

## 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]



### Lecture Parallel Programming WS 2014/2015 / Slide 87

**Objectives:**

Notions of synchronous message passing

**In the lecture:**

- Explain the operations.
- Compare with asynchronous messages.

**Questions:**

- Compare the notions of synchronous and asynchronous messages.

## Notations for synchronous message passing

**Notation** in CSP und Occam:

**p:** ... **q ! ex ...** **send** the value of the expression **ex** to process **q**

**q:** ... **p ? v ...** **receive** a value from process **p** and assign it to variable **v**

**multiple ports** and **composed messages** may be used:

**p:** ... **q ! Port1 (a1,...,an) ...**

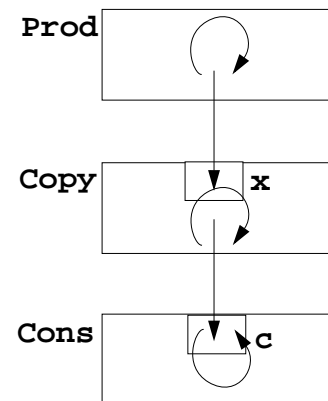
**q:** ... **p ? Port1 (v1,...,vn) ...**

**Example:** copy data from a producer to a consumer:

```
Prod:  var p: int;
       do true -> p :=...; Copy ! p od

Copy:  var x: int;
       do true -> Prod ? x; Cons ! x od

Cons:  var c: int;
       do true -> Copy ? c; ... od
```



## Lecture Parallel Programming WS 2014/2015 / Slide 88

### Objectives:

Notations of synchronous message passing

### In the lecture:

- Explain the notations.
- Synchronization without a reply channel.
- Example: copy process.

## 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
```

## Lecture Parallel Programming WS 2014/2015 / Slide 89

### Objectives:

Understand guards

### In the lecture:

- Guarded commands are needed to check whether a message is available without blocking the process.
- Explain the 3 states of a guard.
- Conditions are evaluated only once.

### Questions:

- Compare selective wait with the operations empty and receive-if-not-empty of asynchronous messages.

## Guarded loops

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

```
do
  Condition1; p ! x-> Statement1
[] Condition2; r ? z-> Statement2
od
```

**Example:** bounded buffer:

```
process Buffer
  do
    cnt < N; Prod ? buf[rear] -> cnt++; rear := rear % N + 1;
  [] cnt > 0; Cons ! buf[front] -> cnt--; front := front % N + 1;
  od
end

process Prod
  var p:=0: int;
  do p<42; Buffer ! p -> p:=p+1;
  od
end

process Cons
  var c: int;
  do Buffer ? c -> print c;
  od
end
```

## Lecture Parallel Programming WS 2014/2015 / Slide 90

### Objectives:

Understand guarded loops

### In the lecture:

Explain

- the example,
- mutual exclusion: process with synchronization points,
- condition synchronization: condition in a guard.

## Prefix sums computed with synchronous messages

Synchronous communication provides both **transfer of data and synchronization**.

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

```

const N := 6; var a [0:N-1] : int;

process Worker (i := 0 to N-1)           a process for each element
  var d := 1, sum, new: int

  sum := a[i];

