Mohamed Boudiaf
University
-M'sila-

جامعة محمد بوضياف - المسيلة
Université Mohamed Boudiaf - M'sila

Faculty of Mathematics
and Computer Science

Computer Science
Department

OPERATING SYSTEMS II

CHAPTER III

INTER-PROCESS COMMUNICATION (IPC)

3rd Year, Computer Science – SI

Dr. Hanene CHERAIT

2025-2026

INTERPROCESS COMMUNICATION

- Processes need to communicate with one another by exchanging information.
- Example: A web browser may request a page from a server, which then sends back the HTML data.
- There is a need for communication between a set of processes in a well-structured way that avoids potential interruptions.

INTERPROCESS COMMUNICATION

- We distinguish two complementary types of communication:

- **Indirect communication**:
  - Shared variables
  - Mailboxes

- **Direct communication**:
  - Message exchange

INDIRECT COMMUNICATION - SHARED VARIABLES

- Processes communicate by sharing common global variables.
- Managing the communication is the responsibility of the application programmer.
- This type of communication is based on the explicit use of synchronization mechanisms (semaphores, monitors, locks, etc.).
- **Examples:**
  - Producer/Consumer model
  - Reader/Writer model

INDIRECT COMMUNICATION - MAILBOXES

- Mailboxes can be considered as a buffer
  - a certain number of messages can be placed into it and removed by a group of processes (same principle as producers/consumers).
- Each mailbox is identified by a unique identifier.
- It can only be manipulated using two primitives:
  - send(B, M): Sends message M into mailbox B
  - receive(B, M): Retrieves message M from mailbox B

INDIRECT COMMUNICATION - MAILBOXES

- Mailboxes can have two different origins:
  - **Operating system:**
    - The system allows the mailbox to be created (or deleted) and manages sending and receiving within the mailbox.
    - The mailbox is not associated with any specific process.
    - The termination of a process has no effect on the existence of the mailbox.
  - **Process:**
    - The owner process is allowed only to receive messages.
    - The user process is allowed only to send messages.
    - Each mailbox is created by a single owner process.
    - Each mailbox disappears when its owner process terminates.

DIRECT COMMUNICATION

MESSAGE EXCHANGE

- Example: the Client/Server model
- Information exchange is done through:
  - **Send(Pid, message)**
    - Sends a message to the process with the identification number *Pid*.
    - The message is copied into the receiver's message queue.
    - Any process can send a message to any other process, provided that the latter exists.
  - **Receive(Pid, message)**
    - Receives a message from the process with the identification number *Pid*.
    - Messages are received in the order they were sent.
    - If there are no messages in the queue, the process remains blocked until a message is sent (otherwise, it returns an error code).

DIRECT COMMUNICATION

MESSAGE EXCHANGE

- Problems related to the message exchange system:

Problem	Solution
Lost message	Use of acknowledgment (ACK)
Message received successfully but acknowledgment lost	Retransmission of the same message
Ambiguity of similar process names in the network	Management of the process naming mechanism so that a process passed as a parameter in send or receive does not cause ambiguity.

DIRECT COMMUNICATION
MESSAGE EXCHANGE

- **Producer/Consumer Problem:**
  - All messages have the same size.
  - The operating system automatically buffers messages that have been sent but not yet received.

PRODUCER-CONSUMER PROBLEM

```
int N = 9999; // buffer size

void producer() {
    in obj;
    message m;
    while (true) {
        // produce the object to send to the consumer
        obj = produceObject();
        // wait for an empty message from the consumer
        receive(consumer, m);
        // construct the message to send
        buildMessage(m, obj);
        // send the object to the consumer
        send(consumer, m);
    }
}
```

PRODUCER-CONSUMER PROBLEM

```
void consumer() {
    in obj, i;
    message m;
    // broadcast N empty messages to the producers
    for (i = 0; i < N; i++) send(producer, m);
    while (true) {
       receive(producer, m); // retrieve a message m that contains the object
       obj = extractObject(m); // extract the object to consume from message m
       send(producer, m); // send an empty reply back to the producer
       consumeObject(obj); // consume the received object
    }
}
```

COMMUNICATION IN HIGH-LEVEL LANGUAGES

- JAVA
- CSP
- Ada

COMMUNICATION IN HIGH-LEVEL LANGUAGES

JAVA

CSP

Ada

COMMUNICATION IN HIGH-LEVEL LANGUAGES - JAVA -

- Communicate via “sockets”

A “socket” refers to the endpoint of a communication channel through which a Java program can send or receive data.

