

# Unit 4

## Message Passing: Synchronous Communication

**4.1 Introduction**

**4.2 Message passing models**

**4.3 Synchronous communication**

**4.3.1 Intro to synch. comm.**

**4.3.2 Selective waiting**

**4.3.3 Guarded selective waiting**

**4.3.4 Selective waiting – terminate**

**4.3.5 Selective waiting – else,timeout,pri**

## **4.1 Introduction**

### **4.2 Message passing models**

### **4.3 Synchronous communication**

#### **4.3.1 Intro to synch. comm.**

#### **4.3.2 Selective waiting**

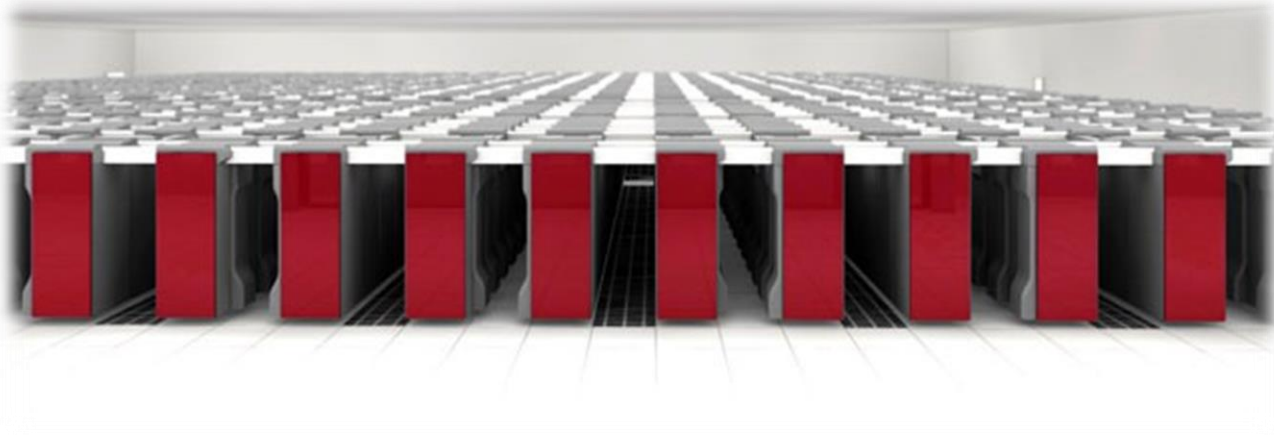
#### **4.3.3 Guarded selective waiting**

#### **4.3.4 Selective waiting – terminate**

#### **4.3.5 Selective waiting – else,timeout,pri**

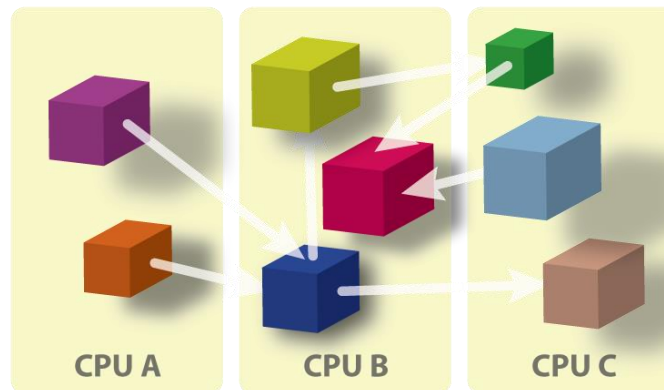
# 4.1 Introduction

- Semaphores, CCRs and Monitors are concurrent programming tools based on **shared memory**.
- If our program is to be run in a **distributed system**, in which physical memory is not shared, then these tools are not useful anymore.



# 4.1 Introduction

- In a distributed system:
  - Several processors are connected through a network.
  - Processors do not share memory nor clock.
  - Connected systems and hardware may be different among them
  - The network can be scaled up to no limit (Internet).



- A concurrent program which uses **message-passing** can be executed in one single **platform** (messages passed using shared memory). However, the contrary is not true.

