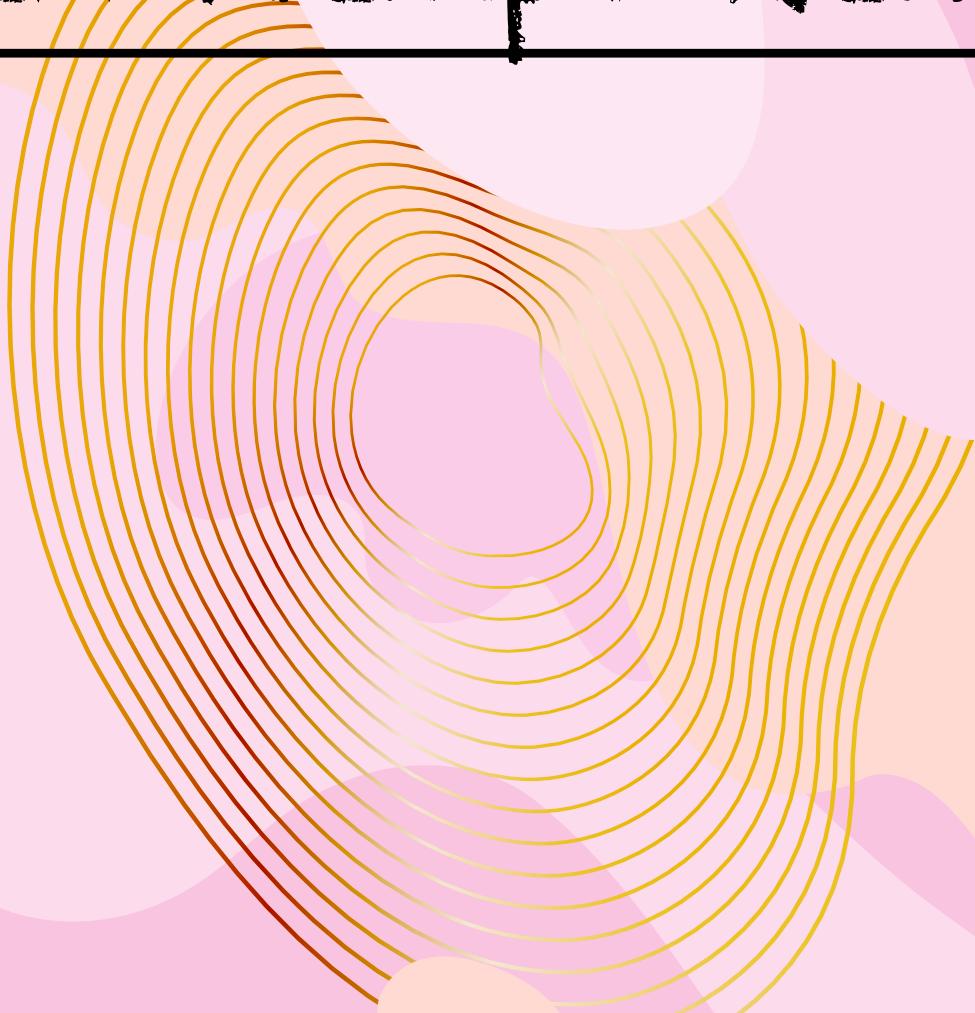


# Programación Concurrente y en Tiempo Real



# Unit 1

# Introduction and Basic Concepts

- 1.1 Baseline definitions**
- 1.2 Benefits and issues of concurrency**
- 1.3 Correctness**
- 1.4 Atomic statements and volatile variables**
- 1.5 Specification of Concurrent Execution**
- 1.6 Processes vs. Threads**
- 1.7 Architectures providing concurrency**
- 1.8 Java Threads**
- 1.9 Pascal FC**

- 1.1 Baseline definitions**
- 1.2 Benefits and issues of concurrency
- 1.3 Correctness
- 1.4 Atomic statements and volatile variables
- 1.5 Specification of Concurrent Execution
- 1.6 Processes vs. Threads
- 1.7 Architectures providing concurrency
- 1.8 Java Threads
- 1.9 Pascal FC

# 1.1 Baseline definitions

- A **program** is a set of instructions written in one or several files.

( static )

```
def process_row(row):
    first = row[0].strip()
    last = row[1].strip()
    address = row[2].strip()

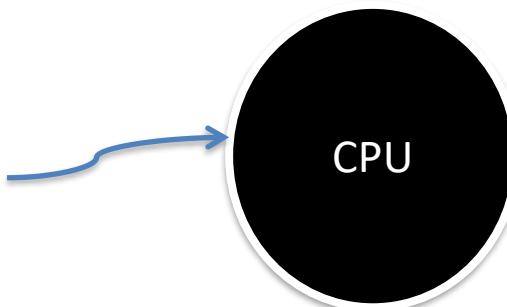
    try:
        (first, middle) = first.split(' ')
        (address, middle) = first.split(' ')
    except ValueError:
        pass

    print first, last
    return (first, last, address)

def process_file(f):
    voters = []
    for row in f:
        try:
            if len(row) > 0:
                (first, last, address) = process_row(row)
                first = first.upper()
                last = last.upper()
                voters.append((first, last, address))
        except IndexError:
            print "Error in row"
            print row
            raise
    print voters
```

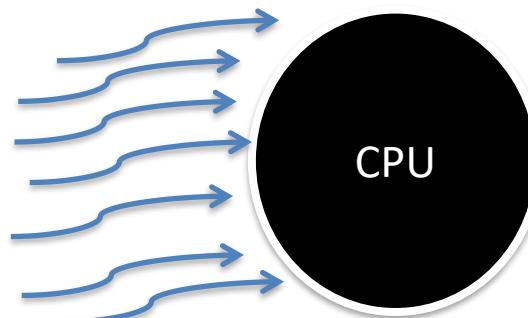
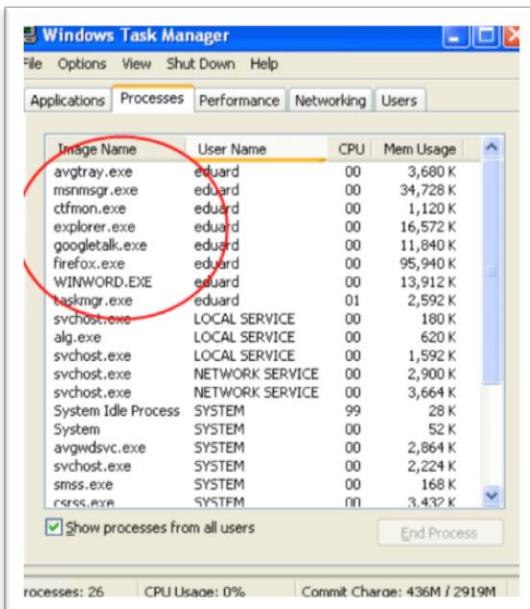
- When you **compile and run the program**, you **create a process** which is the **dynamic execution** of the compiled instructions in the CPU.

( dynamic )



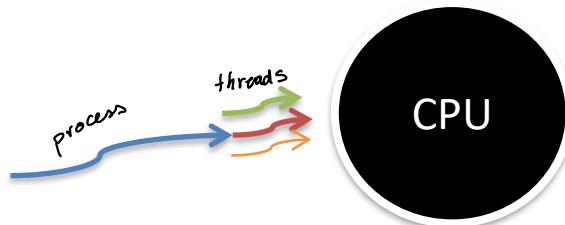
# 1.1 Baseline definitions

- The process needs some help to run successfully: Program Counter (PC), CPU registers, stack, stack pointer, main memory for global variables.
- Each time you run a compiled program, you create one new process with new PC, stack, registers and memory addresses.
- So you have one process running in your computer per program and instance:  
**Multitask Operating Systems.**



# 1.1 Baseline definitions

- Furthermore, each process may contain several sub-processes or threads.

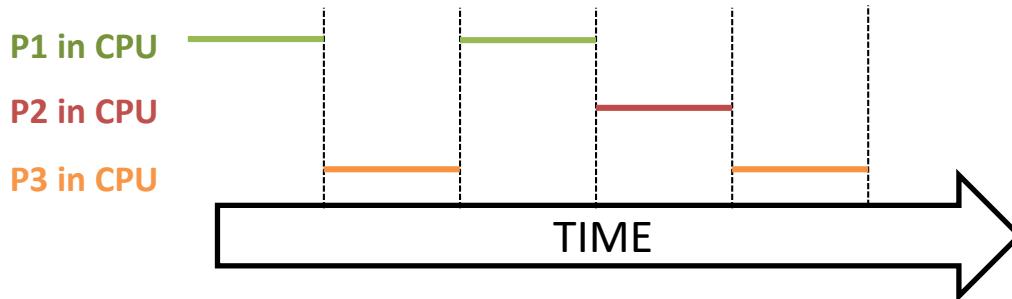


- These threads share the same main memory for global variables. But they still have different PC, stack and registers.
- Blue process: Intelligence test
  - Green thread: display questions and get input
  - Read thread: timer
  - Orange thread: pop-ups advertising

# 1.1 Baseline definitions

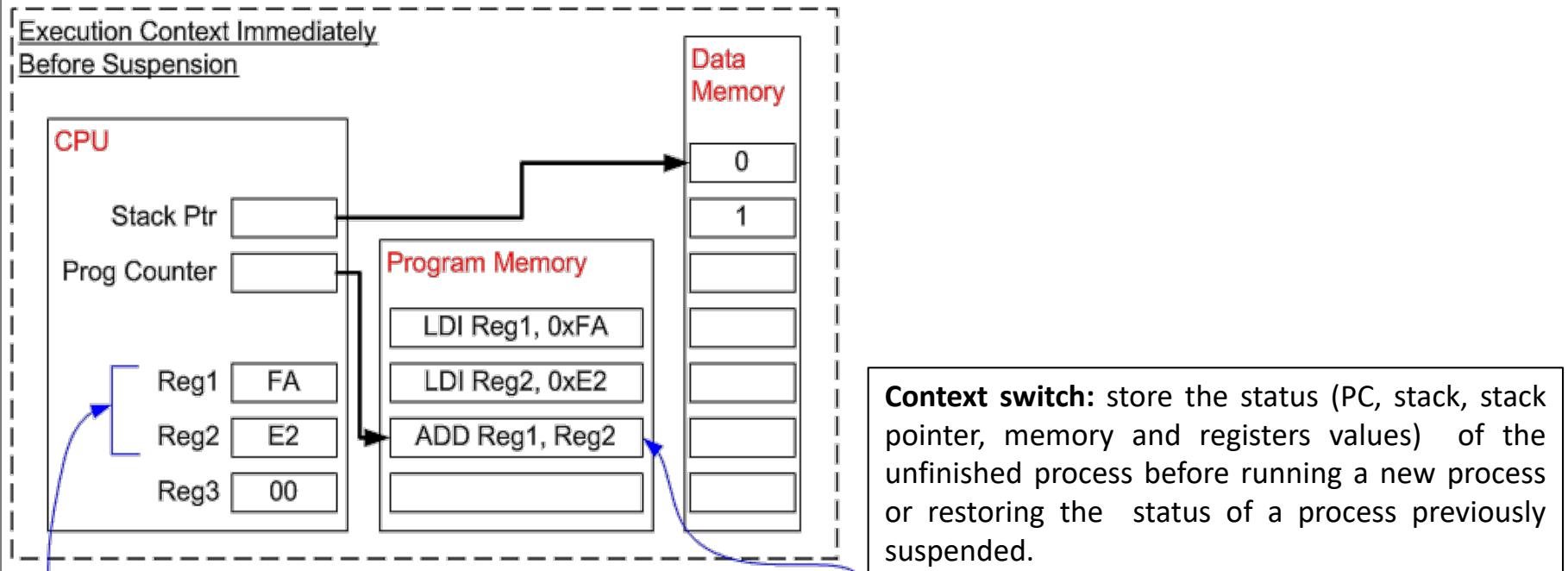
( interleaving )

- The **scheduler** is a built-in process in the kernel of the operating system which decides when a process/thread enters or exits the CPU.



- Each period of time during which a process is granted the CPU is called **time-slice**.
- A process may not be finished when the scheduler decides that another process must enter in CPU. So what about the unfinished process?
  - Can the new running process use the same memory addresses or registers?
  - Should it start from the beginning again once it re-enters the CPU?