COMMUNICATION IN HIGH-LEVEL LANGUAGES - JAVA -

- Creating a socket for a client requires one of the following constructors:

```
public Socket(String host, int port)
throws UnknownHostException,
IOException
```

host: name of the machine
port: port number

```
public Socket(InetAddress address, int port)
throws IOException
```

address: the machine's IP address
port: port number

COMMUNICATION IN HIGH-LEVEL LANGUAGES - JAVA -

```
public Socket(String hostRemote, int portRemote,
    InetAddress localAddr, int localPort)
    throws IOException
```

- hostRemote: name of the remote machine
- portRemote: remote port number
- LocalAdde: address of the local machine
- localPort: local port number

```
public Socket(InetAddress addressRemote, int portRemote,
    InetAddress localAddr, int localPort)
    throws IOException
```

- addressRemote: address of the remote machine
- portRemote: remote port number
- LocalAdde: address of the local machine
- localPort: local port number

COMMUNICATION IN HIGH-LEVEL LANGUAGES - JAVA -

- Once the connection is established, the following methods allow us to retrieve the input stream and the output stream of the TCP connection to the server.

```
public InputStream getInputStream() throws IOException

public OutputStream getOutputStream() throws IOException
```

- The client can exchange data with the server using the standard reading and writing methods provided by the classes of the java.io package.

COMMUNICATION IN HIGH-LEVEL LANGUAGES - JAVA -

- **ServerSocket Class**
  - is used to create a socket on the server side and to define a communication endpoint characterized by an IP address and a port number.
  - All requests made by remote clients are transmitted to the server through this endpoint.

COMMUNICATION IN HIGH-LEVEL LANGUAGES - JAVA -

- Constructors:

```
/* port: the local port specified as a parameter */
public ServerSocket(int port)
throws IOException
```

```
//bindAddr: local adress
public ServerSocket(int port, int backlog,
InetAddress bindAddr)
throws IOException
```

```
/* backlog: specification of the number of simultaneous connections allowed */
public ServerSocket(int port, int backlog)
throws IOException
```

```
//waiting for incoming client connections
public Socket accept()
throws IOException
```

COMMUNICATION IN HIGH-LEVEL LANGUAGES - JAVA -

- **Example of communication in Java:**
  - A communication between a server and a client that both connect on the same port, 8080.
  - When the client is executed, it first asks us to enter the server's IP address in order to connect.
  - Once it is connected to the server, it displays the date that was sent to it by the server.

```java
import java.io.*;
import java.net.*;
import java.util.Date;

/* Server-side.*/
public class MyServer {
   // main method
   public static void main(String[] args) throws IOException {
      /* Create a ServerSocket object
         on port 8080
      */
      ServerSocket listenerObject = new ServerSocket(8080);
      try {
         while (true) {
            /* Create a Socket object
               using the previously created ServerSocket
            */
            Socket socketObject = listenerObject.accept();
            try {
               // Retrieve data
               PrintWriter out = new
PrintWriter(socketObject.getOutputStream(), true);
               // Send the current date and time to the client
               out.println(new Date().toString());
            } finally {
               // Close the client socket
               socketObject.close();
            }
         }
      } finally {
         // Close the server socket
         listenerObject.close();
      }
   }
}
```

```
import java.io.*;
import java.net.*;
import javax.swing.JOptionPane;

/* Client side */
public class MyClient {
   public static void main(String[] args) throws IOException {
      // Input the server's IP address
      String serverAddress = JOptionPane.showInputDialog ( "Enter the server machine's IP
Address:" );
      Socket client = new Socket(serverAddress, 8080);
      BufferedReader input = new BufferedReader (new InputStreamReader(client.getInputStream())
);
      String response = input.readLine();
      JOptionPane.showMessageDialog(null, response);
      System.exit(0);
   }
}
```

COMMUNICATION IN HIGH-LEVEL LANGUAGES

JAVA

CSP

Ada

COMMUNICATION IN HIGH-LEVEL LANGUAGES - CSP -

- **CSP: Communicating Sequential Processes**
- This concept was first introduced by Tony Hoare in 1978.
- It has been used to model communication protocols, hardware, operating systems, as well as real-time control automata.
- A CSP process is an active entity that encapsulates both data and the algorithms that act on this data.