## **4.1 Introduction**

## **4.2 Message passing models**

## **4.3 Synchronous communication**

### **4.3.1 Intro to synch. comm.**

### **4.3.2 Selective waiting**

### **4.3.3 Guarded selective waiting**

### **4.3.4 Selective waiting – terminate**

### **4.3.5 Selective waiting – else, timeout, pri**

## 4.2 Message passing models

- Since memory is not shared, the alternative used for concurrent (parallel) programs is to pass messages among nodes executing processes.
- In message passing, the **mutual exclusion** problem does not exist. However, we still need to solve synchronization problems.
- The **basic operations** needed in message passing are:
  - SEND: the process sends a message
  - RECEIVE: the process receives a message
- The specific implementation of the SEND and RECEIVE operations leads to different message passing models.

## 4.2 Message passing models

- Whatever the model, the generic **communication scheme** is:



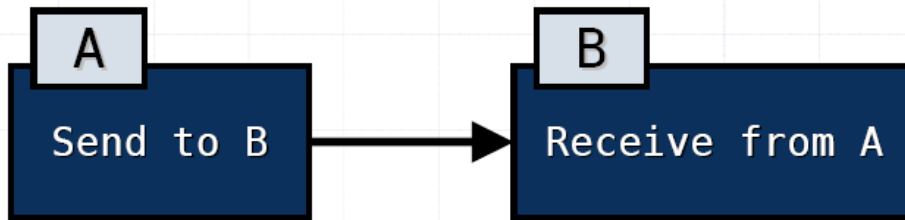
- A taxonomy of the communication models can be depicted attending to 3 different aspects:
  - Addressing method
  - Synchronization method
  - Channel characteristics

## 4.2 Message passing models

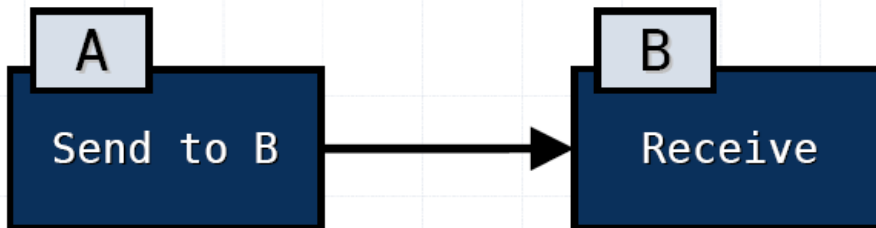
Depending on the **addressing methods**, message passing can be:

- **Direct** communication: explicitly name the process you are communicating with:
  - SEND(A, message): send message to process A
  - RECEIVE(B, message): receive a message from B
  - It is fast, but any **change in the identification** of processes makes it necessary to recompile and launch the program.
  - In a client/server application, the clients knows the receiver but the server cannot know the IDs of the clients. This can be solved with **asymmetric** direct communication:
    - SEND(A, message)
    - RECEIVE(ID, message): the OS tells us the ID of the sender.
- **Indirect** communication: processes are not identified, and messages are sent to/received from mailboxes (ports) or channels.
  - SEND(mailboxA, message)
  - RECEIVE(mailboxA, message)
  - A **channel** is a communication link used by only one sender or receiver at a time. Pascal-FC uses channels.

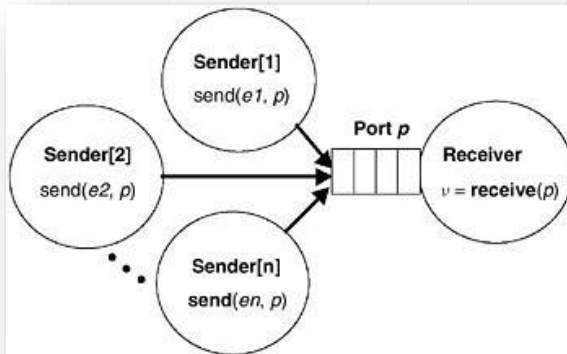
## 4.2 Message passing models



Direct symmetric communication



Direct asymmetric communication



Indirect communication

## 4.2 Message passing models

Depending on the **synchronization method**:

- **Synchronous** communication: the sending/receiving process is delayed until the corresponding *RECEIVE/SEND* is executed: **rendezvous** e.g. phone call
  - messages do not need to be buffered
  - both the send and receive **operations** are **blocking**: the process which tries to communicate **first will be blocked**.
  - If an answer is sent back, this is **extended rendezvous or remote invocation**
- **Asynchronous** communication: the sender sends a message and continues executing without waiting for the message to be received: *fax, print server*
  - **SEND** operation is **non-blocking**.
  - message delivery is not guaranteed (channel failures can occur).
  - messages have to be buffered: Problem?