# 1.1 Baseline definitions



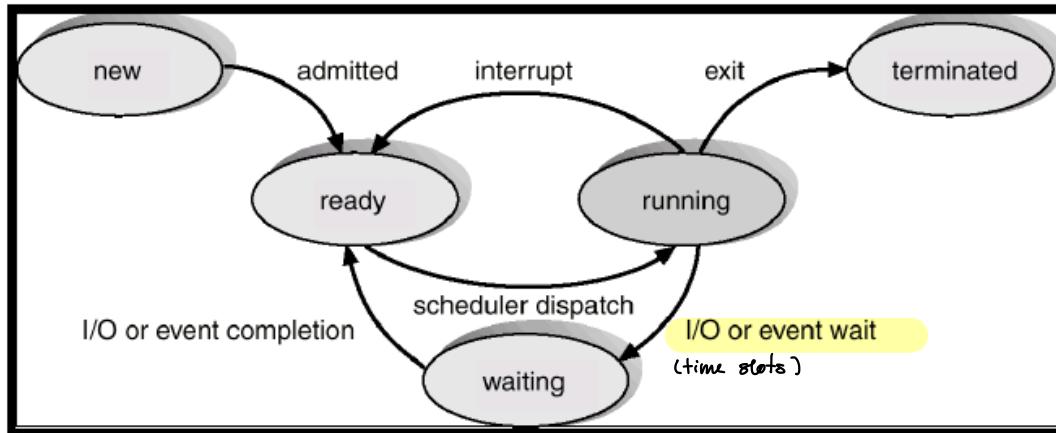
The task gets suspended as it is about to execute an ADD.

The previous instructions have already set the registers used by the ADD. When the task is resumed the ADD instruction will be the first instruction to execute. The task will not know if a different task modified Reg1 or Reg2 in the interim.

*Image obtained from [www.freertos.org](http://www.freertos.org)*

# 1.1 Baseline definitions

- Thus, a process may be running, waiting, stopped... A generic state diagram for a process is:



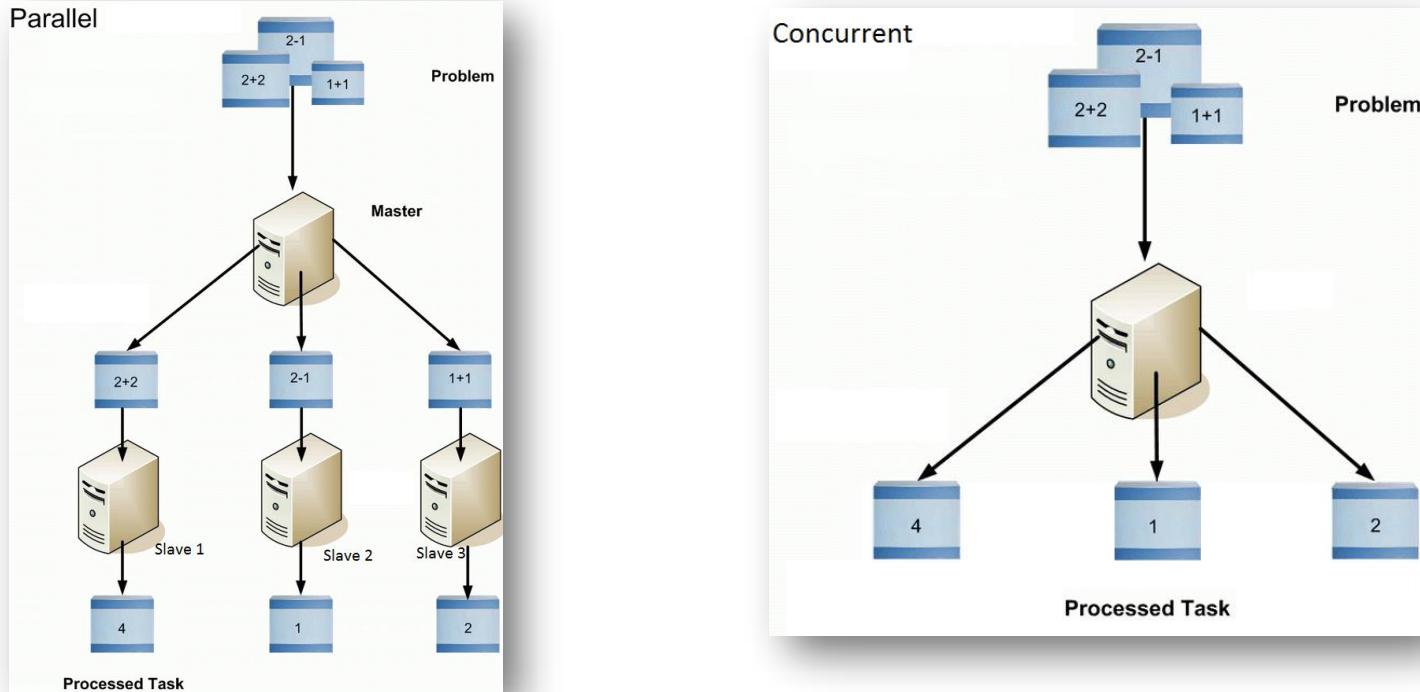
- **Ready:** the process wants to start/resume its execution.
- **Running:** the process is inside CPU and the current context is its own.
- **Waiting:** the process is waiting for a signal, or it has been interrupted by an I/O device.
- **Admitted:** the process enters in the ready-queue managed by the OS scheduler
- **Interrupt:** the scheduler decides to give a new time-slice to another process.
- **Sch. dispatch:** the scheduler gives a new time-slice to a ready process.

All transitions to or from *running* perform a context switch.

# 1.1 Baseline definitions

- CPUs have such a high clock-frequency that we are not usually aware of context switches, thus having the abstraction or the appearance that all processes and I/O are being executed at the same time: operating system, spreadsheet, word processor, internet browser,...
- **Parallel execution of processes:** two or more processes running at the same time (one CPU per process). Physically simultaneous processing.
- **Concurrent execution of processes:** two or more processes which need to share at least one CPU to run their machine code instructions. Logically simultaneous processing.
- **Concurrent programming:** application of techniques which help us manage the underlying problems arising from concurrent execution: mutual exclusions and synchronization of processes .
- The same definitions hold for execution of threads or sub-processes.
- Your lab assignments apply concurrent programming of threads. That is, 1 process with several threads which share 1 CPU.

# 1.1 Baseline definitions

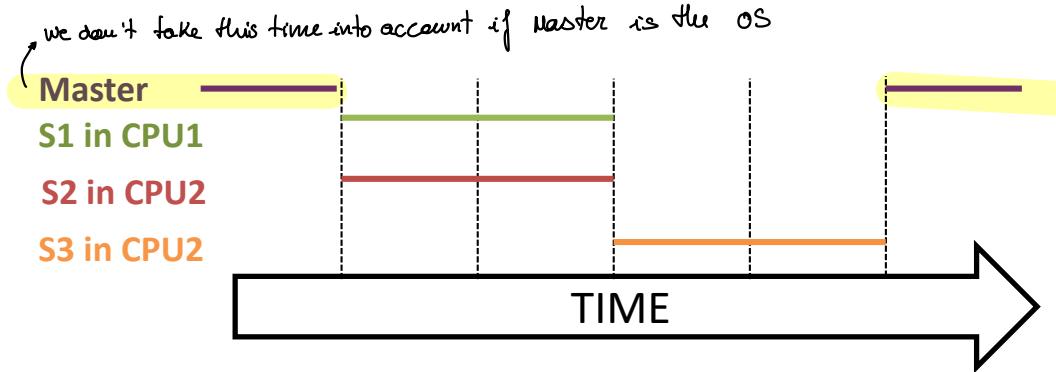


- In parallel computation, memory may be shared or not. In this example, it is not.
- Concurrent computation always uses shared memory.
- Whenever memory is **shared**, access must be controlled.
- When processes work together to solve a given problem, **synchronization** is necessary.
- From now on, with *concurrent* we refer to any computation with shared memory or need for synchronization.

- 1.1 Baseline definitions**
- 1.2 Benefits and issues of concurrency**
- 1.3 Correctness**
- 1.4 Atomic statements and volatile variables**
- 1.5 Specification of Concurrent Execution**
- 1.6 Processes vs. Threads**
- 1.7 Architectures providing concurrency**
- 1.8 Java Threads**
- 1.9 Pascal FC**

# 1.2 Benefits and issues of concurrency

- **Performance** gain from multiprocessing hardware
  - Several slave processes and one master which merges the results.
  - E.g.: In the previous example of parallel computation, if we only had 2 CPUs for slave processes:



- If we only had 1 CPU, the processing would take 2 extra time-slices.

# 1.2 Benefits and issues of concurrency

- Increased application **throughput**
  - an I/O call blocks the thread in the corresponding CPU without delaying the others.
  - E.g.: A process for a video-game with 2 threads: moving the enemy and moving Player 1. The enemy thread is always working, while the thread for player 1 is blocked waiting for the input device (mouse, keyboard, sensor,...)



# 1.2 Benefits and issues of concurrency

- Increased application **responsiveness**
  - high priority threads.
  - E.g.: the main thread is not responding and it would never release the CPU, but a higher priority thread gains access and gives us the opportunity to stop it.
- More **appropriate structure** (*not always necessary / not always a benefit*)
  - for programs which interact with the environment (sensors, data analysis, alarms), control multiple activities and handle multiple events (chats).
  - Databases: several readers, one writer.



# 1.2 Benefits and issues of concurrency

- Depending on the programming techniques, the resulting high-level code may result very **difficult to understand or maintain.**
- **Error-prone:** as you will find in your lab assignments, many of the concurrent programming techniques used for controlling shared-memory or synchronization can be distributed along all your code, making it difficult to get rid of execution-time errors:
  - E.g.: access to memory is never unlocked
  - E.g.: signals for synchronization do not reach the appropriate process
- **Indeterminism** in execution: the scheduler cannot be controlled, so:
  - **Interleaving**
    - we cannot predict the **execution order for threads**
    - we cannot predict how many lines of code a thread will run in each time-slice
  - Thus, different runs of a concurrent program produces different interleaving of its threads access to CPU.
  - But, if the concurrency is correctly controlled/programmed, the result is always the same.
  - Program expecting the worst case: each line of code is interleaved.

# 1.2 Benefits and issues of concurrency

- Example of indeterminism in execution without controlling access to memory:

| Shared memory |               |
|---------------|---------------|
| int x;        |               |
| Process p     | Process q     |
| x = 2         | x = 1         |
| int y = x + 3 | int z = x + 4 |
| print y       | print z       |

- Instructions p1 and q1 are atomic: 1 machine instruction after compilation.
- Instructions p2 and q2 are not atomic: 3 machine instructions: LOAD, ADD and STORE which can be interleaved due to the scheduler. But they do not affect x.
- Try to understand why the possible resulting values for variable y are: 5 and 4.
- Try to understand why the possible number of context switches are: 1 to 9.

# 1.2 Benefits and issues of concurrency

- Example of indeterminism in execution controlling access to memory:

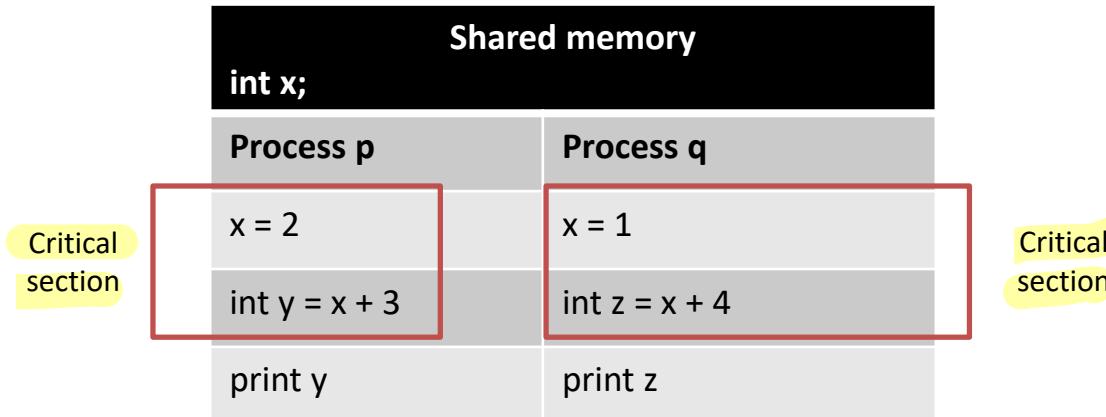
| Shared memory             |                           |
|---------------------------|---------------------------|
| int x;                    |                           |
| Process p                 | Process q                 |
| <i>lock access to x</i>   | <i>lock access to x</i>   |
| x = 2                     | x = 1                     |
| int y = x + 3             | int z = x + 4             |
| <i>unlock access to x</i> | <i>unlock access to x</i> |
| print y                   | print z                   |

- Try to understand now why y is always 5.
- When one process holds the lock, the other is suspended until the lock is released.
- Number of (working) context switches: 1 to 3. (switch to a process which cannot hold the lock, makes it be suspended and switch again)
- The order is still non-deterministic, but the result is always the same.

