ada and other languages

Co-Routines

- *Conway, Melvin E. (1963). "Design of a Separable Transition-Diagram Compiler". Communications of the ACM (New York, NY, USA: ACM) 6 (7): 396–408. doi:10.1145/366663.366704.*
    - Routines can suspend (yield) and resume in their execution
    - Co-routines that always yield new results are also called *generators*
    - Good for concurrent, not for parallel programming
    - Foundation for theoretical and practical message passing concepts
    - Broad language support today

```
var q := new queue
coroutine produce
    loop
        while q is not full
            create some new items
            add the items to q
        yield to consume
coroutine consume
    loop
        while q is not empty
            remove some items from q
            use the items
        yield to produce
```

Co-Routines

- Explicit language primitive to indicate transfer of control flow - **resume** primitive
- Example: Chess game - classical approach demands master procedure
- **detach** primitive: Return to control point that initially activated the co-routine
- Co-routines allow caller / callee model to be expressed in code

```
coroutine PLAYER1;
  initialize local variables
  detach
  while TRUE do
    make a move
    if game won then
      print message
      detach
    else
      resume(PLAYER2)
```

```
coroutine PLAYER2;
  initialize local variables
  detach
  while TRUE do
    make a move
    if game won then
      print message
      detach
    else
      resume(PLAYER1)
```

Communicating Sequential Processes

- Developed by Tony Hoare at University of Oxford from 1977
- Formal process algebra to describe concurrent systems
- Book: T. Hoare, Communicating Sequential Processes, 1985
- Basic idea
  - Computer **systems** act and interact with the environment continuously
    - Decomposition in **subsystems** (**processes**) which operate concurrently
    - **Interact** with other processes - subsystems or the environment
    - Modular approach
- Based on mathematical theory, described with algebraic laws
- Direct mapping to Occam programming language

CSP: Processes

- Behavior of real-world objects can be described through their interaction with other objects, leaving out internal implementation details
- Interface of a process is described as set of atomic events
- Event examples for an ATM:
  - *card* – insertion of a credit card in an ATM card slot
  - *money* – extraction of money from the ATM dispenser
- **Alphabet** - set of relevant (!) events for the description of an object
  - Event may never happen in the interaction
  - Interaction is restricted to this set of events
  - $\alpha_{ATM} = \{card, money\}$
- A CSP **process** is the behavior of an object, described with its alphabet

CSP: Processes

- **Event** is an atomic action without duration
  - Time is expressed with start/stop events, can overlap
  - Timing of events is not relevant for logical correctness, but ordering
  - Makes reasoning independent of execution speed and performance
- No concept of simultaneous events
  - May be represented as single event, if synchronization is modeled
- **$STOP_A$**
  - Process with alphabet A which never engages in any of the events of A
  - Expresses a non-working part of the system

CSP: Process Description through Prefix Notation

- **(x -> P)** **„x then P"**
  - x: event, P: process
  - Behavioral description of an object which first engages in x and than behaves as described with P
  - Prefix expression itself is a process (== behavior), chainable approach
  - **α(x -> P) = αP** - Processes must have the same alphabet
  - Example 1:
    (card -> $STOP_{\alpha ATM}$)
    „ATM which takes a credit card before breaking"
  - Quiz:
    „ATM which serves one customer and breaks while serving the second customer" - **αATM_Q={card, money}**

