9. Synchronous message passing

Processes communicate and synchronize directly, space is provided for **only one message** (instead of a channel).

**Operations:**

- **send (b):** blocks until the partner process is ready to receive the message
- **receive (v):** blocks until the partner process is ready to send a message.

When both sender and receiver processes are ready for the communication, the message is transferred, like an assignment \( v := b; \)

A send-receive-pair is both **data transfer and synchronization point**

**Origin:** Communicating Sequential Processes (CSP) [C.A.R. Hoare, CACM 21, 8, 1978]
Notations for synchronous message passing

Notation in CSP und Occam:

\[ p: \ldots q ! \text{ex} \ldots \text{send} \text{ the value of the expression \text{ex} to process } q \]

\[ q: \ldots p ? v \ldots \text{receive a value from process } p \text{ and assign it to variable } v \]

Multiple ports and composed messages may be used:

\[ p: \ldots q ! \text{Port1 (a1,\ldots,an)} \ldots \]
\[ q: \ldots p ? \text{Port1 (v1,\ldots,vn)} \ldots \]

Example: copy data from a producer to a consumer:

Prod: var p: int;
\hspace{1cm} do true -> p :=\ldots; Copy ! p od

Copy: var x: int;
\hspace{1cm} do true -> Prod ? x; Cons ! x od

Cons: var c: int;
\hspace{1cm} do true -> Copy ? c; \ldots od
Selective wait

Guarded command: (invented by E. W. Dijkstra)
a branch may be taken, if a condition is true and a communication is enabled (guard)

```
if Condition1; p ! x -> Statement1
[] Condition2; q ? y -> Statement2
[] Condition3; r ? z -> Statement3
fi
```

A communication statement in a guard yields
- **true**, if the partner process is ready to communicate
- **false**, if the partner process is terminated,
- **open** otherwise (process is not ready, not terminated)

Execution of a guarded command depends on the guards:
- If **some guards are true**, one of them is chosen,
  the communication and the branch statement are executed.
- If **all guards are false** the guarded command is completed without executing anything.
- **Otherwise** the process is blocked until one of the above cases holds.

Notation of an indexed selection:
- **if (i: 1..n) Condition; p[i] ? v -> Statements fi**
Guarded loops

A guarded loop repeats the execution of its guarded command until all guards yield false:

\[
\begin{align*}
\text{do} & \\
& \quad \text{Condition1; p ! x} \rightarrow \text{Statement1} \\
& \quad \text{[]} \text{Condition2; r ? z} \rightarrow \text{Statement2} \\
\text{od}
\end{align*}
\]

Example: bounded buffer:

process Buffer
  \[
  \begin{align*}
  & \text{do} \\
  & \quad \text{cnt < N; Prod ? buf[rear] \rightarrow cnt++; rear := rear \% N + 1;} \\
  & \quad \text{[]} \text{cnt > 0; Cons ! buf[front] \rightarrow cnt--; front := front \% N + 1;} \\
  \text{od}
  \end{align*}
  \]
end

process Prod
  \[
  \begin{align*}
  & \text{var p:=0: int;} \\
  & \text{do p<42; Buffer ! p \rightarrow p:=p+1;} \\
  \text{od}
  \end{align*}
  \]
end

process Cons
  \[
  \begin{align*}
  & \text{var c: int;} \\
  & \text{do Buffer ? c \rightarrow print c;} \\
  \text{od}
  \end{align*}
  \]
end
Prefix sums computed with synchronous messages

Synchronous communication provides both transfer of data and synchronization.

Necessary synchronization only (cf. synchronous barriers, PPJ-48)

\[
\begin{align*}
\text{const } N &:= 6; \text{ var } a [0:N-1] : \text{int}; \\
\text{process Worker (i := 0 to N-1)} &\text{ a process for each element} \\
&\text{ var } d := 1, \text{sum, new: int} \\
&\text{sum} := a[i]; \\
&\{\text{Invariant SUM: sum} = a[i-d+1] + \ldots + a[i]\} \\
&\text{do } d < N-1 \rightarrow \\
&\quad \text{if (i+d) < N }\rightarrow \text{ Worker(i+d) ! sum fi} \quad \text{shift old value to the right} \\
&\quad \text{if (i-d) }\geq 0\rightarrow \text{ Worker(i-d) ? new; sum} := \text{sum} + \text{new fi} \quad \text{get new value from the left} \\
&\quad d := 2*d \\
&\text{od} \\
&\text{end}
\end{align*}
\]

Why can deadlocks not occur?
No deadlocks in synchronous prefix sums

synchronization pattern

\[
\begin{array}{cccc}
0 & \cdots & \cdots & N-1 \\
\downarrow & \downarrow & \downarrow & \\
i-d & i & i+d & \\
j-d & j & \\
\end{array}
\]

- \textbf{! and ? operations occur always in pairs:}

  if \( i+d < N \) and \( i \geq 0 \), process \( i \) executes \( \text{Worker}(i+d)!\text{sum} \)

  let \( j = i+d \), i.e. \( j-d = i \geq 0 \), hence process \( j \) executes \( \text{Worker}(j-d)\text{?new} \)

- There is always a process that does \textbf{not send but receives:}

  Choose \( i \) such that \( i < N \) and \( i+d \geq N \), then process \( i \) only receives:

  Prove by induction.