# 1.2 Benefits and issues of concurrency

*safe section*

- **Critical section**: it is the portion of code which accesses a shared resource which might be changed by some process.
- If more than 1 process is running code from the critical section, unexpected results occur.
- In the previous example, without controlling access to memory, the critical section is composed of those instructions which access variable x.



- 1.1 Baseline definitions**
- 1.2 Benefits and issues of concurrency**
- 1.3 Correctness**
- 1.4 Atomic statements and volatile variables
- 1.5 Specification of Concurrent Execution
- 1.6 Processes vs. Threads
- 1.7 Architectures providing concurrency
- 1.8 Java Threads
- 1.9 Pascal FC

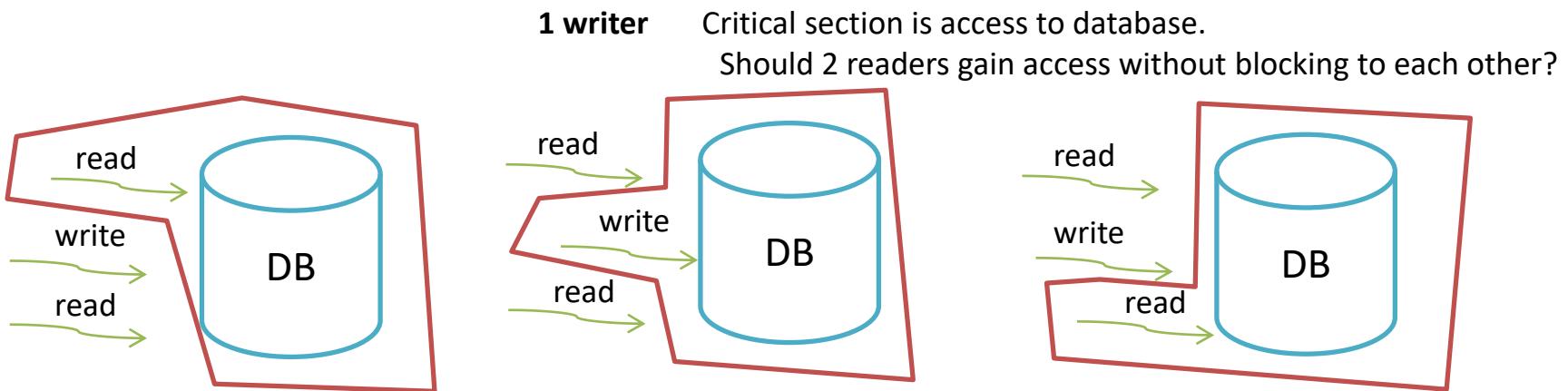
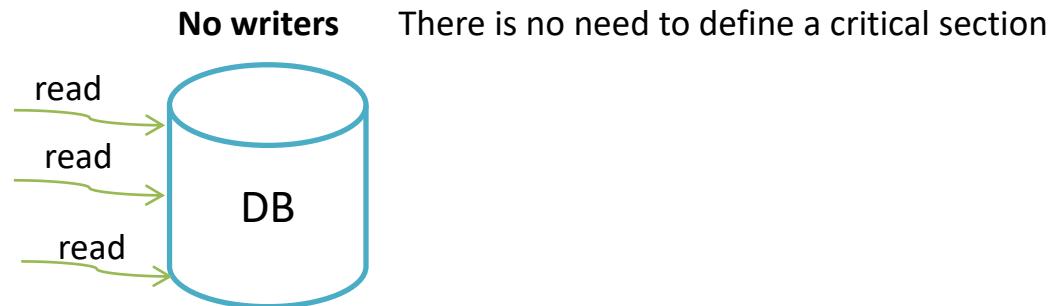
# 1.3 Correctness

- As we said, concurrent programming is error-prone.
- If you run your concurrent program once and you get the expected result, this does not mean it is correct. If you run several times, some error may appear due to a wrong protection of the critical section or wrong synchronization.
- Due to the unpredictability of interleaving:
  - it is impossible to debug a concurrent program to find the source of an error.
  - Correctness can only be proved with formal specification methods (out of the scope of this course).
- A concurrent program is correct if properties of safety (bad events never happen) and liveness (good things eventually happen) hold:
  - ↳ good things surely happen

# 1.3 Correctness

- Safety properties (or problems):

- 1) Achieve **Mutual exclusion**: only 1 process can be running inside the critical section.  
E.g. Several readers and writers and 1 database.



# 1.3 Correctness

- 2) Achieve **Synchronization**: when a process must wait for an I/O event or for another process to do something.

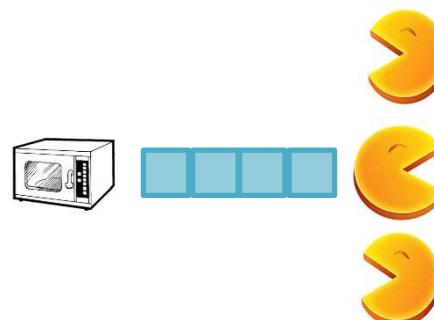
E.g.: Producer-Consumer problem: you cannot consume if the buffer is empty, and you cannot produce if the buffer is full.



Thread Consumer waits for the Producer to insert something into the buffer.  
If it does not wait --> **null pointer!**



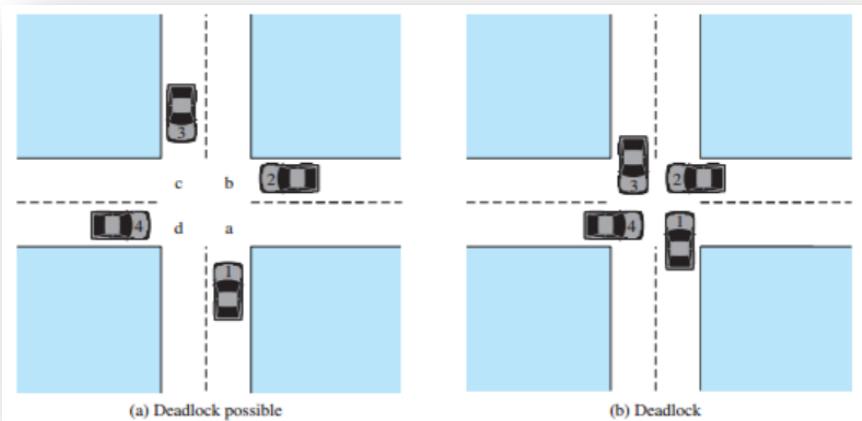
Thread Producer waits for the Consumer to take one element out of buffer.  
If it does not wait --> **index out of bounds!**



We may have several consumers... and even several producers.  
Mutual exclusion and Synchronization may get very confusing depending on the tools that our concurrent programming language provides.

# 1.3 Correctness

**3) Avoid Deadlock:** process A is waiting for process B to do something, but process B is waiting for process A (or a process which depends on A) to do another thing. Process A and B will be always waiting.



Car 1 cannot go on until 2 goes on  
Car 2 cannot go on until 3 goes on  
Car 3 cannot go on until 4 goes on  
Car 4 cannot go on until 1 goes on  
**DEADLOCK**

*Operating Systems: Internals and Design Principles. Ch. 6. W. Stalling.*

- Once in deadlock, processes are blocked forever.
- There is **no general solution for deadlock**. In programming time, you must think of possible deadlocks.
- Possible solutions:
  - Assign a maximum time of wait (cars go backwards after 10 seconds blocked)
  - Mutual exclusion (the cross-road is a critical section, so only 1 car is allowed at a time)
  - Assign a permanent order in which threads must gain access to the resource



Ashwani Gautam

Yesterday at 08:14

I: Explain us deadlock and we'll hire you

Me: Hire me and I'll explain it to you

# 1.3 Correctness

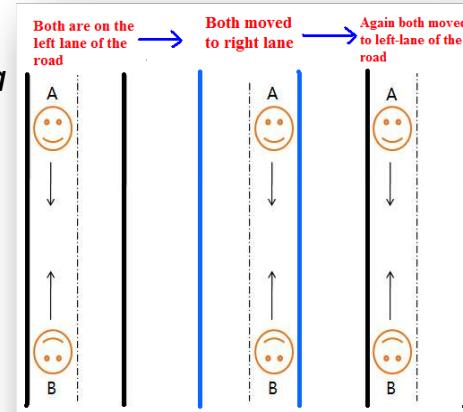
- The **safety** of a concurrent program is assured if it **always** meets these criteria:
  - our code guarantees mutual exclusion of critical sections,
  - processes are synchronized, and
  - processes never reach a deadlock status.

and then we say that our program is **thread-safe**.

- **Liveness properties (or problems):**

- 1) **Livelock:** two processes enter in livelock when they are doing some work responding to each other, but none of them makes any progress. E.g.:

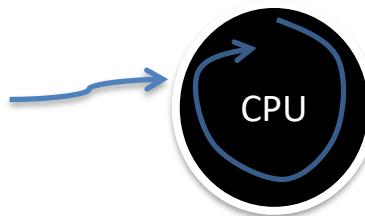
- *Thread A needs the light on to wake up, but thread B needs the light off to sleep. When A switches the light on, B switches it off, and then A switches on again. They will be looping forever.*
- *Two cars on a road, or two people walking through a corridor, moving left or right at the same time.*



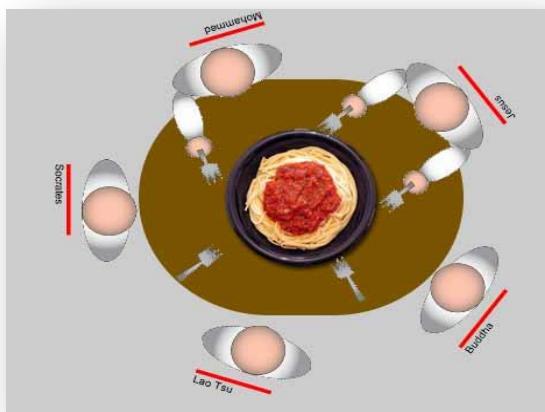
# 1.3 Correctness

**2) Starvation:** a process is virtually dead by starvation when it never gets access to CPU meanwhile other threads do. That is, it never gets out of the *ready* status. This may happen by several **reasons**:

- Other thread never releases the CPU

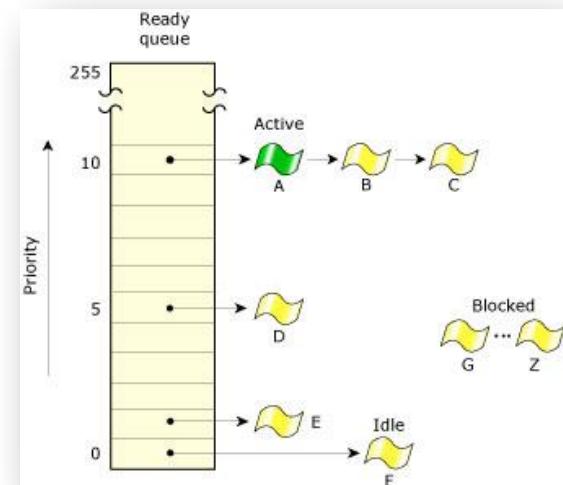


- Other thread never releases a resource which grants access to CPU (locks, semaphores,...)