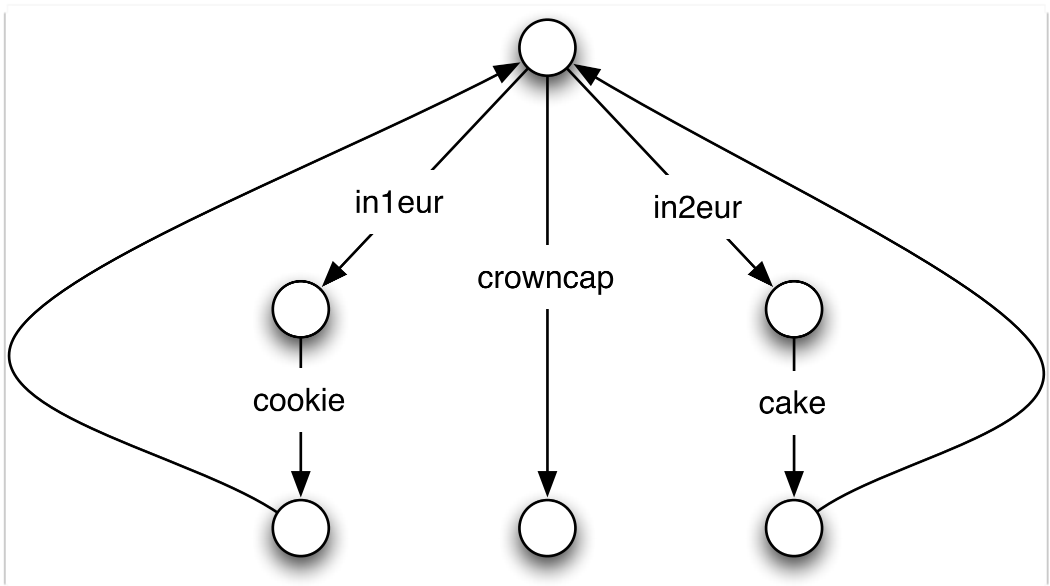
CSP: Recursion

- Prefix notation may lead to long chains of repetitive behavior for the complete lifetime of the object (until **STOP**)
  - Solution: Self-referential recursive definition for the object
- Example: An everlasting clock object
  $\boldsymbol{\alpha_{CLOCK} = \{tick\}}$
  CLOCK = (tick -> CLOCK)
- Enables description of an object with one single stream of behavior through prefixing and recursion

CSP Process Description - Choice

- Object behavior may be influenced by the environment
  - Support for multiple ‘behavior streams’ triggered by the environment
- Externally-triggered choice between two ore more events, leads to different subsequent behavior (== processes), forms a process by itself
  (x -> P | y -> Q)
- Example: Vending machine offers choice of slots for 1€ coin or 2€ coin
  VM = (in1eur -> (cookie -> VM) |
  in2eur -> (cake -> VM) | crowncap -> STOP)
- **|** is an operator on prefix expression, not on the processes itself

Process Description: Pictures



VM =

(in1eur -> (cookie -> VM) |

in2eur -> (cake -> VM) |

crowncap -> STOP)

- Single processes as circles, events as arrows
- Pictures may lead to problems - difficult to express equality, hard with large or infinite number of behaviors