COMMUNICATION IN HIGH-LEVEL LANGUAGES - CSP -

- Communication between processes is carried out exclusively through message exchanges over one or more unidirectional communication channels, rather than via shared variables.

- To manage communication through channels, CSP processes rely on two primitives associated with each channel:
  - Sending, denoted by !, is performed by invoking the write method.
  - Receiving, denoted by ?, is performed by invoking the read method.

COMMUNICATION IN HIGH-LEVEL LANGUAGES - CSP -

- The Producer/Consumer Problem in CSP

```
import jcsp.lang.*;
/* Main class for launching communication between producer and consumer */
public class PC {
   public static void main(String[] args) {
      // Create and run two processes in parallel
      One2OneChannel C2C = new One2OneChannel();
      new Parallel(new CSProcess[] {
         new Prod(C2C), new Cons(C2C)
      }).run();
      System.out.println("CSP-based communication completed!");
   }
}
```

COMMUNICATION IN HIGH-LEVEL LANGUAGES - CSP -

```
import jcsp.lang.*;
/*  Producer class that creates and sends an object (message) through the channel */
public class Prod implements CSProcess {
    // Channel declaration
    private ChannelOutput C;
    private String msg = " CSP";
    // Constructor for Prod
    public Prod(ChannelOutput canal1) {
        C = canal1;
    }
    public void run() {
        // Write to channel C
        C.write(msg);
        System.out.println("Hello " + msg + " from the producer!");
    }
}
```

COMMUNICATION IN HIGH-LEVEL LANGUAGES - CSP -

```
import jcsp.lang.*;
/* Consumer class that retrieves and consumes an object from the channel */
public class Cons implements CSProcess {
    // Channel declaration
    private ChannelInput C;
    private String msg;
    // Constructor for Cons
    public Cons(ChannelInput canal1) {
        C = canal1;
    }
    public void run() {
        // Retrieve the message from channel C
        msg = (String) C.read();
        System.out.println("Hello " + msg + " from the consumer!");
    }
}
```