                                     {Invariant SUM: sum = a[i-d+1] + ... + a[i]}
do d < N-1 ->
  if (i+d) < N -> Worker(i+d) ! sum fi   shift old value to the right
  if (i-d) >= 0 -> Worker(i-d) ? new; sum := sum + new fi
                                     get new value from the left
                                     double the distance
  d := 2*d
od
end                                     {SUM and d >= N-1}

```

Why can deadlocks not occur?

## Lecture Parallel Programming WS 2014/2015 / Slide 91

### Objectives:

See an application of synchronous messages

### In the lecture:

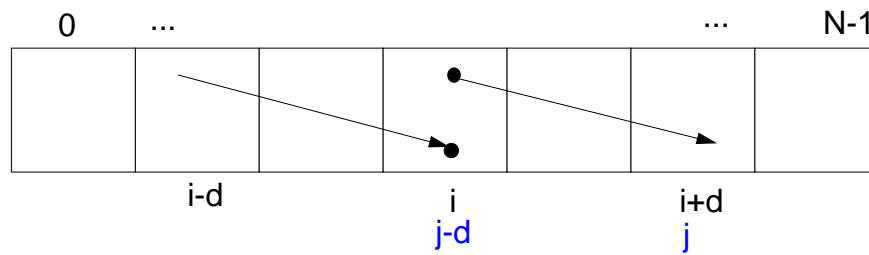
- Explain the communication graphically.
- Compare with asynchronous messages

### Questions:

- Why are programs based on synchronous messages more compact and less redundant than those with asynchronous messages?

## No deadlocks in synchronous prefix sums

synchronization pattern



- ! and ? operations occur always **in pairs**:

if  $i+d < N$  and  $i \geq 0$  process i executes `Worker(i+d)!sum`  
 let  $j = i+d$ , i.e.  $j-d = i \geq 0$ , hence process j executes `Worker(j-d)?new`

- There is always a process that does **not send but receives**:

Choose i such that  $i < N$  and  $i+d \geq N$ , then process i only receives:  
 Prove by induction.

- As **no process first receives and then sends**, there is **no deadlock**

## Lecture Parallel Programming WS 2014/2015 / Slide 92

### Objectives:

Deadlock proof for PPJ-91

### In the lecture:

Explain

- why absence of deadlocks is crucial,
- the steps of the proof.

## 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
```

## Lecture Parallel Programming WS 2014/2015 / Slide 93

### Objectives:

Understand the scheme

### In the lecture:

Explain the communication structure

### Questions:

- Describe a server for resource allocation in this scheme.

## 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.

## Lecture Parallel Programming WS 2014/2015 / Slide 94

### Objectives:

Understand the variants

### In the lecture:

Explain

- how pending requests are handled,
- when a channel can be omitted,
- how conversation sequences are handled,

Compare to active monitors.



## 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  
**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)

```
CHAN OF INT chn:
PAR
```

```
  SEQ
```

```
    INT a:
```

```
      a := 42
```

```
      chn ! a
```

```
  SEQ
```

```
    INT b:
```

```
      chn ? b
```

```
      b := b + 1
```

## Lecture Parallel Programming WS 2014/2015 / Slide 94a

### Objectives:

A brief introduction to Occam

### In the lecture:

- Occam: CSP-based language, standard language of Inmos Transputers
- parallel processes are program constructs (PAR)
- ? and !: send and receive as in CSP
- ALT: guarded command; (! not allowed in a guard)
- channels are here 1:1-links between processes for synchronous message passing
- indexed processes: PAR i=1 FOR n ...
- very restricted data types
- program structure by indentation

## 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)

```

## Lecture Parallel Programming WS 2014/2015 / Slide 94aa

### Objectives:

Bounded buffer in Occam

### In the lecture:

Explain

- program structure: 3 processes
- ALT: guarded command; (! not allowed in a guard)
- ? and !: send and receive as in CSP
- 2 channels between producer, consumer, and buffer

## 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**)

```

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;

```

## Lecture Parallel Programming WS 2014/2015 / Slide 94b

### Objectives:

Brief introduction to Ada

### In the lecture:

Explain

- Ada: general purpose language, in particular suitable for embedded systems
- processes are defined as tasks; task types for several processes of the same type
- communicate synchronously by rendezvous: bi-directional communication operation
- parameters may be passed in either direction (call-by-value-and-result)

## 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);
          dequeue(buf);
        end Get;
    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;

```

### Lecture Parallel Programming WS 2014/2015 / Slide 94ba

#### Objectives:

Bounded buffer using Ada rendezvous

#### In the lecture:

Explain

- task declares communication interface: entries
- entries are called by other tasks
- parameters may be passed in either direction (call-by-value-and-result)
- each entry has several accept-statements (communication operation) in the task body
- select is a guarded command
- one-sided anonymous: the task does not know who calls its entry