## 4.2 Message passing models

Depending on the **channel characteristics**:

- **Data flow**: the sense in which data is sent. Unidirectional (e-mail) or bidirectional (chat).
- **Capacity**: the amount of data the channel can store before the messages are retrieved from the receivers.
- **Size of message**: if it is fixed size, the programmer needs to deal with the slicing of oversized messages. If size is not fixed, the designer of the communication system will need to use dynamic memory allocation.
- **Data type**: is a given type mandatory?

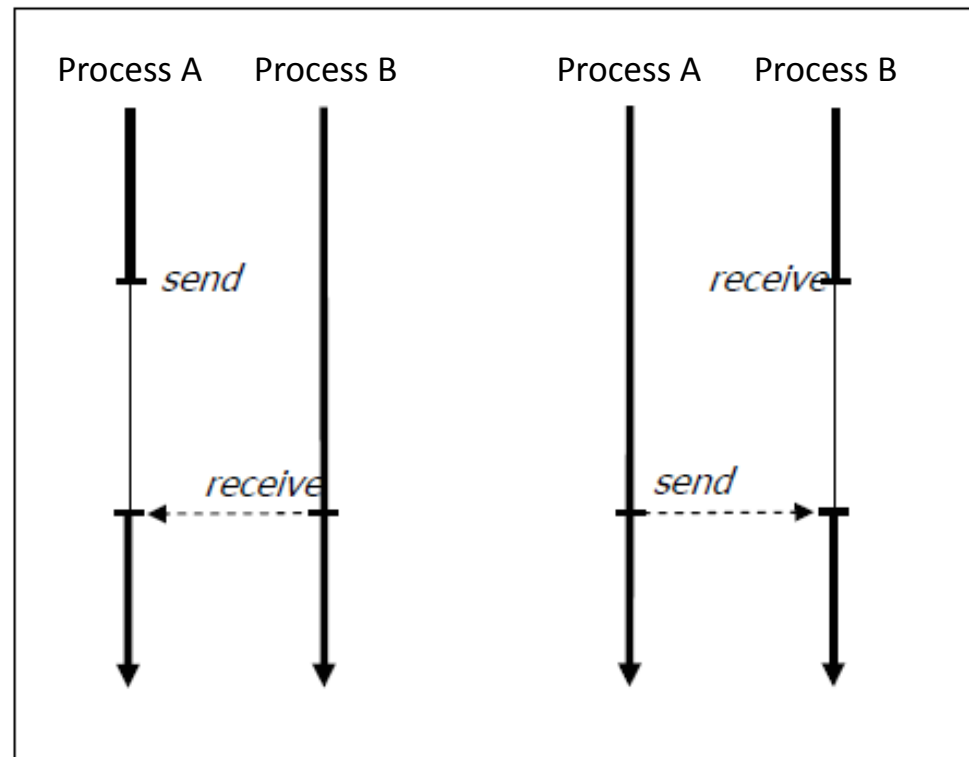
Models provided by some **programming languages**:

- **ADA**: rendezvous or synchronous
- **Erlang**: asynchronous
- **JAVA**: sockets (asynchronous-like) and RMI (synchronous) libraries.
- **Pascal-FC**: synchronous.

- 4.1 Introduction**
- 4.2 Message passing models**
- 4.3 Synchronous communication**
  - 4.3.1 Intro to synch. comm.**
  - 4.3.2 Selective waiting
  - 4.3.3 Guarded selective waiting
  - 4.3.4 Selective waiting – terminate
  - 4.3.5 Selective waiting – else,timeout,pri

## 4.3.1 Intro to Synch. Comm.

- Pascal-FC provides only synchronous communication tools. Java does not provide primitives for synchronous nor asynchronous communication. Thus, we will focus on solving concurrent programming problems using synchronous message passing with Pascal-FC.