- As \textbf{no process first receives and then sends}, there is \textbf{no deadlock}
Client/Server scheme with synchronous messages

Technique:
   for each kind of operation that the server offers, a communication via 2 ports:
   - oprReq for transfer of the parameters
   - oprRepl for transfer of the reply

Scheme of the client processes:
   process Client (I := 1 to N)
       . . .
       Server ! oprReq (myArgs)
       Server ? oprRepl (myRes)
       . . .
   end

Scheme of the server process:
   process Server ()
       . . .
       do (c: 1..N) ConditionOpr1; Client[c] ? oprReq(oprArgs)
           \--> process the request ...
           Client[c] ! oprRepl(oprResults)
           \] correspondingly for other operations ...
       od
   end
Synchronous Client/Server: variants and comparison

Synchronous servers have the
  same characteristics as asynchronous servers,
  i. e. active monitors (PPJ-70).

Variants of synchronous servers:
1. Extension to multiple instances of servers:
   use guarded command loops to check
   whether a communication is enabled
2. If an operation can not be executed immediately,
   it has to be delayed, and
   its arguments have to be stored in a pending queue
3. The reply port can be omitted if
   - there is no result returned, and
   - the request is never delayed
4. Special case: resource allocation with request and release.
5. Conversation sequences are executed in the part „process the request“.
   Conversation protocols are implemented by a
   sequence of send, receive, and guarded commands.
Synchronous messages in Occam

Occam:
• concurrent programming language, based on CSP
• initially developed in 1983 at INMOS Ltd. as native language for INMOS Transputer systems
• a program is a nested structure of parallel processes (PAR), sequential code blocks (SEQ), guarded commands (ALT), synchronous send (!) and receive (?) operations, procedures, imperative statement forms;
• communication via 1:1 channels
• fundamental data types, arrays, records
• extended 2006 to Occam-pi, University of Kent, GB


CHAN OF INT chn:
PAR
SEQ
   INT a:
   a := 42
   chn ! a

SEQ
   INT b:
   chn ? b
   b := b + 1

pi-calculus (Milner et. al, 1999):
formal process calculus where names of channels can be communicated via channels
Kent Retargetable occam Compiler (KRoC) (open source)
Bounded Buffer in Occam

CHAN OF Data in, out:

PAR

SEQ -- process buffer

Queue (k) buf:

Data d:

WHILE TRUE

ALT

in ? d & length(buf) < k

enqueue(buf, d)

out ! front(buf) & length(buf) > 0

! not allowed in a guard

dequeue(buf)

SEQ

-- only one producer process

Data d:

WHILE TRUE

SEQ

d = produce ()

in ! d

SEQ

-- only one consumer process

Data d:

WHILE TRUE

SEQ

out ? d

consume (d)
Synchronous rendezvous in Ada

Ada:

- **general purpose** programming language dedicated for **embedded systems**
- 1979: Jean Ichbiah at CII-Honeywell-Bull (Paris) wins a **competition** of language proposals initiated by the **US DoD**
- Ada 83 reference manual
- Ada 95 ISO Standard, including oo constructs
- Ada 2005, extensions
- **concurrency notions**: processes (**task, task type**), shared data, synchronous communication (**rendezvous**), entry operations pass data in both directions, guarded commands (**select, accept**)

```ada
task type Producer;
task body Producer is
d: Data;
begin
    loop
        d := produce ();
        Buffer.Put (d);
    end loop;
end Producer;

task type Consumer;
task body Consumer is
d: Data;
begin
    loop
        Buffer.Get (d);
        consume (d);
    end loop;
end Consumer;
```
Ada: Synchronous rendezvous

```
task type Buffer is -- interface
  entry Put (d: in Data); -- input port
  entry Get (d: out Data); -- output port
end Buffer;

task body Buffer is
  buf: Queue (k);
  d: Data;
begin
  loop
    select -- guarded command
      when length(buf) < k =>
        accept Put (d: in Data) do
          enqueue(buf, d);
        end Put;
      or
      when length(buf) > 0 =>
        accept Get (d: out Data) do
          d := front(buf);
        end Get;
        dequeue(buf);
    end select;
  end loop;
end Buffer;
```

```
task type Producer;

task body Producer is
  d: Data;
begin
  loop
    d := produce ();
    Buffer.Put (d);
  end loop;
end Producer;

task type Consumer;

task body Consumer is
  d: Data;
begin
  loop
    Buffer.Get (d);
    consume (d);
  end loop;
end Consumer;
```