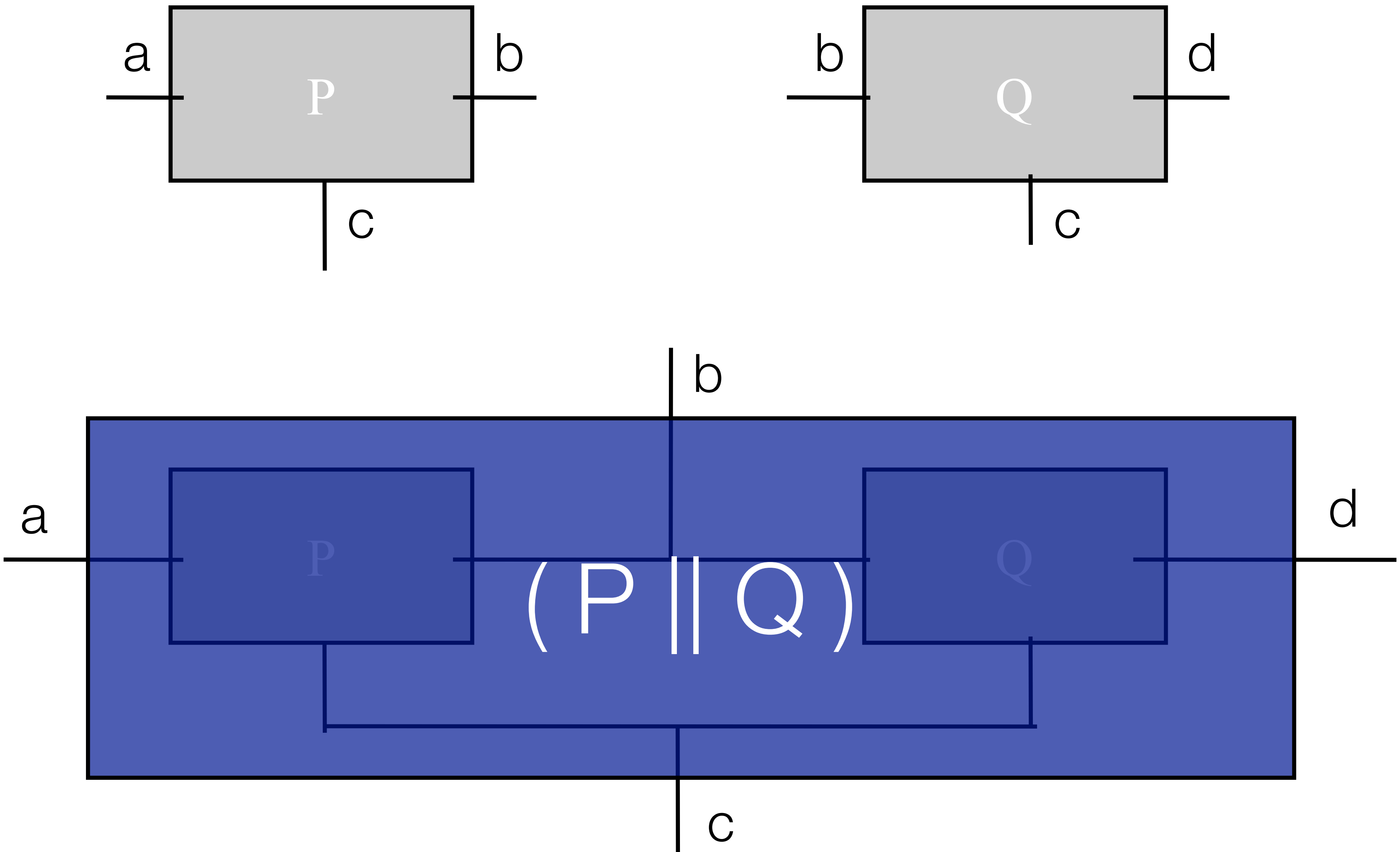
Concurrency in CSP

- Process = Description of possible behavior
- Set of occurring events depends on the environment, which may also be described as a process
- Allows to investigate a complete system, were the description is again a process
- Formal modelling of interacting processes
  - Formulate events that trigger simultaneous participation of multiple processes
- **Parallel combination**: Process which describes a system composed of the processes P and Q:

$$P \parallel Q \qquad\qquad \alpha(P \parallel Q) = \alpha P \cup \alpha Q$$

- **Interleaving**: Parallel activity with different events

Graphical Representation

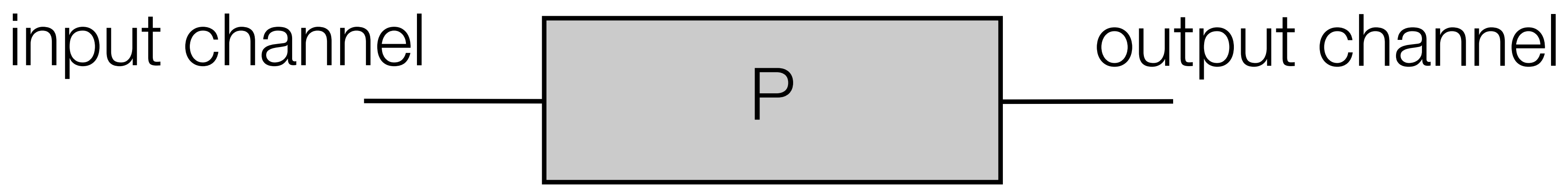


Communication in CSP

- Special class of event: communication
  - Modeled as uni-directional channel, only between two processes
  - Channel name is a member of the alphabets of both processes
  - Described by the events **c.v** which are part of the processes alphabet
    c: name of a channel on which communication takes place
    v: value of the message being passed
- Set of all messages which P can communicate on channel c:
  α c(P) = {v | c.v ε αP}
- **channel(c.v) = c, message(c.v) = v**
- Input choice: **(c?x -> P(x) | d?y -> Q(y))**

Communication (contd.)

- Process which first outputs v on the channel c and then behaves like P:
  (c!v -> P) = (c.v -> P)
- Process which is initially prepared to input any value x from the channel c and then behave like P(x):
  (c?x -> P(x)) = (y: {y | channel(y) = c} -> P(message(y)))



Communication (contd.)