## 4.3.1 Intro to Synch. Comm.

- **One channel allows 2 processes** to communicate to each other through a link.
- This link:
  - is established between 1 sender and 1 receiver.
  - Is unidirectional: only the receiver or sender can use it at the same time.
- The channel is typed: only data of such type can be sent through the link
- Pascal-FC provides the following operators:

`ch ! e`    *sends **e** through channel **ch***

`ch ? v`    *receives **v** from channel **ch***

## 4.3.1 Intro to Synch. Comm.

- A channel is declared with keyword *channel*.
- If you want to define one channel for integer data, and a channel for a given structure:

```
var link : channel of integer;  
  
type package=  
  record  
    (* some structure*)  
  end;  
var network : channel of package;
```

## 4.3.1 Intro to Synch. Comm.

```
program basicExample;
var
ch: channel of integer;

process S;
var x:integer;
begin
repeat
ch ! x;
x:=x+1;
until x=10;
end;

process R;
var y:integer;
begin
repeat
ch ? y;
writeln('Message ', y, ' received');
until y=9;
end;
```

```
begin
cobegin
S;
R;
coend
end.
```

*Why is the stop condition y=9 in process R?*

*Can we be sure that the values printed by R will be ordered?*

## 4.3.1 Intro to Synch. Comm.

- The previous example shows that process R is synchronized with S: it does not write until it receives the message.
- We may want R to be the producer and writer of the integer value, maintaining the synchronization.
- We can define a channel used just for synchronization, and send a signal through the channel.

```
var ch: channel of synchronous;  
ch ! any  
ch ? any
```

- “any” is a variable of type synchronous. It is defined by default in Pascal-FC

## 4.3.1 Intro to Synch. Comm.

```
program basicSyncExample;  
  
var ch: channel of synchronous;  
  
process S;  
begin  
  repeat  
    ch ! any;  
  forever  
end;  
  
process R;  
var x:integer;  
begin  
  repeat  
    ch ? any;  
    writeln('Action ', x, ' synchronized');  
    x:=x+1;  
  forever  
end;
```

```
begin  
  cobegin  
    S;  
    R;  
  coend  
end.
```

- 4.1 Introduction**
- 4.2 Message passing models**
- 4.3 Synchronous communication**
  - 4.3.1 Intro to synch. comm.**
  - 4.3.2 Selective waiting**
  - 4.3.3 Guarded selective waiting
  - 4.3.4 Selective waiting – terminate
  - 4.3.5 Selective waiting – else,timeout,pri

## 4.3.2 Selective Waiting

- Several process may want to send messages (each one trough a different channel) to the same receiver.
- The receiver needs to listen to the channels in a non-sequential manner to **avoid to get blocked on one channel** without messages, while other channels do have messages waiting to be read: *selective waiting*.
- Pascal-FC provides the primitive “**select**”, which **randomly** selects the message from channels **which have 1 message** waiting to be read. We can specify all channels, or use an array.

```
select
  ch1 ? message1;
or
  ch2 ? message 2;
or
  ...
or
  chN ? message N;
end
```

Simplified manner  
with **replicate**

```
select
  for cont:=1 to N replicate
  begin
    ch[cont] ? message[cont];
  end;
or
  another ? anotherMessage;
end
```

- Keyword “**another**” listens to all channels not specified in the array, and writes the message in “anotherMessage”.

## 4.3.2 Selective Waiting

Selective waiting helps us solve some concurrent programming problems such as the Ornamental Gardens: visitors may enter from 2 turnstiles, and a counter of visitors needs to be updated.

```
program OrnamentalGardens;
type syncChannel = channel of synchronous;
var paths : array[1..2] of syncChannel;
(*or var paths : array[1..2] of channel of synchronous;*)

process type turnstile(id, people: integer);
var i:integer;
begin
    for i:=1 to people do
        paths[id] ! any;
    end;

process counter;
var count,i : integer;
begin
    count:=0;
    for i:=1 to 40 do
        begin
            select
                paths[1] ? any;
            or
                paths[2] ? any;
            end;
            count := count+1;
        end;
    writeln('People who visited the Gardens:',count);
end;
```

```
var turn1,turn2:turnstile;
begin
cobegin
    counter;
    turn1(1,20);
    turn2(2,20);
coend
end.
```

If both alternatives in select contain a message, only 1 channel is randomly chosen.