[www.danfinlay.com/](http://www.danfinlay.com/)

- Other threads have higher priority



[www.qnx.com](http://www.qnx.com)

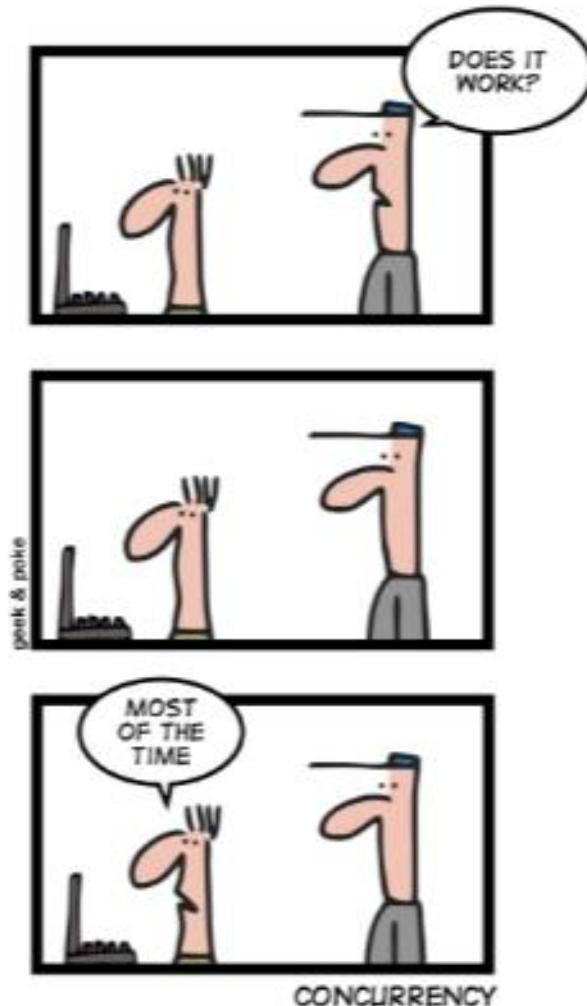
# 1.3 Correctness

- If our processes or threads may not gain access to CPU sometime due to livelock or starvation, then our program does not provide **liveness**.
- A concurrent program cannot (should not) be debugged using traditional methods, since liveness problems may happen from time to time. Moreover, the use of a **debugger makes changes in the scheduler** so they might never appear.
- **Careful design** is encouraged, as well as the use of the **highest-level tools** available to solve our concurrent problem.
- A program is **correct** if it is thread-safe and free of liveness problems.

# 1.3 Correctness

- If there is not any relation between the activity of 2 processes, we say they are **independent**.
- We can find two kinds of interaction between processes which might make correctness fail:
  - **Competence:** several processes must share common resources from the system (processor, memory, disk, printers,...), so they need to compete to get them. When the shared resource is a variable in memory, the competition is known as **race condition**. The value of such variable might be different depending of which process gains access to it first.
  - **Cooperation:** several processes must work on different parts of a problem to solve it together.

## SIMPLY EXPLAINED



- 1.1 Baseline definitions**
- 1.2 Benefits and issues of concurrency**
- 1.3 Correctness**
- 1.4 Atomic statements and volatile variables**
- 1.5 Specification of Concurrent Execution
- 1.6 Processes vs. Threads
- 1.7 Architectures providing concurrency
- 1.8 Java Threads
- 1.9 Pascal FC

# 1.4 Atomic and Volatile

- A statement is **atomic** if its resulting machine code is executed without any interleaving.

- An assignment like

$x = y$

is compiled to LOAD and WRITE instructions, so interleaving may happen.

- Some languages, like Java, assure us that **assignment** of integers and Boolean, and evaluation of boolean conditions are executed in an atomic manner.

$x = 3$  is atomic

$x = x + 3$  is not atomic!!

- A **combination** of atomic statements is not atomic!

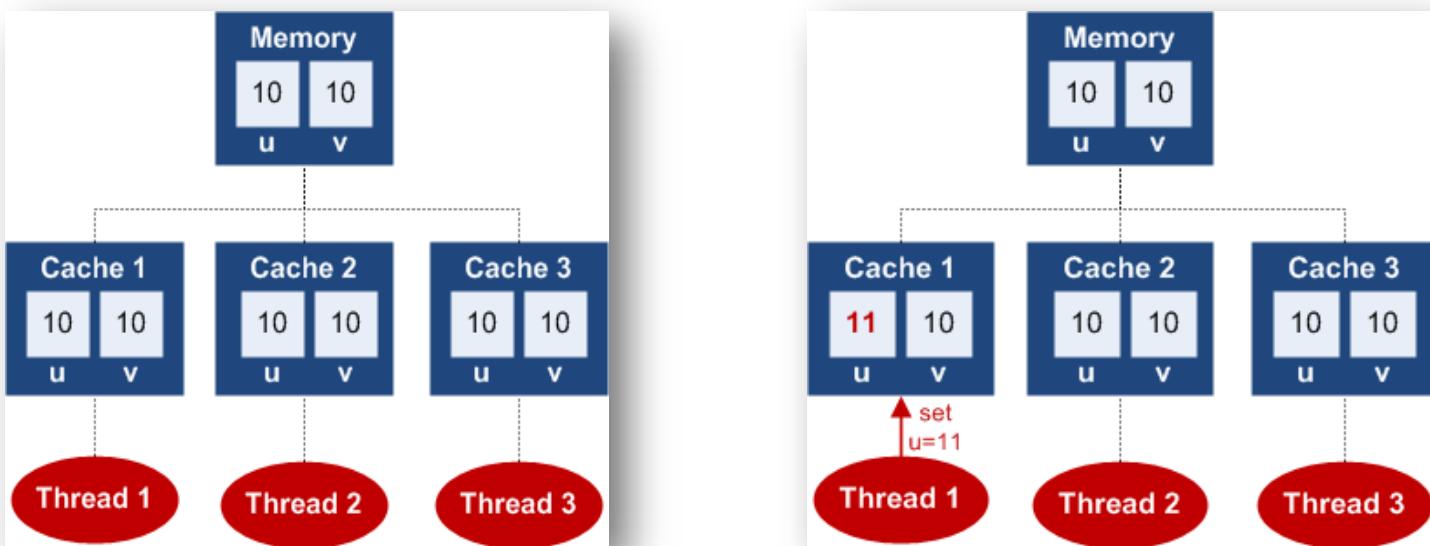
`if(condition) x = 3`

if *condition* is true, another thread may set it to false before x is set to 3.

- Assignment statements of **long** and **double** variables are not atomic! (they need 2 words to store their value, and words reading may suffer of interleaving)

# 1.4 Atomic and Volatile

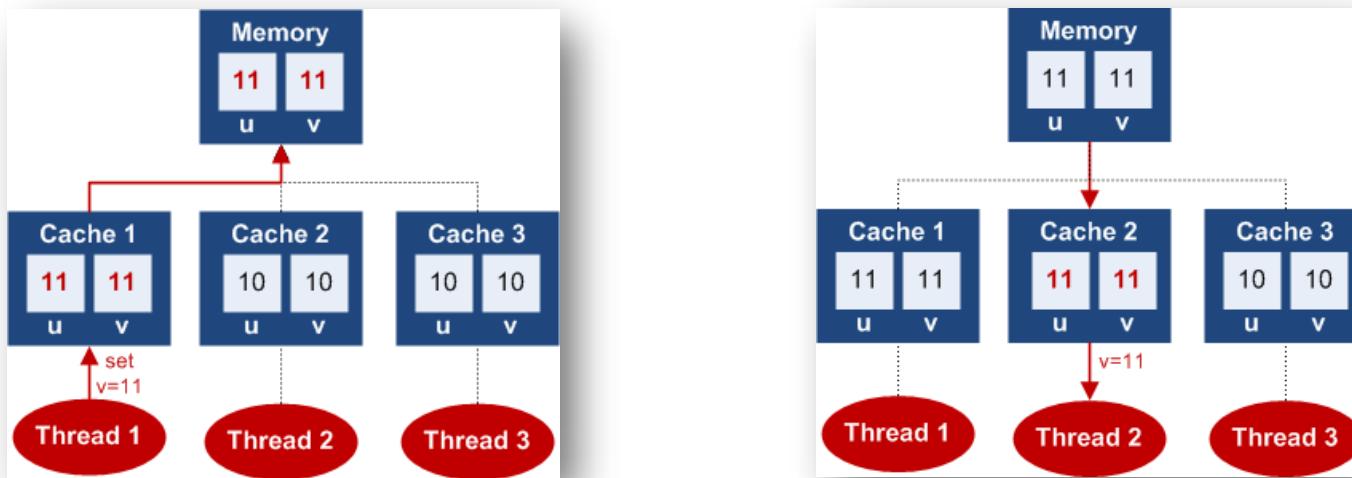
- Shared variables are first created in main memory.
- In compiling time, the compiler optimizes our code by making each process **cache a copy of all variables** or move them to registers.
- So, what happens if 3 threads work with a copy of a shared variable?



<http://igoro.com/>

# 1.4 Atomic and Volatile

- In order to avoid this, Java lets us declare variables as **volatile** to tell the compiler that a variable is accessed by 2 or more threads.
- Thus, sets and gets are now volatile. Some languages, like C#, make volatile reads and writes by default.
- Of course, this makes the computation with such variables slower because each access to it derives in reading or writing in main memory.



<http://igoro.com/>

- Variable *v* is volatile, but *u* is not. However, the set of a new value to *v* flushes all cache values in main memory. (*this is why sometimes your code may work even if you forget to define some variable as volatile*)
- A get call to *v* in Thread 2 flushes all main memory values in its cache.

# 1.4 Atomic and Volatile

- Atomic reads and writes of volatile variables are still atomic.
- Reads and writes of volatile long and double are now atomic.
- Imagine 10 processes running the following code on a volatile shared variable which is initiated as *double d=0* :

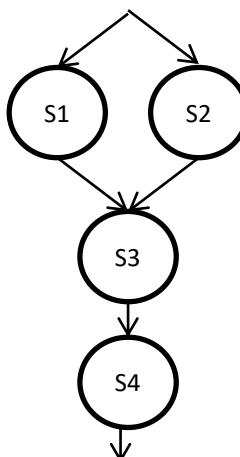
```
for(int i=0; i<5;i++) d++;
```
- Can we say that the resulting value of d is 50?

- 1.1 Baseline definitions**
- 1.2 Benefits and issues of concurrency**
- 1.3 Correctness**
- 1.4 Atomic statements and volatile variables**
- 1.5 Specification of Concurrent Execution**
- 1.6 Processes vs. Threads**
- 1.7 Architectures providing concurrency**
- 1.8 Java Threads**
- 1.9 Pascal FC**

# 1.5 Specification of Concurrent Execution

- There exist several methods to specify the order of execution of the instructions in our concurrent program. Two well-known methods are:
  - Precedence graphs
  - Cobegin-coend statements
- **Precedence Graphs:** an acyclic graph, in which a node represents a set of instructions. An arrow from node A to node B means that B cannot start until A ends. Two parallel nodes means they can be executed concurrently.

$S1 \rightarrow a := x + y;$   
 $S2 \rightarrow b := z - 1;$   
 $S3 \rightarrow c := a - b;$   
 $S4 \rightarrow w := c + 1;$



Palma et al. Ch. 1. 2003

# 1.5 Specification of Concurrent Execution

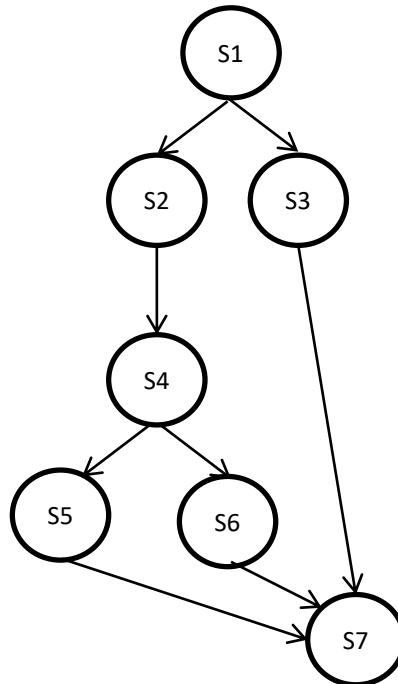
- **Cobegin/coend block:** instructions which can be executed in parallel are written inside a cobegin/coend block. Instructions inside these blocks can be run in any order (concurrently), the rest is run sequentially.

