



Concurrent Object-Oriented Programming

Bertrand Meyer, Volkan Arslan



Lecture 7: Classic Approaches to Concurrent Programming Message-based synchronisation and communication

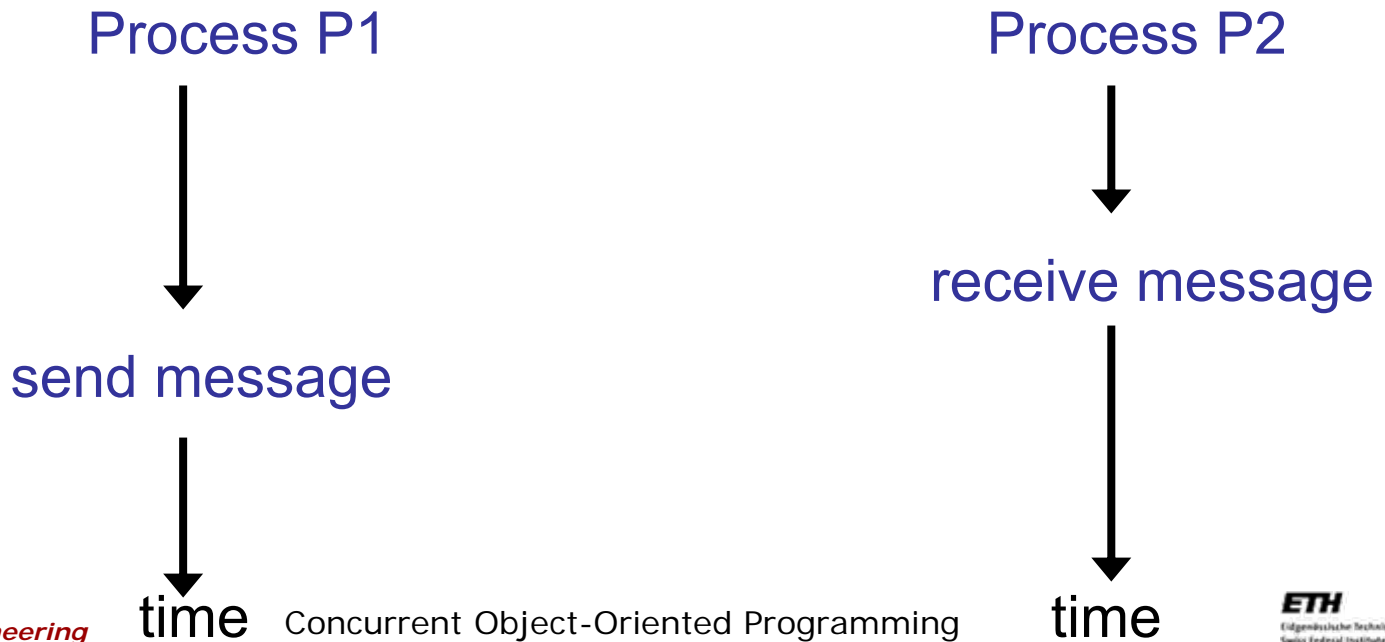


Message-based synchronization and communication

- Process synchronization
- Process naming and message structure
- Message-passing semantics of Ada and occam2
- Selective waiting



- Alternative to shared variable synchronisation and communication is based on **message passing**
- Use of a **single construct** for both synchronisation and communication
- Three issues:
 - the model of synchronisation
 - the method of process naming
 - the message structure





- There is the **implicit** synchronisation with all message-based systems, that a **receiver process** cannot obtain a message **before that message has been sent**.
- This is not the case with shared variable; a **receiver process can read a variable** and **not know whether** it has been written to by the sender process.