If no alternative contains a message, the process is suspended.

## 4.3.2 Selective Waiting

- The previous example is a read alternative. In general, each alternative within a **select** statement can be one of **four types**:
  1. Message read alternative
  2. Message write alternative
  3. *timeout* alternative
  4. *terminate* alternative
- In addition there may be a default **else** alternative, and different priorities.
- If, when a **select** is executed, there are no ready alternatives then the process is suspended until one becomes ready (unless there is an **else** alternative).

- 4.1 Introduction**
- 4.2 Message passing models**
- 4.3 Synchronous communication**
  - 4.3.1 Intro to synch. comm.**
  - 4.3.2 Selective waiting**
  - 4.3.3 Guarded selective waiting**
  - 4.3.4 Selective waiting – terminate
  - 4.3.5 Selective waiting – else,timeout,pri

## 4.3.3 Guarded selective waiting

- We may want all alternatives not to be selectable in each **select** execution.
- **Guarded alternatives**: an alternative which is ignored if its condition is not fulfilled. When the condition is TRUE, then the alternative is **open**.

```
select
  when condition1 =>
    ch1 ? message1;
  or
    ch2 ! message 2;
  or
    ...
  or
    when conditionN =>
      chN ? message N;
end
```

- Only **open ready** alternatives and **non-guarded ready** alternatives are candidates to be randomly chosen for execution.

## 4.3.3 Guarded selective waiting

### Producer/Consumer problem

- Guards are useful in buffer problems, such as the Producer/Consumer problem: we need to control the *insert* and *remove* operations when the buffer is full and empty, respectively.
- Remember that **synchronized communication needs 1 channel for each pair sender-receiver**: we need 1 channel for each producer and also for each consumer.
- In order to solve the problem, we need to define:
  - Array of insertion channels.
  - Array of removal channels
  - Process buffer, which controls insert and remove operations
  - Insert index, remove index
  - numItems

```

program producerConsumer;
const
    SIZE=8;
    PRODUCERS=3;
    CONSUMERS=3;
    N=10; (*items to produce per producer*)
var
    insertChannel: array[1..PRODUCERS] of channel of integer;
    removeChannel: array[1..CONSUMERS] of channel of integer;
    itemValue:integer;

process bufferController;
var
    data: array[0..SIZE] of integer;
    insertIndex, removeIndex, numItems, producerID, consumerID: integer;
begin
    insertIndex:=1;
    removeIndex:=1;
    numItems:=0;
    repeat
        select
            for producerID:=1 to PRODUCERS replicate
            when numItems < SIZE =>
                insertChannel[producerID] ? data[insertIndex];
                insertIndex := insertIndex MOD SIZE + 1;
                numItems:=numItems+1;

        or
            for consumerID:=1 to CONSUMERS replicate
            when numItems > 0 =>
                removeChannel[consumerID] ! data[removeIndex];
                removeIndex := removeIndex MOD SIZE + 1;
                numItems := numItems - 1;

        or
            terminate;
        end
    forever
end;

```

### 4.3.3 Guarded selective waiting

## 4.3.3 Guarded selective waiting

```
process type tProducer(ID:integer);
var i:integer;
begin
  for i:=1 to N do
  begin
    insertChannel[ID] ! i;
    writeln('item ',i, '
      inserted by producer [' ,i, '].');

  end;
end;

process type tConsumer(ID:integer);
var i:integer;
begin
  for i:=1 to N do
  begin
    removeChannel[ID] ? i;
    writeln('item ',i, '
      removed by consumer [' ,i, '].');
    end;
  end;
```

```
var
  myProducers: array[1..PRODUCERS] of tProducer;
  myConsumers: array[1..CONSUMERS] of tConsumer;
  x,y:integer;

begin
  cobegin
    for x:=1 to PRODUCERS do
      myProducers[x](x);
    for y:=1 to CONSUMERS do
      myConsumers[y](y);
    bufferController;
  coend
end.
```

Note each consumer is allowed to remove only N elements. Thus, we are sure all consumer processes end.