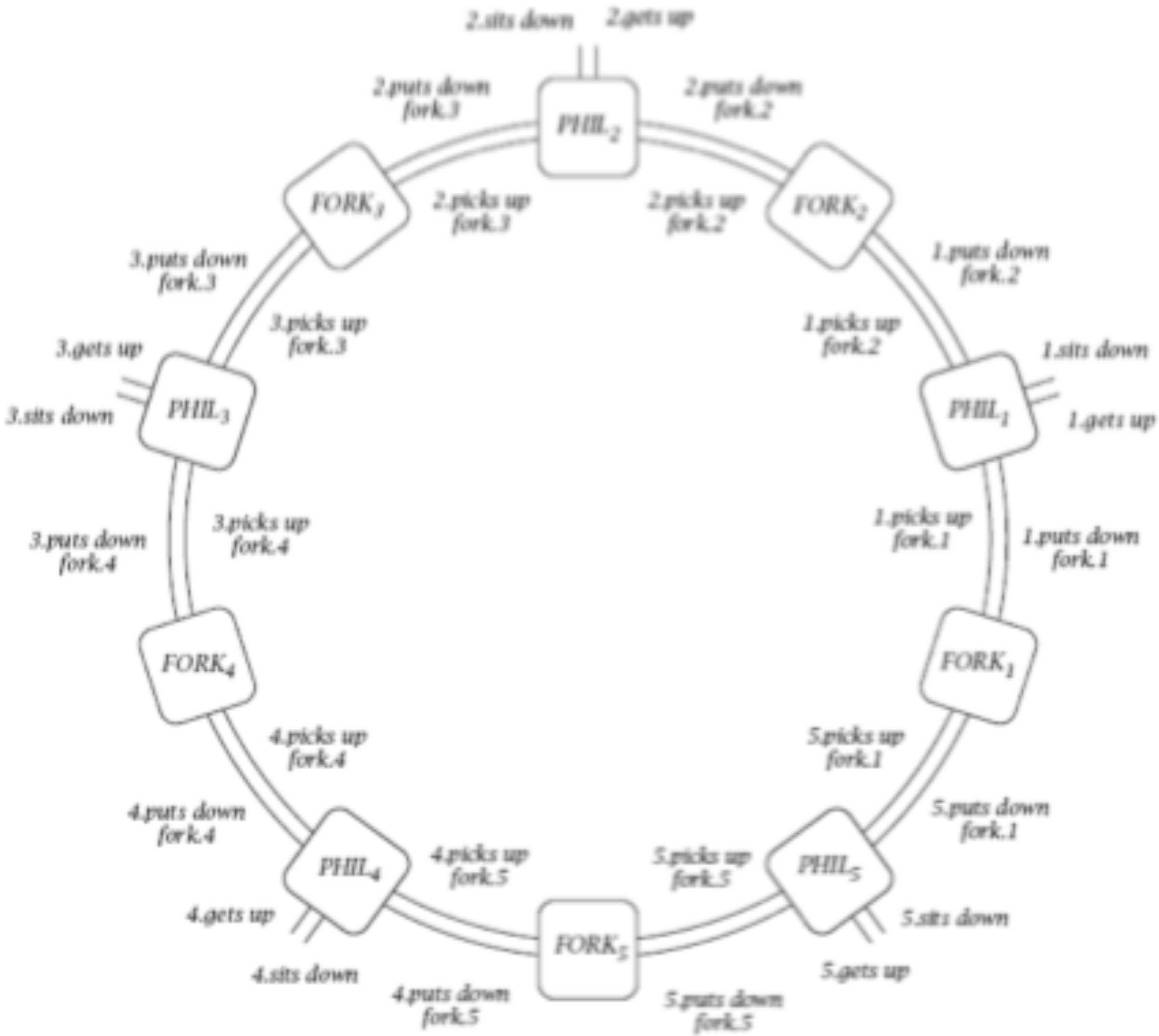
- Channel approach assumes **rendezvous behavior**
  - Sender and receiver block on the channel operation until the message was transmitted
  - Meanwhile common concept in messaging-based concurrency approaches
- When two concurrent processes communicate with each other only over a single channel, they cannot deadlock (see book)
- Network of non-stopping processes which is free of cycles cannot deadlock
  - Acyclic graph can be decomposed into subgraphs connected only by a single arrow
- Pipes: Processes with only one input and one output channel
- Join of two pipes P and Q : P>>Q

The Dining Philosophers (E.W.Dijkstra)