COMMUNICATION IN HIGH-LEVEL LANGUAGES

JAVA

CSP

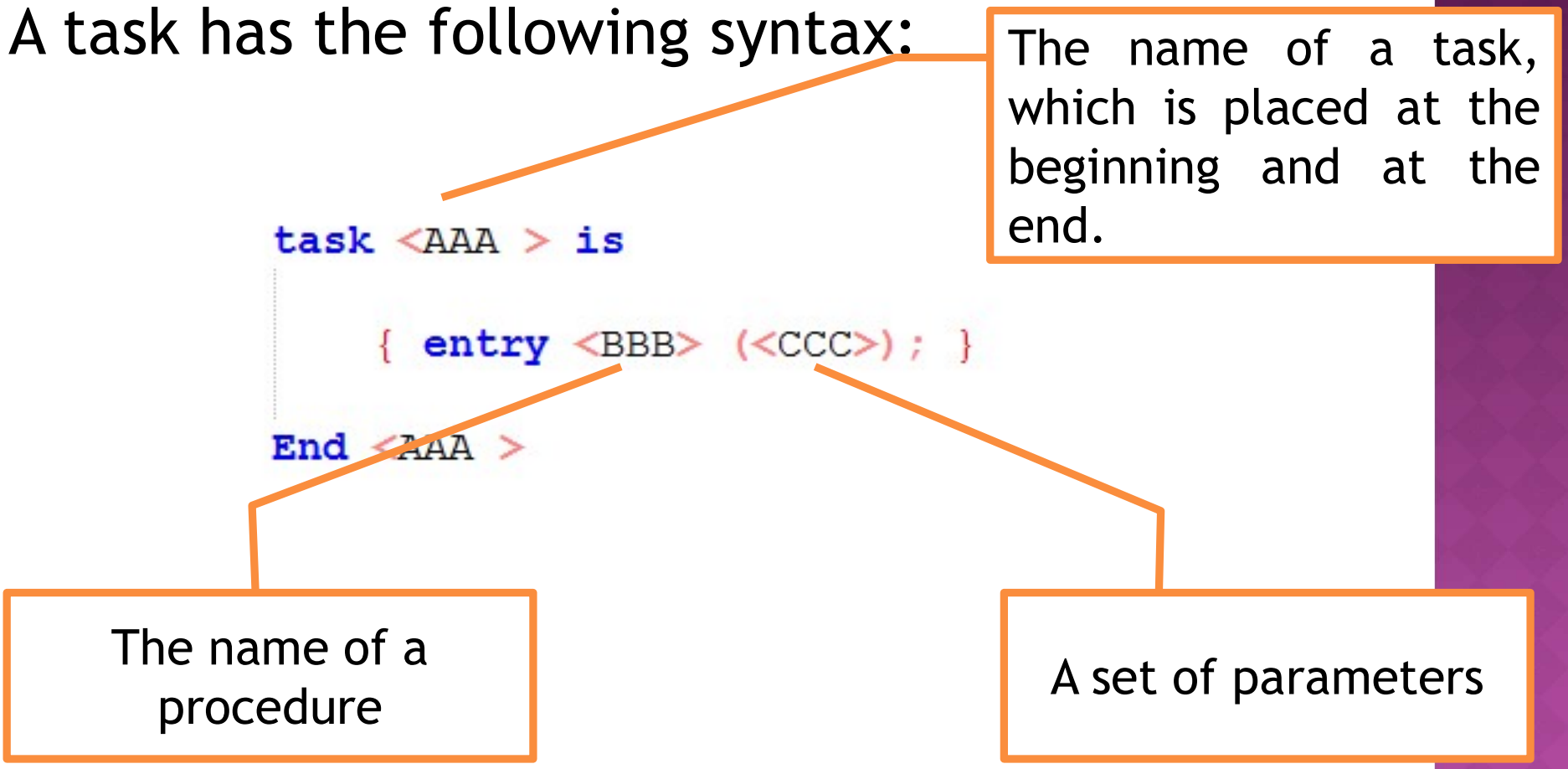
Ada

COMMUNICATION IN HIGH-LEVEL LANGUAGES - ADA-

- **Ada** is a programming language inspired by Pascal, from which it inherited its syntax and architecture.
- It provides a suitable model for distributed systems by supporting multitasking programming.
- The basic concept used in Ada is the **task**, which represents an activity consisting of a sequence of instructions.
- In Ada, tasks are described directly using the language's syntax, unlike in Java or CSP.
- Mechanisms related to concurrency, communication, and synchronization are integrated directly into the language.

COMMUNICATION IN HIGH-LEVEL LANGUAGES - ADA-

- A task has the following syntax:



A procedure of a task is called as follows:
<AAA>.<BBB>(<CCC>);

COMMUNICATION IN HIGH-LEVEL LANGUAGES - ADA-

- The Producer/Consumer Problem in Ada:
  - The solution is expressed in terms of three distinct tasks:
    - Two tasks, Proc and Cons, to simulate the execution behavior of the producer and consumer processes.
    - A third task to manage synchronization during simultaneous access to the shared buffer.

```
task body PC is
   -- Represents the number of empty slots
   Empty : integer := MaxSize;
begin
   loop
      select
         -- If there is at least one empty slot,
         -- accept a deposit
         when Empty > 0 =>
            accept DepositObject(msg : in T) do
               deposit_in_buffer(msg);
            end DepositObject;
            Free := Free - 1;
```

```
         -- If there is at least one filled slot, accept a
         -- withdrawal
               or
               when Empty < MaxSize =>
                  accept WithdrawObject(msg : out T) do
                     msg := withdraw_from_buffer();
                  end WithdrawObject;
                  Empty := Empty + 1;
      end select;
   end loop;
end PC;
```

```
task body Prod is
   -- Declaration section (if needed)
   message : T;
begin
   loop
      -- Produce a new message
      message := Produce(message);

      -- Deposit the message into the shared
      -- buffer via the PC task
      PC.DepositObject(message);

   end loop;
end Prod;
```

```
task body Cons is
   -- Declaration section (if needed)
   message : T;
begin
   loop
      -- Retrieve a message from the
      -- shared buffer via the PC task
      PC.WithdrawObject(message);

      -- Consume the retrieved message
      Consume(message);

   end loop;
end Cons;
```

THANK YOU FOR YOUR ATTENTION

NEXT CHAPTER

DEADLOCK