- 4.1 Introduction**
- 4.2 Message passing models**
- 4.3 Synchronous communication**
  - 4.3.1 Intro to synch. comm.**
  - 4.3.2 Selective waiting**
  - 4.3.3 Guarded selective waiting**
  - 4.3.4 Selective waiting – terminate**
  - 4.3.5 Selective waiting – else,timeout,pri**

## 4.3.4 Selective waiting - terminate

- The **terminate** alternative is necessary because passive processes such as the `bufferController` needs to know when to finish execution. Otherwise it may be blocked forever in the **select** sentence.
- We can control this with a counter, but it is much more efficient to make it finish when no more active processes are working.
- Thus, a process only enters in the **terminate** and finishes if and only if:
  - No more alternatives are ready
  - And the other processes are finished or also blocked in a **select** sentence with **terminate** alternative.

## 4.3.4 Selective waiting - terminate

- Thus, in the problem of the Ornamental Gardens, we do not need to specify the number of iterations for the counter process:

```
process counter;  
var count,i : integer;  
begin  
  count:=0;  
  for i:=1 to 40 do  
    begin  
      select  
        paths[1] ? any;  
      or  
        paths[2] ? any;  
      end;  
      count := count+1;  
    end;  
    writeln('People who visited the  
    Gardens:',count);  
  end;
```



```
process counter;  
var count: integer;  
begin  
  count:=0;  
  repeat  
    select  
      paths[1] ? any;  
    or  
      paths[2] ? any;  
    or  
      terminate  
    end;  
    count := count+1;  
  forever;  
  writeln('People who visited the  
  Gardens:',count);  
end;
```

The process finishes correctly, but... can you see a problem?

## 4.3.4 Selective waiting - terminate

```
process counter(var count:integer);
begin
  repeat
    select
      paths[1] ? any;
    or
      paths[2] ? any;
    or
      terminate
    end;
    count := count+1;
  forever;
end;

var
  turn1,turn2:turnstile;
  number:integer;
begin
  number:=0;
  cobegin
    counter(number);
    turn1(1,20);
    turn2(2,20);
  coend;
  writeln('People who visited the Gardens:',number);
end.
```

Counter process for the Ornamental Gardens problem, using the terminate alternative.

- 4.1 Introduction**
- 4.2 Message passing models**
- 4.3 Synchronous communication**
  - 4.3.1 Intro to synch. comm.**
  - 4.3.2 Selective waiting**
  - 4.3.3 Guarded selective waiting**
  - 4.3.4 Selective waiting – terminate**
  - 4.3.5 Selective waiting – else,timeout,pri**

## 4.3.5 Selective waiting – else, timeout, pri

- Sometimes we may want the sender or receiver not to be blocked until its message has been received or sent.
- Alternative **else** :
  - tells the process to do another thing if no alternative is ready.
  - Only 1 per **select**.
  - It cannot be guarded.
- Alternative **timeout**
  - tells the process the time it may remain blocked waiting for an alternative to be ready, and then do another thing.
  - It may be guarded.
  - Useful in real-time systems: trigger alarm if there is no communication with the plane.
- Alternatives **terminate**, **else** and **timeout** cannot be used in the same **select**.