- Five philosophers work in a college, each philosopher has a room for thinking
- Common dining room, furnished with a circular table, surrounded by five labeled chairs
- In the center stood a large bowl of spaghetti, which was constantly replenished
- When a philosopher gets hungry:
  - Sits on his chair
  - Picks up his own fork on the left and plunges it in the spaghetti, then picks up the right fork
  - When finished he put down both forks and gets up
  - May wait for the availability of the second fork

Mathematical Model

- Philosophers: $\texttt{PHIL}_0$ … $\texttt{PHIL}_4$
- $\alpha\texttt{PHIL}_i$ = { i.sits down, i.gets up,
  i.picks up fork.i, i.picks up fork.(i⊕1),
  i.puts down fork.i, i.puts down fork.(i⊕1) }
- ⊕: Addition modulo 5 == i⊕1 is the right-hand neighbor of PHIL_i
- Alphabets of the philosophers are mutually disjoint, no interaction between them
- $\alpha\texttt{FORK}_i$ = { i.picks up fork.i,
  (i⊖1).picks up fork.i,
  i.puts down fork.i,
  (i⊖1).puts down fork.i }


2.sits down
2.gets up
2.puts down fork.3
PHIL2
2.puts down fork.2
FORK3
2.picks up fork.3
2.picks up fork.2
FORK2
3.puts down fork.3
1.puts down fork.2
3.picks up fork.3
1.picks up fork.2
3.gets up
1.sits down
PHIL3
PHIL1
3.sits down
1.gets up
3.puts down fork.4
3.picks up fork.4
1.picks up fork.1
1.puts down fork.1
FORK4
FORK1
4.picks up fork.4
5.picks up fork.1
4.puts down fork.4
5.puts down fork.1
PHIL4
4.picks up fork.5
5.picks up fork.5
PHIL5
4.gets up
5.sits down
FORK5
4.sits down
5.gets up
4.puts down fork.5
5.puts down fork.5

Behavior of the Philosophers

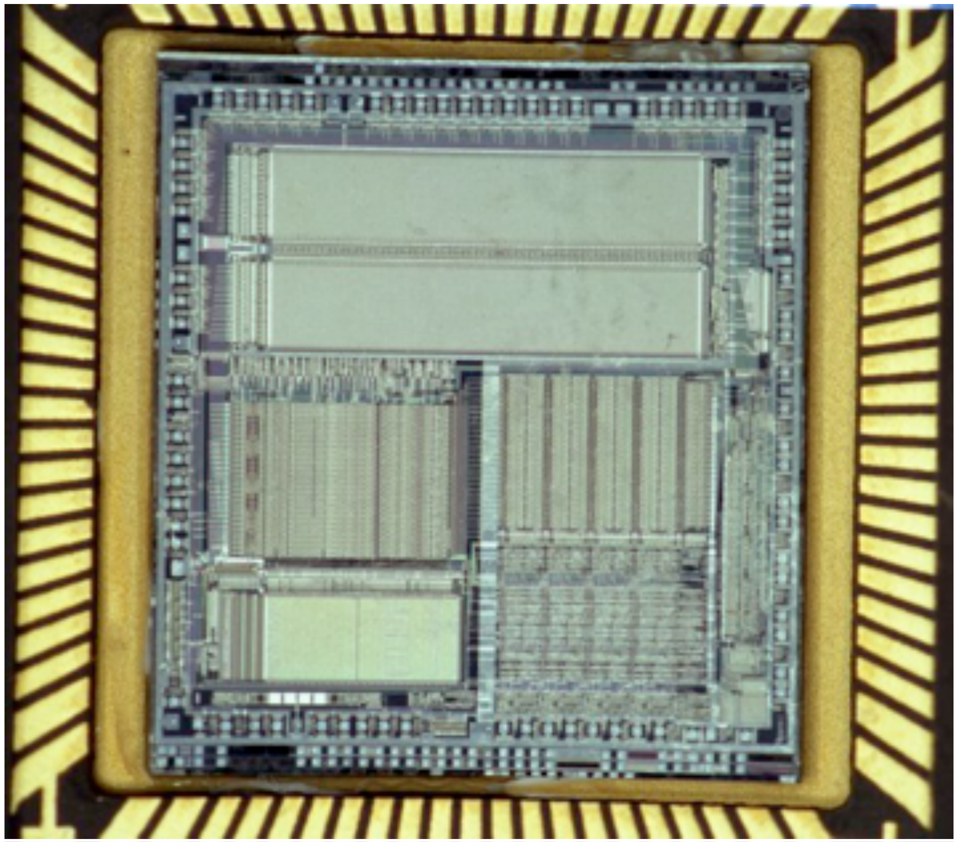
- $PHIL_i$ = (i.sits down ->
  i.picks up fork.i ->
  i.picks up fork.(i⊕1) ->
  i.puts down fork.i ->
  i.puts down fork.(i⊕1) ->
  i.gets up -> $PHIL_i$)
- $FORK_i$ = (i.picks up fork.i ->
  i.puts down fork.i -> $FORK_i$ |
  (i⊖1).picks up fork.i ->
  (i⊖1).puts down fork.i -> $FORK_i$)
- PHILOS=($PHIL_0$||$PHIL_1$||$PHIL_2$||$PHIL_3$||$PHIL_4$)
- FORKS=($FORK_0$||$FORK_1$||$FORK_2$||$FORK_3$||$FORK_4$)
- COLLEGE=(PHILOS||FORKS)