S1 →  $a := x + y;$   
S2 →  $b := z - 1;$   
S3 →  $c := a - b;$   
S4 →  $w := c + 1;$

```
begin
  cobegin
    a := x + y;
    b := z - 1;
  coend;
  c := a - b;
  w := c + 1;
end;
```

# 1.5 Specification of Concurrent Execution

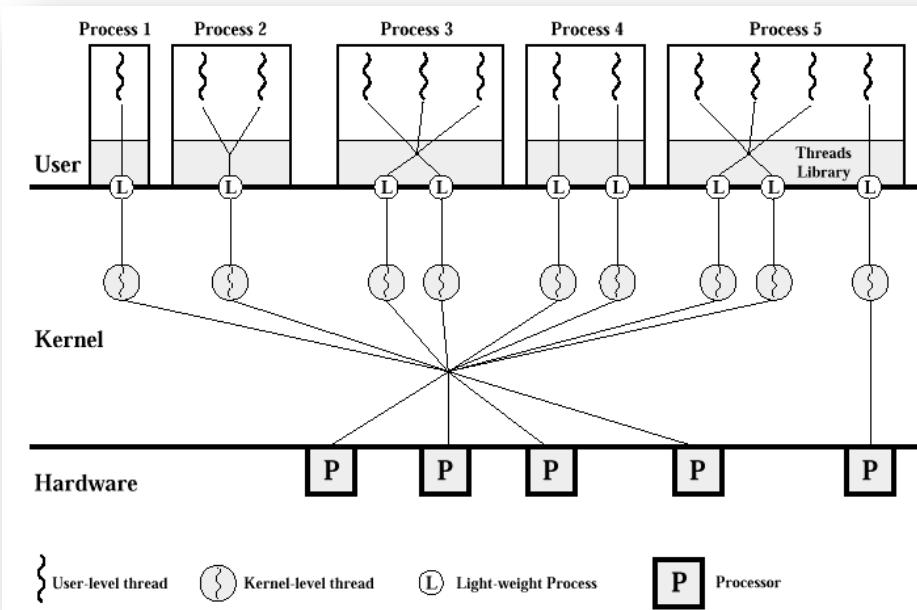
*Use the precedence graph to write the specification of concurrent execution with cobegin/coend statements.*



- 1.1 Baseline definitions**
- 1.2 Benefits and issues of concurrency**
- 1.3 Correctness**
- 1.4 Atomic statements and volatile variables**
- 1.5 Specification of Concurrent Execution**
- 1.6 Processes vs. Threads**
- 1.7 Architectures providing concurrency**
- 1.8 Java Threads**
- 1.9 Pascal FC**

# 1.6 Processes vs. Threads

- Processes are run by the operating system.
- **Threads** are independent running sequences inside a process.
- Both processes and threads can be run concurrently:
  - 1<sup>st</sup> level of concurrency: processes
  - 2<sup>nd</sup> level of concurrency: threads
- Context **switch is lighter in threads than in processes**:
  - Some of the context information of a process belongs to the OS kernel
  - All the information related to a thread belongs to the OS user space

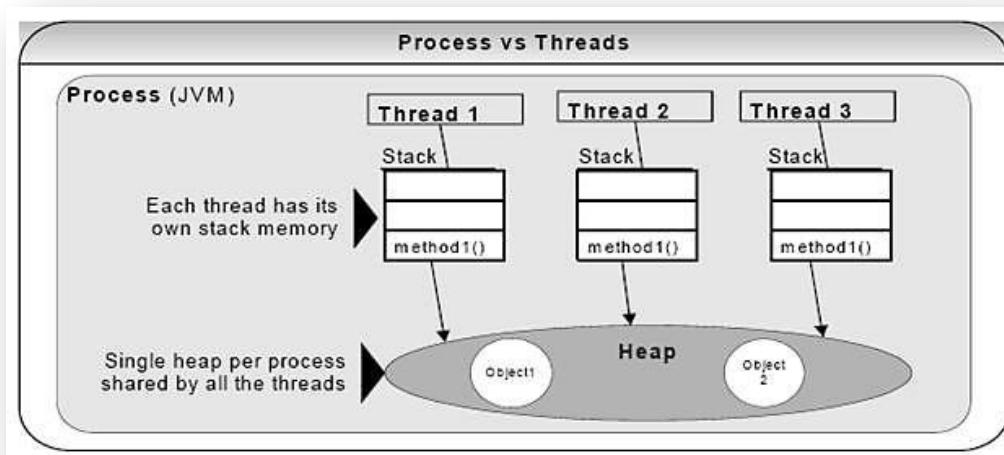


**1st level scheduling:** user threads compete for access to a kernel thread

**2nd level scheduling:** system threads compete for access to CPU

# 1.6 Processes vs. Threads

- Different **processes** use **different memory** addresses.
- **Threads** of the same process **share memory** addresses.
- A process allocates a **shared memory** space (heap space) for the **shared variables** of all its threads. Although each thread has its own **stack** (**local variables** from methods).

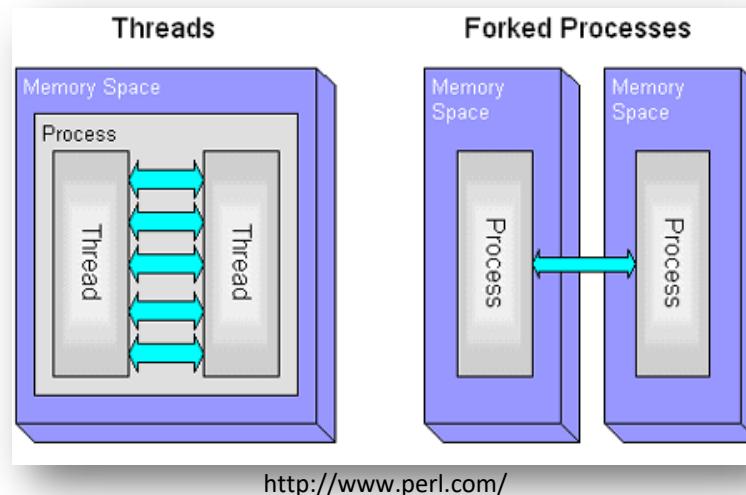


<http://www.java-forums.org/>

- Threads can **control** other threads (kill, create,...). Processes can only manage their children.
- Threads are also called *light weight processes*.

# 1.6 Processes vs. Threads

- Threads can communicate with each other directly (signals), while processes need calls to the operating systems, pipes, ...
- Children and parent threads share heap space.



- There are two **levels of threads-programming:**
  - System (kernel) threads
  - User threads

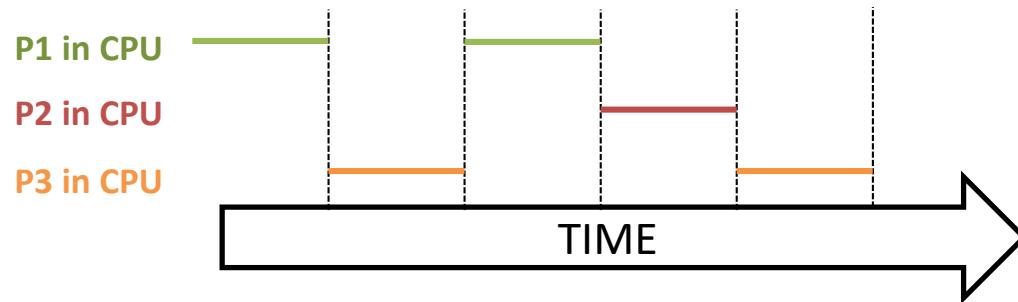
# 1.6 Processes vs. Threads

- **User threads:** threads created inside the user space of the OS. These are created from our high-level programming language and are used to create concurrent programs.
- **System threads:** threads provided by the operating system to give support to user threads. There exist 3 standards of system threads:
  - Win32 (proprietary), implemented in the OS kernel
  - OS/2 (proprietary), implemented in the OS kernel
  - POSIX (UNIX and Linux), implemented in the user space of the OS
- The way the programming language uses the native system threads is transparent for the developer.

- 1.1 Baseline definitions**
- 1.2 Benefits and issues of concurrency**
- 1.3 Correctness**
- 1.4 Atomic statements and volatile variables**
- 1.5 Specification of Concurrent Execution**
- 1.6 Processes vs. Threads**
- 1.7 Architectures providing concurrency**
- 1.8 Java Threads**
- 1.9 Pascal FC**

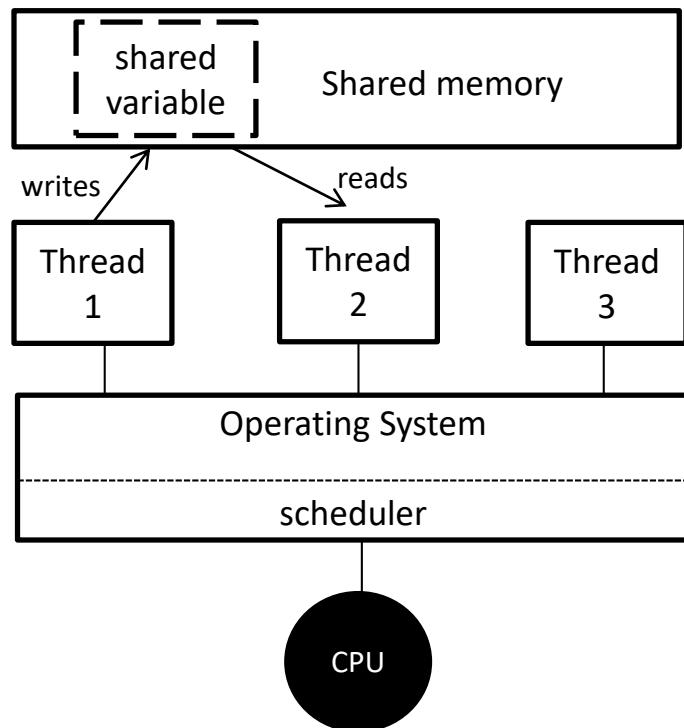
# 1.7 Architectures providing concurrency

- There are three kinds of hardware architecture which provide concurrency:
  - Uniprocessor: 1 computer with 1 processor
  - Multiprocessor: 1 computer with more than 1 processor
  - Distributed Systems: several computers (uni or multiprocessor) in a network.
- Uniprocessor:
  - Processes share the processor by interleaving.



# 1.7 Architectures providing concurrency

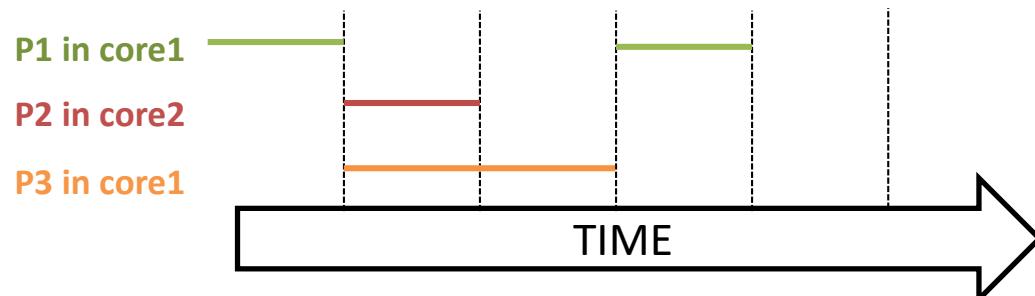
- Interleaving is controlled by the scheduler of the operating system
- Threads share the same memory: communication and synchronization is performed by shared variables.