## 4.3.5 Selective waiting – else, timeout, pri

```
select
  ch1 ? message1;
or
  ch2 ? message 2;
or
  ...
else (*do something*)
end
```

```
select
  ch1 ? message1;
or
  ch2 ? message 2;
or
  ..
or
  timeout n;
  (*do something*)
end
```

- Imagine a producer process generates numbers in a forever loop.

```
type chInt = channel of integer;

process producer (var ch1: chInt);
var i:integer;
begin
  i:=1;
  repeat
    ch1 ! i;
    i:=i+1;
  forever
end;
begin
end.
```

*How to tell the producer to stop sending numbers?*

## 4.3.5 Selective waiting – else, timeout, pri

```
program stopProducerWithElse;

type chInt = channel of integer;
type chSync = channel of synchronous;

process producer (var ch1: chInt;
  var chSyn: chSync);
var i: integer;
stop: boolean;
begin
  i:=1;
  stop:=false;
  while not stop do
    select
      chSyn ? any;
      stop:=true;
    else
      ch1 ! i;
      i:=i+1;
    end;
  writeln('producer finished.');
```

```
process consumer(var ch1: chInt;
  var chSyn: chSync);
var i,n: integer;
begin
  for i:=1 to 10 do
    begin
      ch1 ? n;
      writeln(n);
    end;
    chSyn ! any;
    writeln('consumer finished.');
```

```
end;

var
  ch1: chInt;
  chSyn: chSync;
begin
  cobegin
    producer(ch1, chSyn);
    consumer(ch1, chSyn);
  coend
end.
```

## 4.3.5 Selective waiting – else, timeout, pri

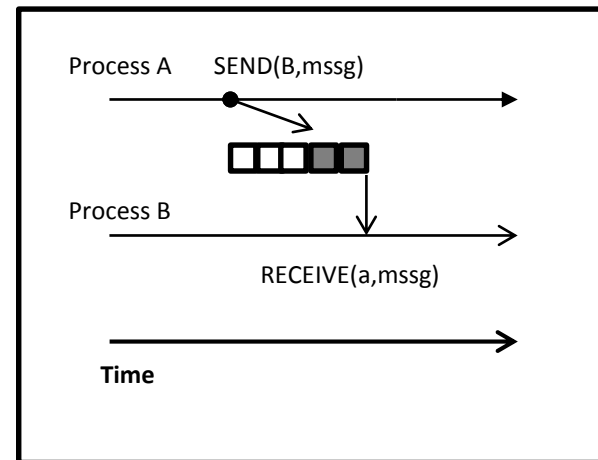
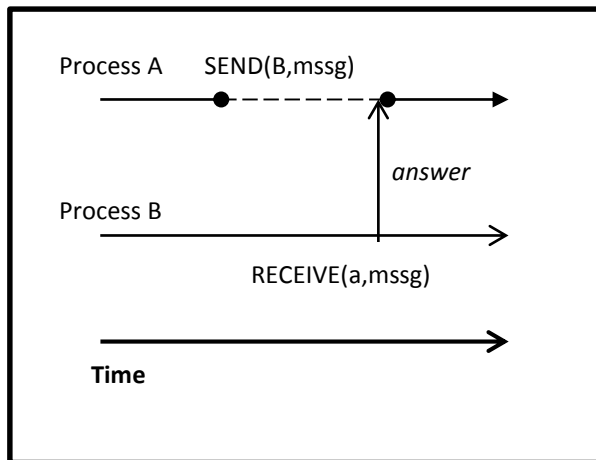
- If we use the modifier **pri** in a select sentence, the alternative to execute is not chosen randomly but by order of appearance.

```
pri select
  ch1 ? message1;
or
  ch2 ? message 2;
or
  ...
or
  chN ? message N;
end
```

**Synchronous message passing simulated in Java is one of the choices for Task 2.  
See Section 9.4.1 in the book of Palma et.al**

# Potential Mid-term Exam Questions

1. What is the difference between the **else** and **terminate** alternatives?
2. When can a process be blocked inside a **select** which uses an **else** alternative?
3. Which kind of communication do you think these schemes represent? (from the point of view of synchronization)



# Keywords phonetics

- distribute /dɪ'strɪbjʊ:t/ 
- direct /daɪ'rekt/ 
- asymmetric /eɪsɪ'metrɪk/ 
- asynchronous /eɪ'sɪŋkrənəs/ 
- priority /prɪ'tɪəri/ 
- guarded /'gɑ:dɪd/ 
- message /'mesɪdʒ/ 
- receive /rɪ'si:v/ 
- indirect /,ɪndɪ'rekt/ 