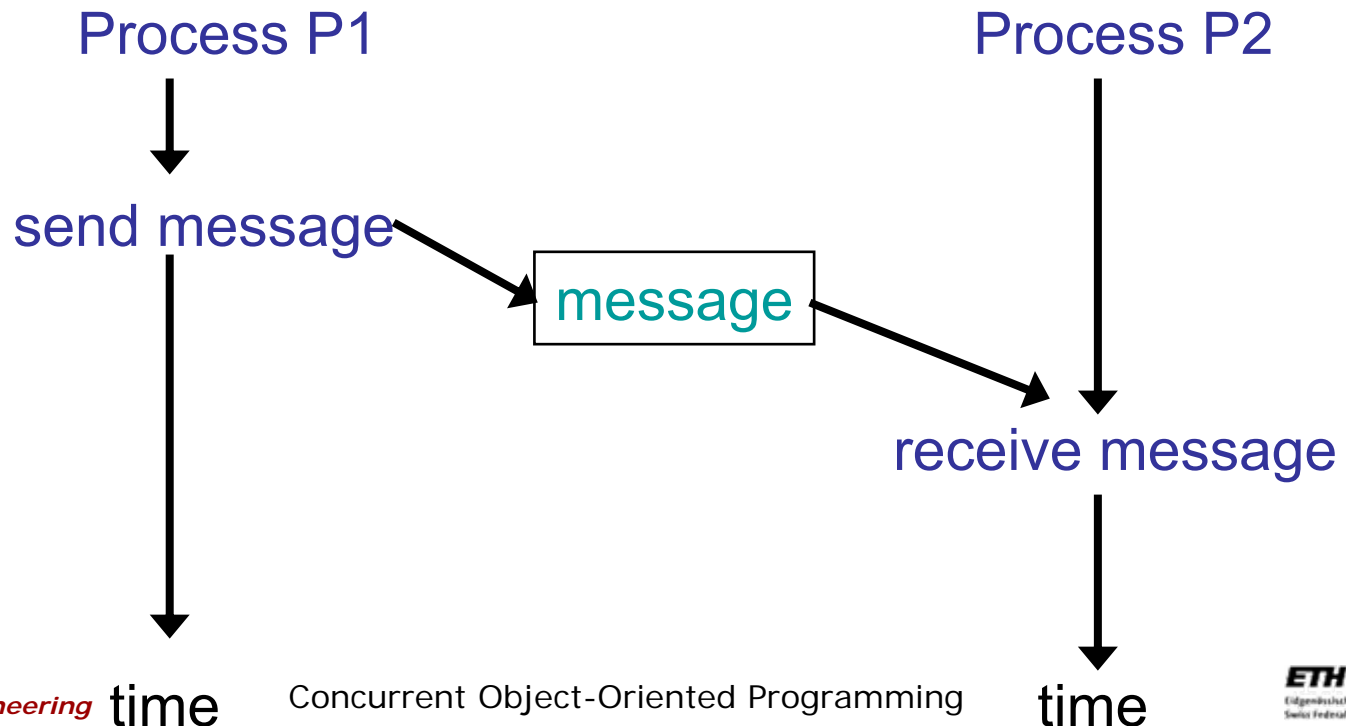


Variations in the process synchronisation model arise from the semantics of the **send** operation