*Concurrency in uniprocessor architecture*

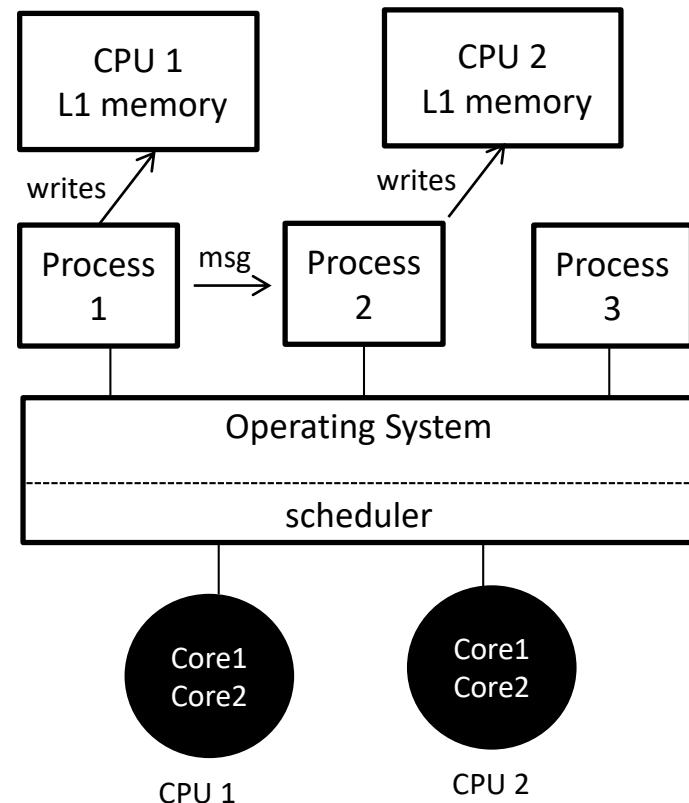
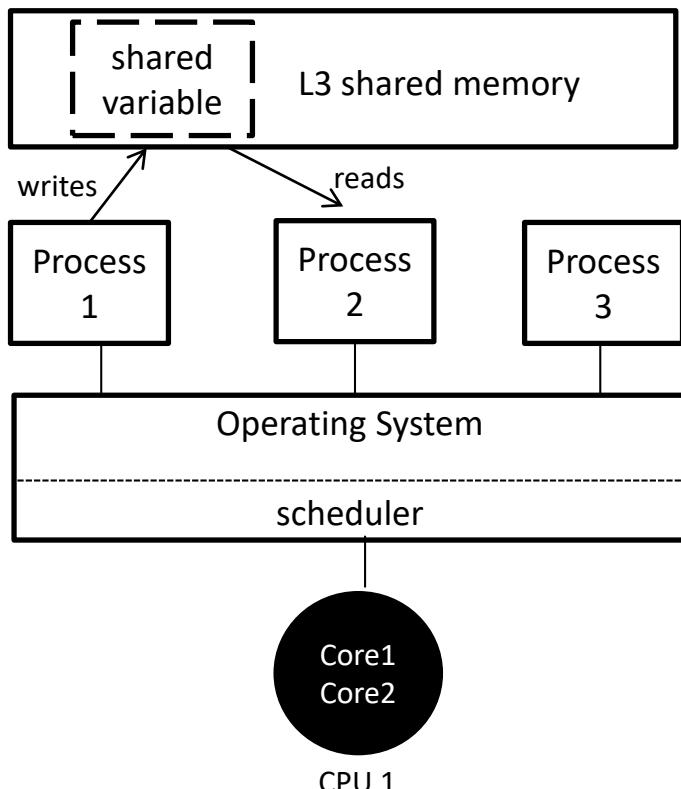
# 1.7 Architectures providing concurrency

- Multiprocessor and Multicore:
  - Now one processor has several cores, each one capable of running parallel instructions.  
They are integrated in one chip: multicore
  - There might be several multicore chips: multicore multiprocessor (cluster)
  - Real parallelism happens, but interleaving is still necessary (commonly, the number of processes is higher than the number of cores)



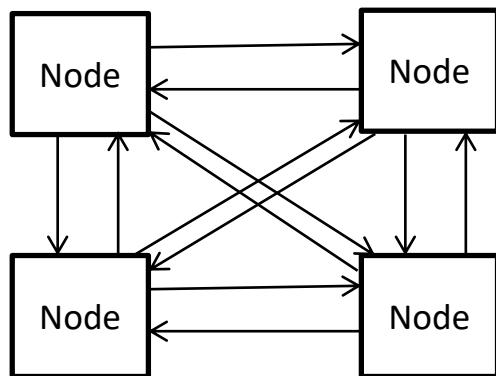
# 1.7 Architectures providing concurrency

- Cores in one processor may share the same memory or have different levels each one.
- Communication and synchronization may be performed by shared memory or message passing, depending on the architecture.



# 1.7 Architectures providing concurrency

- Distributed systems:
  - Nodes (processors) are connected to each other through a network
  - Communication and synchronization by message passing.
  - Each node may contain processors with different architecture or systems.
  - Parallelism and concurrency occur.



*Message passing in a distributed system*

# 1.7 Architectures providing concurrency

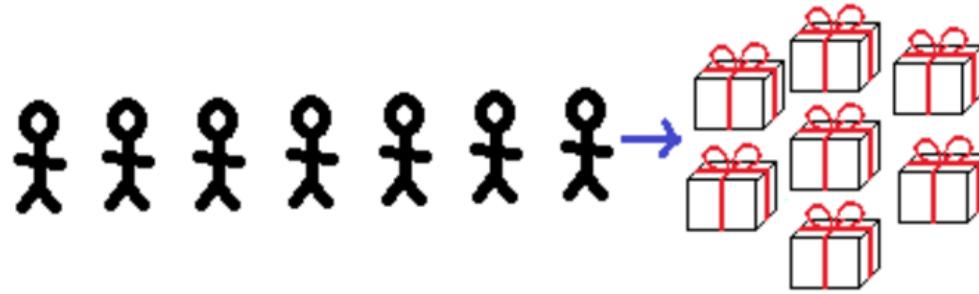
- Depending on the architecture, we define 3 **kinds of scheduling**:
  - **Multiprocessing**: several cores or processors are available, and shared memory is used. It happens in multicore and multiprocessor architecture.
  - **Distributed processing**: several cores or processors are available. It happens in distributed systems.
  - **Multiprogramming**: only one processing unit is available. Shared memory used, and it happens in uniprocessor architecture. Parallelism is not possible.

# 1.7 Architectures providing concurrency

- In order to avoid correctness problems, and to help us to get rid of low level details, we should implement concurrent programs from the **concurrent programming abstraction**:

*The execution of a concurrent program proceeds by executing a sequence of atomic statements obtained by random interleaving of the atomic statements of each process.*

- So we should think or assume that:
  - The final execution is a **single sequential program**, which is made of atomic statements of all processes, randomly interleaved.
  - Since there exist only 1 CPU, the **clock frequency is not important**.
  - And, most importantly, **INTERLEAVING MAY HAPPEN AT ANY TIME. THUS, USE CONCURRENT PROGRAMMING TOOLS TO PROTECT CRITICAL SECTIONS AND CORRECTLY PERFORM SYNCHRONIZATION.**



Concurrency: 7 kids queueing for presents from 1 heap



Parallelism: 7 kids getting 7 labeled presents, no queue

<http://yosefk.com>

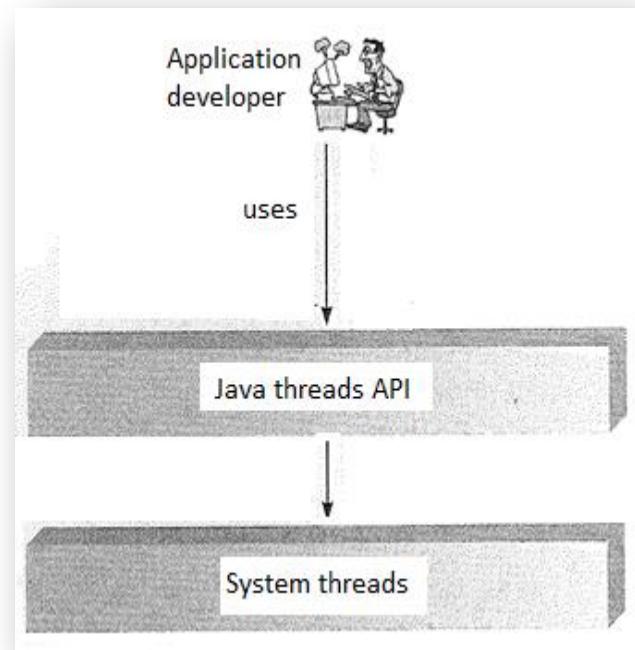
**join.quizizz.com**

**Game code: 473434**

- 1.1 Baseline definitions**
- 1.2 Benefits and issues of concurrency**
- 1.3 Correctness**
- 1.4 Atomic statements and volatile variables**
- 1.5 Specification of Concurrent Execution**
- 1.6 Processes vs. Threads**
- 1.7 Architectures providing concurrency**
- 1.8 Java Threads**
- 1.9 Pascal FC**

# 1.8 Java Threads

- Java threads are implemented on the Java Virtual Machine (JVM), which is built on the corresponding operating system threads.



Palma et al. Ch. 2. 2003

- JAVA makes concurrent programming possible without taking into account the underlying system threads library.

# 1.8 Java Threads

- When you run the main method in a Java program, you create one thread (**Main Thread**).
- From the main thread, you can create **new threads**. And from each new thread, you can create new threads as well.
- When the only existing thread is the Main Thread, you can be sure of the order of execution: **sequential**
- Once you create a second thread and start it, you can never know the order of execution of time-slices for the coexisting threads: you may find **indeterminism and non thread-safe** situations if you do not control access to critical sections and synchronization.

# 1.8 Java Threads

- In Java, a thread is represented by a `java.lang.Thread` object. The **two ways to create a thread** are:
  1. Instantiate a class which extends `java.lang.Thread` and overrides method `run()`
  2. Instantiate `Thread` passing as argument a class implementing method `run()` of interface `java.lang.Runnable`

```
1 package lectures.unit1;
2
3 public class HelloThread extends Thread {
4
5     public void run() {
6         System.out.println("Hello from a thread!");
7     }
8
9     public static void main(String args[]) {
10        (new HelloThread()).start();
11    }
12}
13 }
```

```
1 package lectures.unit1;
2
3 public class HelloRunnable implements Runnable {
4
5     public void run() {
6         System.out.println("Hello from a thread!");
7     }
8
9     public static void main(String args[]) {
10        (new Thread(new HelloRunnable())).start();
11    }
12}
13 }
```

- Extending `Thread` is more simple and intuitive, but your new class cannot extend anymore classes (Java does not allow multiple inheritance)
- The second method is more complex but your thread can extend another class.

# 1.8 Java Threads

- The official number of states for a Java thread is 6, since version 1.5:

| Enum Constant and Description  |
|--|
| <b>BLOCKED</b><br>Thread state for a thread blocked waiting for a monitor lock.          |
| <b>NEW</b><br>Thread state for a thread which has not yet started.                       |
| <b>RUNNABLE</b><br>Thread state for a runnable thread.                                   |
| <b>TERMINATED</b><br>Thread state for a terminated thread.                               |
| <b>TIMED_WAITING</b><br>Thread state for a waiting thread with a specified waiting time. |
| <b>WAITING</b><br>Thread state for a waiting thread.                                     |

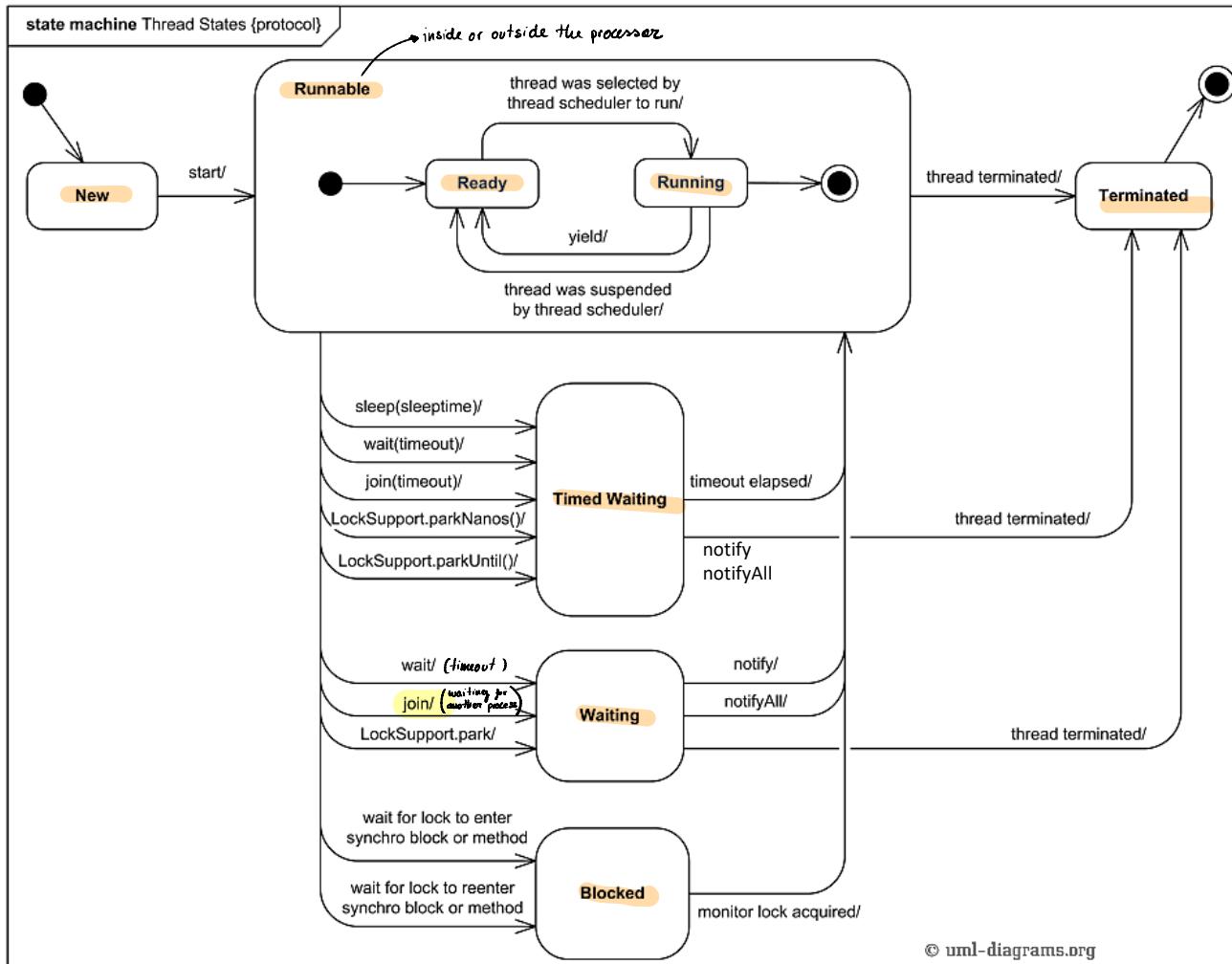
A hand-drawn bracket on the left side of the table groups the first three rows (BLOCKED, NEW, RUNNABLE). A hand-drawn arrow points from the RUNNABLE row to the right, pointing towards the handwritten notes. Handwritten notes to the right of the arrow say "inside processor" above "outside processor".