We leave out the proof here ;-) ...

What‘s the Deal ?

- Any possible system can be modeled through event chains
  - Enables mathematical proofs for deadlock freedom, based on the basic assumptions of the formalism (e.g. channel assumption)
- Some tools available (look at the CSP archive)
- CSP was the formal base for the Occam language
  - Language constructs follow the formalism, to keep proven properties
  - Mathematical reasoning about behavior of written code
- Still active research topics, channel concept adopted at several places
  - CSP channel implementation for Java, MPI design
  - Other formalism, e.g. Task / Channel model

Occam Example



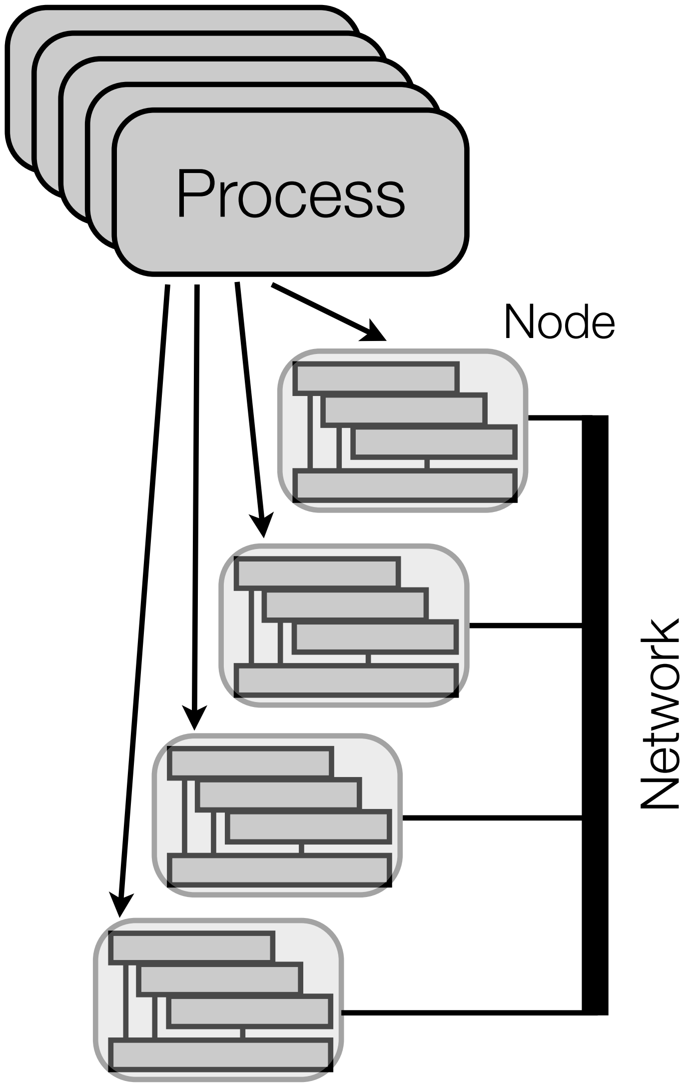
```
PROC producer (CHAN INT out!)
  INT x:
  SEQ
    x := 0
    WHILE TRUE
      SEQ
        out ! x
        x := x + 1
:

PROC consumer (CHAN INT in?)
  WHILE TRUE
    INT v:
    SEQ
      in ? v
      .. do something with `v'
:

PROC network ()
  CHAN INT c:
  PAR
    producer (c!)
    consumer (c?)
:
```