- Asynchronous (or no-wait)
- Synchronous
- Remote invocation



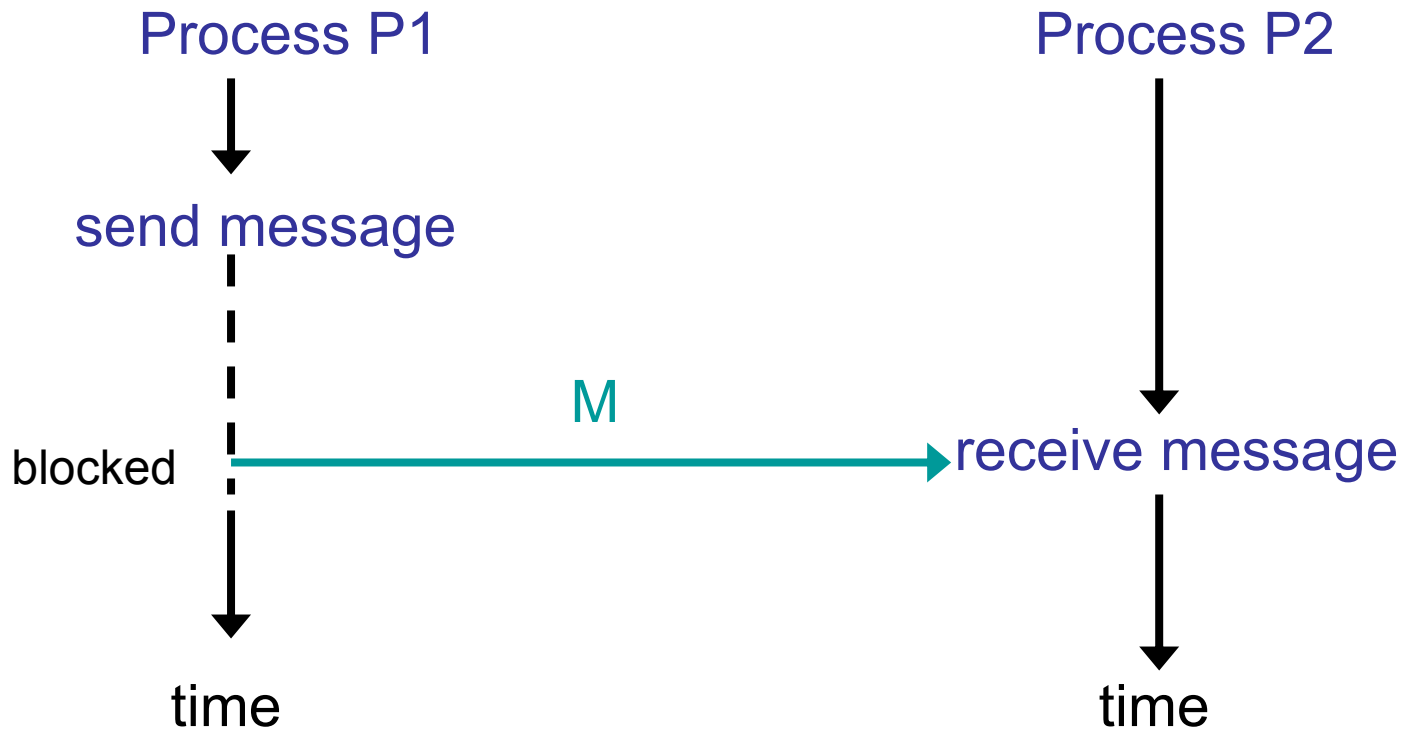
- **Asynchronous** (or **no-wait**) (e.g. POSIX, several OS) the sender proceeds **immediately**, regardless of whether the message is received or not
 - Requires buffer space. What happens when the buffer is full?
- Analogy: Posting of a letter