<https://docs.oracle.com/javase/9/docs/api/java/lang/Thread.State.html>

- These are the states inside the JVM. That is, we cannot ask if the thread is Running or Sleeping using method Thread.getState().

# 1.8 Java Threads

- The Java thread status does not let us always know how the thread reached it.



In Runnable, we do not know when it is actually running. We do not really need it, when it Runs your code it means it is running!

A thread can be requested to finish while waiting

A thread *waiting* is waiting for another thread to send a signal

A thread is blocked when waiting for another thread to release a lock

# 1.8 Java Threads

- We can **change the state** of a thread by calling **methods** provided by Thread and Object.
  - To wait state:
    - *sleep(milliseconds)*: current thread waits the given period of time
    - *join()*: current thread waits until the thread on which this method is called is terminated
    - *wait()*: current thread waits until it receives a signal from other thread.
  - To ready state:
    - *yield()*: current thread tells the scheduler that it wants to release the processor
  - Indicate that it should go to terminate state:
    - *interrupt()*: current thread sets the *interrupted* flag of the thread on which this method is called. The interrupted thread should check this flag and finish when it corresponds.
- Be careful, do not use deprecated methods!: *stop(), destroy(), suspend(), resume()*

# 1.8 Java Threads

- In order to call a method on a thread different to the current thread, we need to keep references when they are instantiated. Think what happens in the following examples:

[ interleaving ]

```
class extendedThread extends Thread{

    @Override
    public void run() {
        for(int i=0;i<1000;i++) {
            System.out.println("hi there from "+Thread.currentThread());
        }
    }
}

public class Calling {

    public static void main(String args[]) {

        extendedThread t1=new extendedThread();
        extendedThread t2=new extendedThread();

        t1.start();
        t2.start();

    }
}
```

# 1.8 Java Threads

[ JOIN ]

```
public static void main(String args[]) throws Exception{  
    extendedThread t1=new extendedThread();  
    extendedThread t2=new extendedThread();  
  
    t1.start();  
    t1.join(); (waits for t1 to finish)  
    t2.start();  
}  
}
```

```
class extendedThread extends Thread{  
  
    @Override  
    public void run(){  
        for(int i=0;i<1000 && !Thread.currentThread().isInterrupted();i++){  
            System.out.println("hi there from "+Thread.currentThread());  
        }  
    }  
}  
  
public class Calling {  
  
    public static void main(String args[]) throws Exception{  
  
        extendedThread t1=new extendedThread();  
        extendedThread t2=new extendedThread();  
  
        t1.start();  
        t2.start();  
        t1.interrupt();  
  
    }  
}
```

# 1.8 Java Threads

```
class extendedThread extends Thread{

    @Override
    public void run() {
        for(int i=0;i<1000;i++){
            System.out.println("hi there from "+Thread.currentThread());
            if(Thread.currentThread().getName().equals("T1")) yield();
        }
    }

    public class Calling {
        public static void main(String args[]) throws Exception{
            extendedThread t1=new extendedThread();
            extendedThread t2=new extendedThread();

            Thread.sleep(3000);
            t1.setName("T1");
            t2.setName("T2");
            t1.start();
            t2.start();
        }
    }
}
```

✓  
release the  
processor

# 1.8 Java Threads

- When you print a thread using `Thread.currentThread()` method, 3 values are printed: [name of thread, priority, threads group]
- The default priority is 5, which can be changed calling method `setPriority(int n)` being n from 1 to 10 (min to max priority).
  - *you cannot determine the order in which the sentences are running*
- In theory,
  - a thread with higher priority will gain access to CPU before a lower priority thread. However,
  - A ready thread will not switch context with a running higher-priority thread however, Java does not guarantee this is true at any moment.

- 1.1 Baseline definitions**
- 1.2 Benefits and issues of concurrency**
- 1.3 Correctness**
- 1.4 Atomic statements and volatile variables**
- 1.5 Specification of Concurrent Execution**
- 1.6 Processes vs. Threads**
- 1.7 Architectures providing concurrency**
- 1.8 Java Threads**
- 1.9 Pascal FC**

# 1.9 Pascal FC

- In this course, we will learn concurrent programming using Pascal-FC and Java.
- Pascal FC is based on Pascal, which is enhanced and reduced to support concurrent programming and to be used in educational contexts.
- Developed by Alan Burns and Geoff Davies, at the University of York.
- The official webpage maintained by the authors is

<http://www-users.cs.york.ac.uk/~burns/pf.html>



## Pascal-FC

by Alan Burns and Geoff Davies

# 1.9 Pascal FC

- Pascal-FC was developed to provide the most common tools to achieve correctness in our concurrent programming language, whose primitive commands or objects may not contain the desired tools.
- Program structure:

```
program name;  
(* global declarations:*)  
(* variables, processes, monitors,... *)  
  
begin  
(* statements *)  
end.
```

# 1.9 Pascal FC

- Declaration and use of 3 processes:

```
program threeprocesses;
  process type MYPROCESS(I : integer);
  begin          (thread)
    writeln(I);
  end;
var
  P1, P2, P3: MYPROCESS;
begin
  (*... statements executed sequentially*)
  cobegin
    P1(1);
    P2(2);
    P3(3);
  coend
  (* ... statements executed sequentially*)
end.
```

*concurrent => NO parallel*

Processes P1, P2 and P3:

- are type MYPROCESS
- are run concurrently (we do not know the order)
- cannot start until the sequential statements prior to **cobegin** are finished.
- must finish before the sequential statements after **coend** start.

# 1.9 Pascal FC

- Program which defines two processes. Each one prints its id 5 times.

```
program printID;
process First;
var
  i: integer;
begin
  for i:=1 to 5 do
    writeln(1);
end;
process Second;
var
  i: integer;
begin
  for i:=1 to 5 do
    writeln(2);
end;
begin
  writeln('This is executed sequentially');
  writeln('and the following cobegin/coend');
  writeln('block concurrently');
  cobegin
    First;
    Second;
  coend;
  writeln('When the 2 processes end, ');
  writeln(', this is run sequentially');
end.
```

Instead of defining the process type, since we only want 1 occurrence of each type, processes are defined directly.

Burns et al. Ch2. 1993.

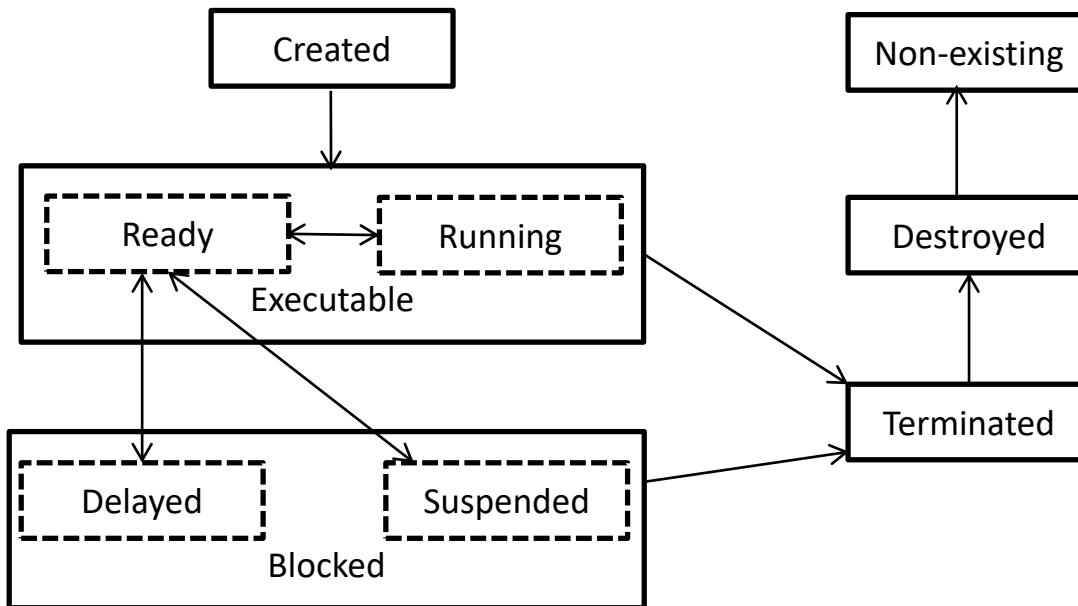
# 1.9 Pascal FC

- Modify the previous program so that we only need to define 1 kind of process: declare the type of process, instantiate as many as necessary, use parameters.

```
program print1D;  
process type myprocess(id: integer);  
var i: integer;  
begin  
for i := 1 to 5 do  
    writeLn(id);  
end;
```

# 1.9 Pascal FC

- States diagram of a process in Pascal-FC



A process is delayed by *sleep()*. It returns to Ready state after a given time.

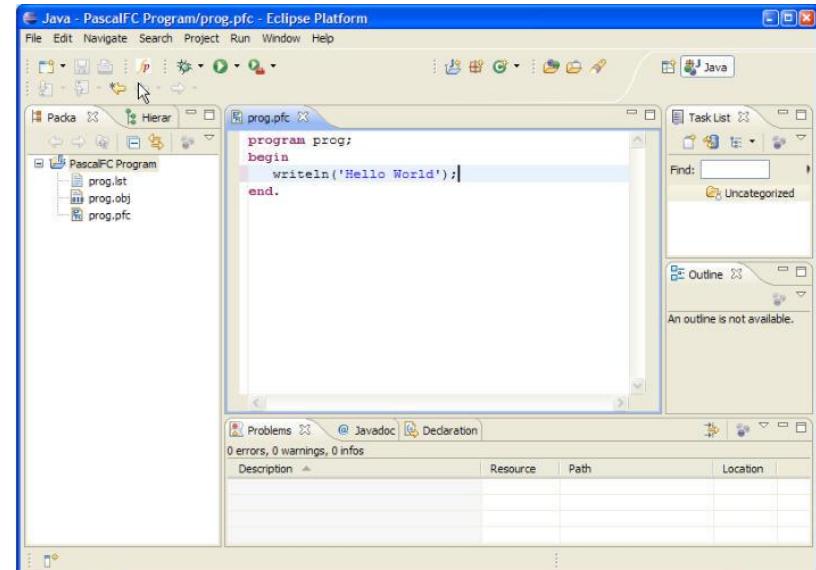
A process is suspended by calling a primitive which blocks it: read channel, request semaphore... **A suspended process can only go back to Ready state by the action of another process.** A suspended process is Terminated because it is selected as alternative (advanced topic).

# 1.9 Pascal FC

- Pascal-FC is designed to be run in OS without support to concurrency. In order to achieve this, it compiles all processes in **one single sequential program**.
- This means that if the code of one process halts, all the others halt. Do not misunderstand with state Blocked. By 'halt' we mean the process cannot go on, for example due to a deadlock problem or waiting for I/O which never happens.
- We can choose **2 kinds of execution**:
  - **Unfair** (without time-slices): one process cannot start until other is Terminated
  - **Fair** (time-slices): pieces of code are interleaved in the compiled *sequential* program.