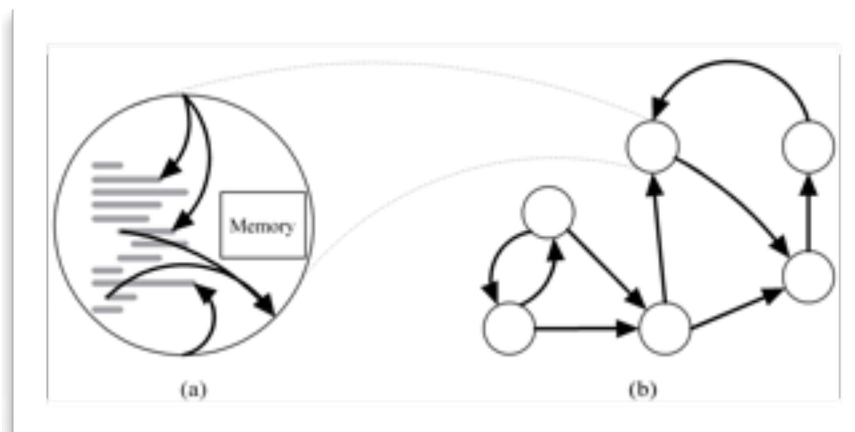
Task-Channel Model [Foster]

- **Computational model** for multi-computer case
- Parallel computation consists of one or more **tasks**
  - Tasks execute concurrently
  - Number of tasks can vary during execution
  - Task encapsulates **sequential program** with **local memory**
  - A task has **in-ports** and **outports** as interface to the environment
  - **Basic actions**: read / write local memory, send message on outport, receive message on in-port, create new task, terminate



Task-Channel Model [Foster]

- Outport / in-port pairs are connect by message queues called **channels**
  - Channels can be created and deleted
  - Channels can be referenced as **ports**, which can be part of a message
  - **Send** operation is asynchronous
  - **Receive** operation is synchronous
  - Messages in a channel stay in order
- Tasks are **mapped** to physical processors
  - Multiple tasks can be mapped to one processor
- Data locality is explicit part of the model
- Channels can model **control** and **data dependencies**



Task-Channel Model [Foster]

- Effects from channel-only interaction model
  - Performance optimization does not influence semantics
    - Example: Shared-memory channels for multiple tasks on one machine
  - Task mapping does not influence semantics
    - Align number of tasks to problem, not to execution environment
    - Improves scalability of implementation
  - Modular design with well-defined interfaces
  - Determinism made easy
    - Verify that each channel has a single sender and receiver

Task-Channel Model [Foster]

- Model results in some algorithmic style
  - Task graph algorithms, data-parallel algorithms, master-slave algorithms
- Theoretical performance assessment
  - Execution time: Period of time where at least one task is active
  - Number of communications / messages per task
- Rules of thumb
  - Communication operations should be balanced between tasks
  - Each task should only communicate with a small group of neighbors
  - Task should perform computations concurrently (task parallelism)
  - Task should perform communication concurrently

Actor Model

- *Carl Hewitt, Peter Bishop and Richard Steiger. A Universal Modular Actor Formalism for Artificial Intelligence IJCAI 1973.*
    - Another mathematical model for concurrent computation
    - No global system state concept (relationship to physics)
    - Actor as computation primitive, which can make local decisions, concurrently creates more actors, or concurrently sends / receives messages
    - Asynchronous one-way messaging with changing topology (CSP communication graph is fixed), no order guarantees
    - CSP relies on hierarchy of combined parallel processes, while actors rely only on message passing paradigm only
    - Recipient is identified by *mailing address*, can be part of a message
- „Everything is an actor“

Actor Model

- Influenced the development of the Pi-Calculus
- Serves as theoretical base to reason about concurrency, and as underlying theory for some programming languages (Erlang, Scala)
- Influences by Lisp, Simula, and Smalltalk
- Behavior as mathematical function - describes activity on message processing

Other Formalisms

- Lambda calculus by Alonzo Church (1930s)
  - Concept of procedural abstraction, originally via variable substitution
  - Functions as first-class citizen
  - Inspiration for concurrency through functional programming languages
- Petri Nets by Carl Adam Petri (since 1960s)
  - Mathematical model for concurrent systems
  - Directed bipartite graph with places and transitions
  - Huge vibrant research community
- Process algebra, trace theory, ...