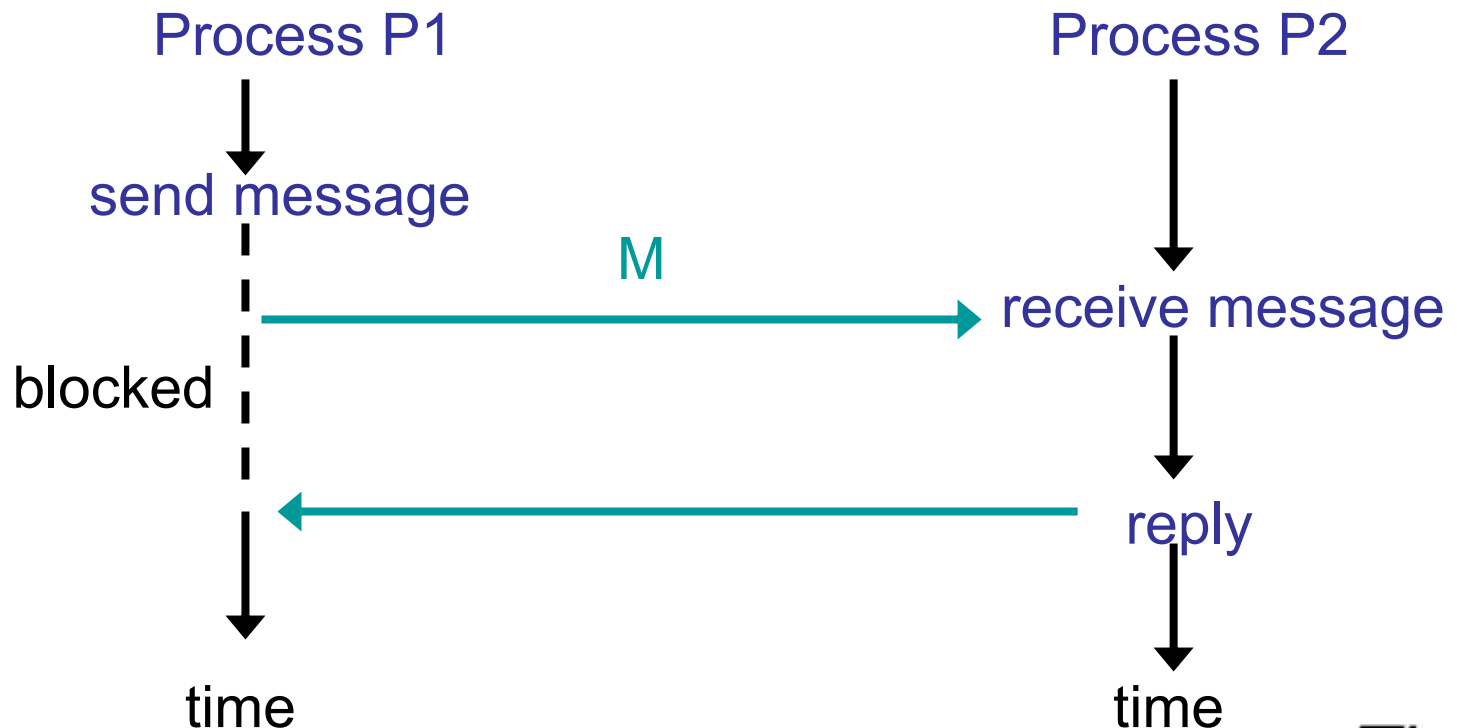
Synchronous semantics

- **Synchronous** (rendezvous) (e.g. CSP (Hoare, 1985), occam2) the sender proceeds **only** when the **message has been received**
- Analogy: telephone call





- **Remote invocation** (also extended rendezvous) (e.g. Ada) the sender proceeds **only** when a **reply has been returned** from the **receiver**. Models the request-response paradigm of communication
- Analogy: telephone call where the receiver can reply immediately, that is during the same call





- Asynchronous communication can implement synchronous communication:

P1

asyn_send (M)

wait (ack)

P2

wait (M)

asyn_send (ack)



- Two synchronous communications can be used to construct a remote invocation:

P1

syn_send (message)

wait (reply)

P2

wait (message)

...

construct reply

...

syn_send (reply)



Asynchronous send having the greatest flexibility?

- Potentially **infinite buffers** are needed to store **unread messages** (perhaps because the receiver has terminated)
- Asynchronous communication is **out-of-date**; most sends are programmed to expect an acknowledgement
- **More communications** are needed with the asynchronous model, hence programs are more complex
- It is more difficult to prove the correctness of the complete system
- Where asynchronous communication is desired with synchronised message passing then **buffer processes can easily be constructed**; however, this is **not without cost**



- Two distinct sub-issues
 - direction versus indirection
 - symmetry
- With direct naming, the sender explicitly names the receiver:
send <message> to <process-name>
- With indirect naming, the sender names an intermediate entity (e.g. a channel, mailbox, link or pipe):
send <message> to <mailbox>
- With a mailbox, message passing can still be synchronous
- Direct naming has the advantage of simplicity, while indirect naming aids the decomposition of the software;
a mailbox can be seen as an interface between parts of the program



- A naming scheme is **symmetric** if both sender and receiver name each other (directly or indirectly)
 - send <message> to <process-name>
 - wait <message> from <process-name>
 - send <message> to <mailbox>
 - wait <message> from <mailbox>
- It is **asymmetric** if the receiver names no specific source but accepts messages from any process (or mailbox)
 - wait <message>
- Asymmetric naming fits the **client-server** paradigm



If naming is indirect the intermediary could have

- **a many-to-one structure**
any number of processes could write to it but only one process can read from it (fits the client-server paradigm)
- **a many-to-many structure**
many clients and many servers
- **one-to-one structure**
one client and one server (no queue needed by RTSS)
- **a one-to-many**
useful when a process wishes to send a request to a group of worker processes and it does not care which process services the request



- A language usually allows any **data object** of any **defined type** (predefined or user) to be transmitted in a message
- Need to **convert to a standard format** for transmission across a network in a heterogeneous environment
- OS allow only **arrays of bytes** to be sent



- Both Ada and occam2 allow communication and synchronisation to be based on message passing
- With occam2 this is the **only** method available
- occam2 uses **indirect symmetric synchronous message passing**
- Ada uses **direct asymmetric remote invocation message passing**



- occam2 processes are **not named**; therefore it is necessary during communication to use **indirect naming** via a **channel**
- Each channel can only be used by a single writer and a single reader process.
- `ch ! X` write value of expression X onto channel ch
- `ch ? Y` read from channel ch into variable Y
- Communication is **synchronous**; therefore whichever process accesses the channel first will be suspended
- When the other process arrives, data will pass from X to Y



```
CHAN OF INT ch:
PAR
  INT V:
  SEQ i = 0 FOR 1000          -- process 1
    SEQ
    -- generate value V
    ch ! V
  INT C:
  SEQ i = 0 FOR 1000          -- process 2
    SEQ
    ch ? C
    -- use C
```



- Ada supports a form of message-passing between tasks
- Based on a client/server model of interaction
- The server declares a set of services that it is prepared to offer other tasks (its clients)
- It does this by declaring one or more public entries in its task specification
- Each entry identifies the name of the service, the parameters that are required with the request, and the results that will be returned
- In order for a task to receive a message, it must define an entry
- To call a task (that is, send it a message) simply involves naming the receiver task and its entry (naming is direct)



```
entry_declaration ::=  
  entry defining_identifier[(discrete_subtype_definition)]  
  parameter_profile;
```

```
entry Syn;  
entry Send(V : Value_Type);  
entry Get(V : out Value_Type);  
entry Update(V : in out Value_Type);  
entry Mixed(A : Integer; B : out Float);  
entry Family(Boolean)(V : Value_Type);
```



```
task type Telephone_Operator is
  entry Directory_Enquiry(
    Person : in Name;
    Addr   : Address;
    Num    : out Number);
  -- other services possible
end Telephone_Operator;

An_Op : Telephone_Operator;

-- client task executes
An_Op.Directory_Enquiry ("Stuart_Jones",
                        "11 Main, Street, York"
                        Stuarts_Number);
```



- To receive a message involves accepting a call to the appropriate entry

```
accept_statement ::=  
  accept entry_direct_name[(entry_index)]  
    parameter_profile [do  
      handled_sequence_of_statements  
    end [entry_identifier]];
```

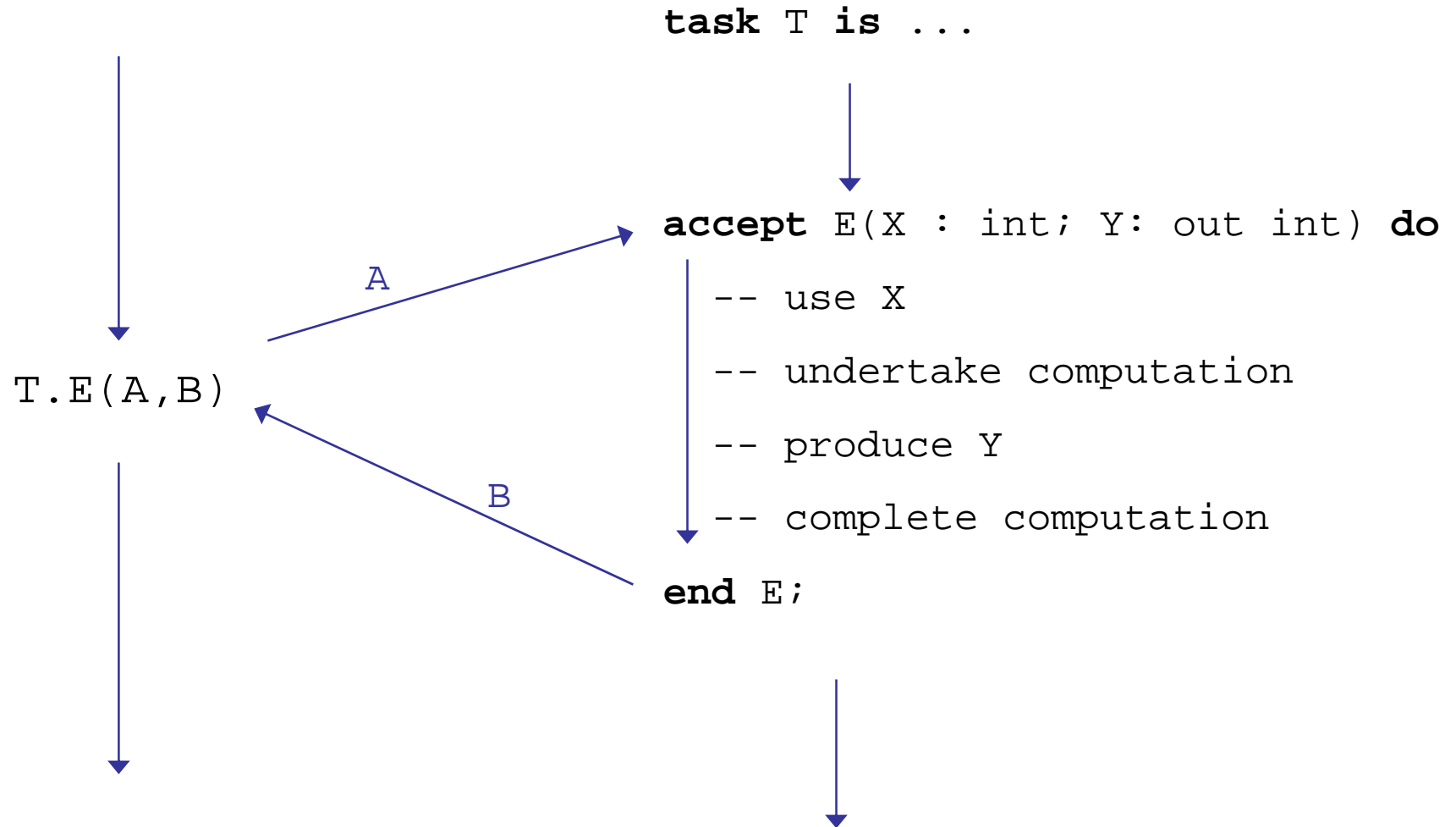
```
accept Family(True)(V : Value_Type) do  
  -- sequence of statements  
exception  
  -- handlers  
end Family;
```



```
task body Telephone_Operator is  
begin  
  ...  
  loop  
    --prepare to accept next call  
    accept Directory_Enquiry (...) do  
      -- look up telephone number  
    exception  
      when Illegal_Number =>  
        -- propagate error to client  
      end Directory_Enquiry;  
      -- undertake housekeeping  
    end loop;  
  ...  
end Telephone_Operator;
```



```
task type Subscriber;  
task body Subscriber is  
begin  
  ...  
  loop  
    ...  
    An_Op.Directory_Enquiry(...);  
    ...  
  end loop;  
  ...  
end Subscriber;
```





- Both tasks must be prepared to enter into the communication
- If one is ready and the other is not, then the ready one waits for the other
- Once both are ready, the client's parameters are passed to the server
- The server then executes the code inside the accept statement
- At the end of the accept, the results are returned to the client
- Both tasks are then free to continue independently



Two tasks looping round and passing data 28

```
procedure Test is
  Number_of_Exchanges : constant Integer := 1000;
  task T1 is
    entry Exchange (I : Integer; J: out Integer);
  end T1;
  task body T1 is
    A, B: Integer;
  begin
    for K in 1 .. Number_of_Exchanges loop
      -- produce A
      accept Exchange (I: Integer; J: out Integer) do
        J := A;
        B := I;
      end Exchange;
      -- Consume B
    end loop
  end T1;
```



Two tasks looping round and passing data 29

```
task body T2 is
  C, D: Integer;
begin
  for K in 1 .. Number_of_Exchanges loop
    -- produce C
    T1.Exchange (C, D)
    -- Consume D
  end loop
end T2

begin
  null;
end Test;
```



- So far, the **receiver** of a message **must wait** until the **specified** process, or mailbox, **delivers** the communication
- A receiver process **may actually wish to wait for any one of a number of processes** to call it
- Server processes **receive request messages** from a number of clients; the order in which the clients call being unknown to the servers
- To facilitate this common program structure, **receiver processes are allowed to wait selectively for a number of possible messages**
- Based on Dijkstra's **guarded commands**



- A **guarded command** is one which is executed **only** if its **guard** evaluates to **True**

- If x is less than y , then assign the value of x to m .

$$x < y \rightarrow m := x$$

- A guarded command is part of a **guarded command set**
 - means **choice**

$$\text{if } x \leq y \rightarrow m := x$$

$$\square \quad x \geq y \rightarrow m := y$$

fi

- m will be either assigned to x or y
- If **both alternatives** are **true** ($x=y$), then an **arbitrary choice** is made; \rightarrow **non-deterministic**
- If the guarded command is a **message operator**, then the statement is known as **selective wait**



The select statement comes in four forms:

```
select_statement ::=
    selective_accept |
    conditional_entry_call |
    timed_entry_call |
    asynchronous_select
```



The selective accept allows the server to:

- wait for more than a single rendezvous at any one time
- time-out if no rendezvous is forthcoming within a specified time
- withdraw its offer to communicate if no rendezvous is available immediately
- terminate if no clients can possibly call its entries



```
selective_accept ::=
  select
    [guard]
    selective_accept_alternative
  { or
    [guard]
    selective_accept_alternative
  [ else
    sequence_of_statements ]
  end select;
```

```
guard ::= when <condition> =>
```



```
selective_accept_alternative ::=  
    accept_alternative |  
    delay_alternative |  
    terminate_alternative
```

```
accept_alternative ::=  
    accept_statement [ sequence_of_statements ]
```

```
delay_alternative ::=  
    delay_statement [ sequence_of_statements ]
```

```
terminate_alternative ::=  
    terminate;
```



Overview Example

```
task Server is  
  entry S1(...);  
  entry S2(...);  
end Server;
```

```
task body Server is
```

```
  ...
```

```
begin
```

```
  loop
```

```
    select
```

```
      accept S1(...) do
```

```
        -- code for this service
```

```
      end S1;
```

```
    or
```

```
      accept S2(...) do
```

```
        -- code for this service
```

```
      end S2;
```

```
    end select;
```

```
  end loop;
```

```
end Server;
```

Simple select with
two possible actions



```
task type Telephone_Operator is  
    entry Directory_Enquiry (P : Name; A : Address;  
                            N : out Number);  
    entry Directory_Enquiry (P : Name; PC : Postal_Code;  
                            N : out Number);  
    entry Report_Fault(N : Number);  
private  
    entry Allocate_Repair_Worker (N : out Number);  
end Telephone_Operator;
```



```
task body Telephone_Operator is  
  Failed : Number;  
task type Repair_Worker;  
Work_Force : array(1.. Num_Workers) of  
  Repair_Worker;  
  
task body Repair_Worker is separate;
```



```
begin
  loop
    select
      accept Directory_Enquiry( ... ; A: Address...) do
        -- look up number based on address
      end Directory_Enquiry;
    or
      accept Directory_Enquiry( ... ;
                                PC: Postal_Code...) do
        -- look up number based on ZIP
      end Directory_Enquiry;
    or
```



or

```
accept Report_Fault(N : Number) do
    ...
end Report_Fault;
if New_Fault(Failed) then
    accept Allocate_Repair_Worker (N : out
        Number) do
        N := Failed;
    end Allocate_Repair_Worker;
end if;
end select;
end loop;
end Telephone_Operator;
```



- If no rendezvous are available, the select statement **waits for one** to become available
- If one is available, it is chosen **immediately**
- If more than one is available, the **one chosen is implementation dependent**
- **More than one task can be queued on the same entry;** default queuing policy is FIFO



- Each select accept alternative can have an associated **guard**
- The **guard** is a **boolean expression** which is **evaluated** when the **select statement** is executed
- If the guard **evaluates to true**, the alternative is **eligible for selection**
- If it is false, the alternative is **not eligible for selection** during this execution of the select statement (even if client tasks are waiting on the associated entry)



Example of Guard

```
task body Telephone_Operator is
begin
  ...
  select
    accept Directory_Enquiry (...) do ... end;
  or
    accept Directory_Enquiry (...) do ... end;
  or
    when Workers_Available => guard
    accept Report_Fault (...) do ... end;
  end select;
end Telephone_Operator;
```



- The semantics of message-based communication are defined by three issues:
 - the **model of synchronisation**
 - the **method of process naming**
 - the **message structure**
- **Variations** in the process synchronisation model arise from the semantics of the **send** operation.
 - **asynchronous, synchronous or remote invocation**
 - Remote invocation can be made to **appear syntactically similar** to a procedure call
- Process naming involves two distinct issues; **direct or indirect**, and **symmetry**



- Ada has **remote invocation** with **direct asymmetric naming**
- Communication in Ada requires one task to **define an entry** and then, **within its body**, **accept any incoming call**.
A rendezvous occurs when one task calls an entry in another
- **Selective waiting** allows a **process** to wait for more than one message to arrive.
- Each **select accept alternative** can have an associated **guard**; if the guard **evaluates to true**, the alternative is eligible for selection
- Ada's **select statement** has two extra facilities: an **else part** and a **terminate alternative**