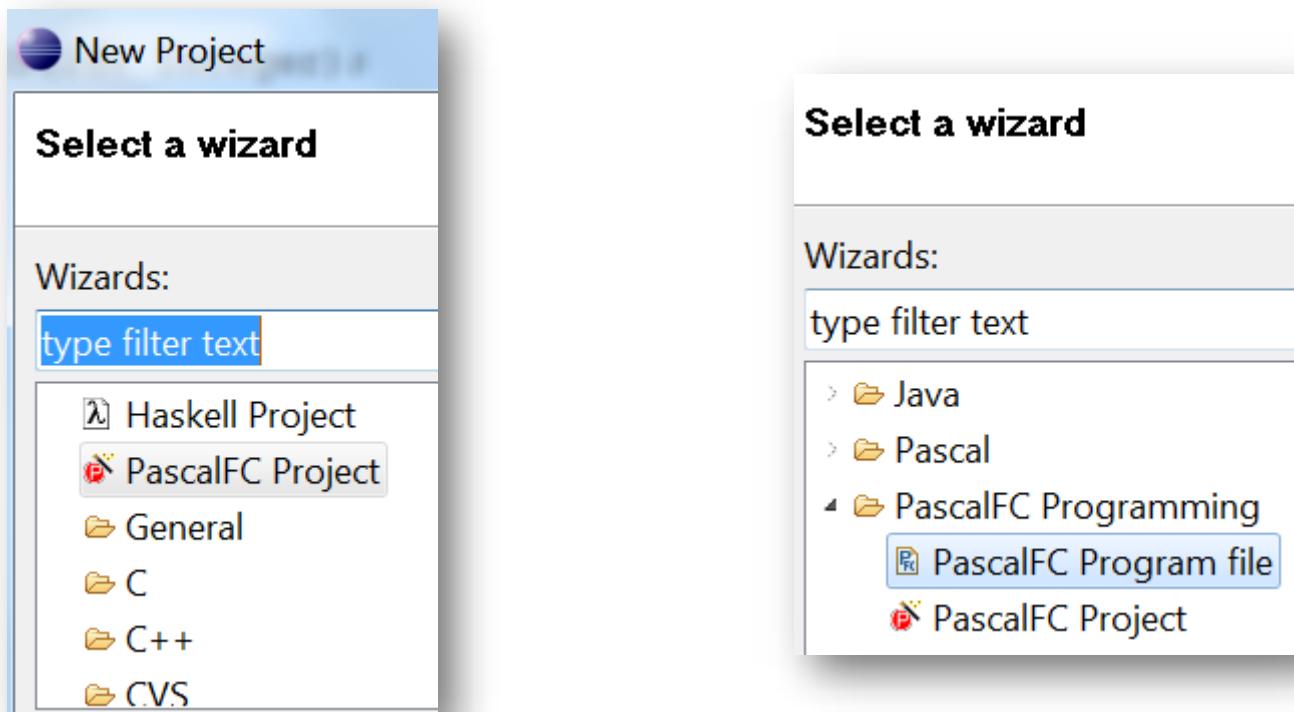
# 1.9 Pascal FC

- The official compiler and interpreter can be downloaded at <http://www-users.cs.york.ac.uk/~burns/pf.html>, but they work by command line.
- Compile and run Pascal FC programs using the Eclipse Gavab version. This is not available on its official webpage: <http://www.gavab.es/eclipsegavab>
- But you can browse the Internet to find it, for example at: <http://eclipsegavab.software.informer.com/2.0/>



# 1.9 Pascal FC

- Eclipse Gavab: new project of type PascalFC, then new File→Other→PascalFC Program file
- You can create folders in a project to sort your programs



# Potential Midterm Exam Questions

1. What is the operating system scheduler? Where is it running?

Is a built-in process in the **Kernel** of the OS which decides when a process / thread enters or exits the CPU.

Switching the context.

2. What is the difference between parallel and concurrent execution? Do you need to previously know the kind of execution when doing concurrent programming?

In parallel each process uses different hardware.

No, the difference is the HW that is used but we do not need to know that in advance

3. What do we mean when we say that concurrency implies indeterminism?

4. What is a critical section?

Two sections of code that are running concurrently can cause a deadlock (... mutual exclusion ...)

# Potential Midterm Exam Questions

5. Identify the critical section in this code:

```
class Counter {  
    int value;  
  
    Counter(int v) {  
        value = v;  
    }  
  
    public void increment() {  
        value++;  
    }  
  
}  
  
class LoopingThread extends Thread {  
    Counter counter;  
  
    LoopingThread(Counter c) {  
        counter = c;  
    }  
  
    @Override  
    public void run() {  
        try {  
            Thread.sleep(100);  
        } catch (InterruptedException e) {  
            e.printStackTrace();  
        }  
        for (int i = 0; i < 5; i++) {  
            counter.increment();  
        }  
    }  
}
```

```
public class Question5 {  
  
    public static void main(String args[]) {  
        Counter c = new Counter(0);  
        LoopingThread t1 = new LoopingThread(c);  
        LoopingThread t2 = new LoopingThread(c);  
        t1.start();  
        t2.start();  
    }  
}
```

# Potential Midterm Exam Questions

- 6. Identify the critical section in this code:

```
class LoopingThread extends Thread {
    int value;

    LoopingThread(int x) {
        value=x;
    }

    @Override
    public void run() {
        try {
            Thread.sleep(100);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        for (int i = 0; i < 5; i++) {
            value++;
        }
    }
}

public class Question6 {

    public static void main(String args[]) {
        int v=0;
        LoopingThread t1 = new LoopingThread(v);
        LoopingThread t2 = new LoopingThread(v);
        t1.start();
        t2.start();
    }
}
```

# Keywords phonetics

- synchronization /sɪnkrəneɪ'zeɪʃən/ 
- starvation /sta:'veɪʃən/ 
- initiate /ɪ'nɪʃeɪt/ 
- variable /'veəriəbl/ 
- instantiate /ɪn'stænʃeɪt/ 
- inheritance /ɪn'herɪtəns/ 
- yield /ji:ld/ 
- architecture /'a:kɪtekʃər/ 
- concurrent /kən'kʌrənt/ 
- precedence /'presədəns/ 
- instruction /in'strækshən/ 
- execution /eksi'kjueʃən/ 
- register /'redʒɪstər/ 
- cache /kæʃ/ 
- correctness /kə'rektnis/ 
- liveness /'laɪvnis/ 
- livelock /'laivlɒk/ 

# Unit 2

## Busy Wait Synchronization

→ Espera Activa

- 2.1 Introduction
- 2.2 Condition Synchronization
- 2.3 Mutual Exclusion
  - 2.3.1 First Attempt
  - 2.3.2 Second Attempt
  - 2.3.3 Third Attempt
  - 2.3.4 Fourth Attempt
  - 2.3.5 Dekker's Algorithm
  - 2.3.6 Several Critical Sections
- 2.4 Busy wait VS. passive wait
- 2.5 Conclusions

## **2.1 Introduction**

2.2 Condition Synchronization

2.3 Mutual Exclusion

    2.3.1 First Attempt

    2.3.2 Second Attempt

    2.3.3 Third Attempt

    2.3.4 Fourth Attempt

    2.3.5 Dekker's Algorithm

    2.3.6 Several Critical Sections

2.4 Busy wait VS. passive wait

2.5 Conclusions

## 2.1 Introduction

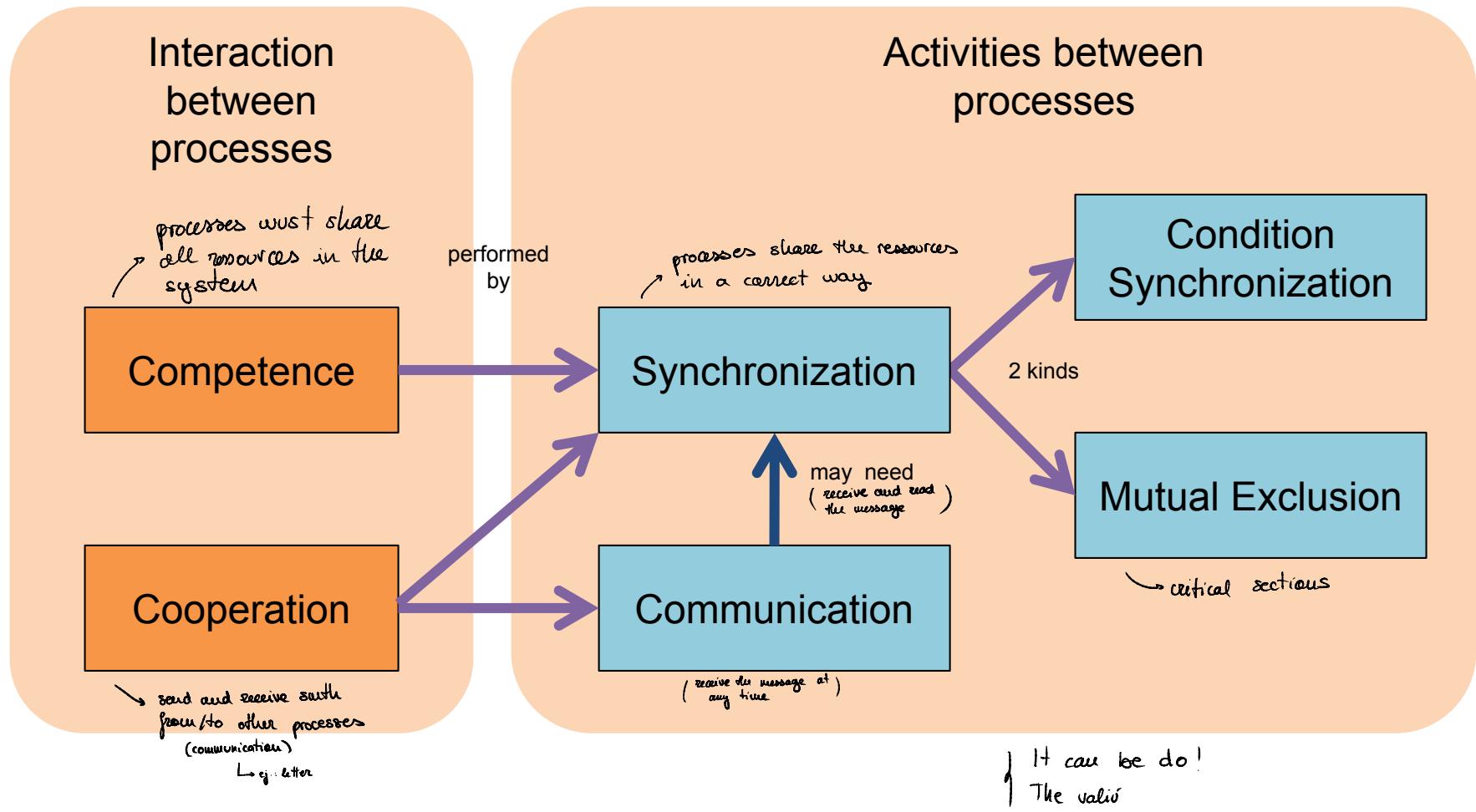
{  
• Interleaving  
• One processor

- Uniprocessors and multiprocessor/multicore use shared memory:
    - Shared variables can be written and read by several processes.
    - What happens if 2 processes read and/or write the same variable ‘at the same time’?
  - By the abstraction of concurrent programming, we must think that any interleaving is possible for all non-atomic statements.
  - Java and Pascal-FC provide atomic read/write operations in primitive variables (except double and long).
- . The OS can get the processor at any time  
↳ worst case : the process executing changes in the processor upon every instruction.

## 2.1 Introduction

- If 2 processes read a variable at the same time...  
→ both process read same value
- If 2 processes write at the same time (any after the other):  
→ the last written value remains, but we cannot predict it.
- If one process reads and the other writes at the same time (any after the other):  
→ the read value could be the former or the latter, but we cannot predict it.

# 2.1 Introduction



# 2.1 Introduction

- **Shared-memory communication:**
  - Shared variables let processes share information by reading and writing on them
- **Shared-memory synchronization:**
  - **Condition Synchronization:** depending on the value of one or more variables
  - **Mutual Exclusion:** use of structures or algorithms which provide exclusive access to critical sections.

## **2.1 Introduction**

## **2.2 Condition Synchronization**

### **2.3 Mutual Exclusion**

#### **2.3.1 First Attempt**

#### **2.3.2 Second Attempt**

#### **2.3.3 Third Attempt**

#### **2.3.4 Fourth Attempt**

#### **2.3.5 Dekker's Algorithm**

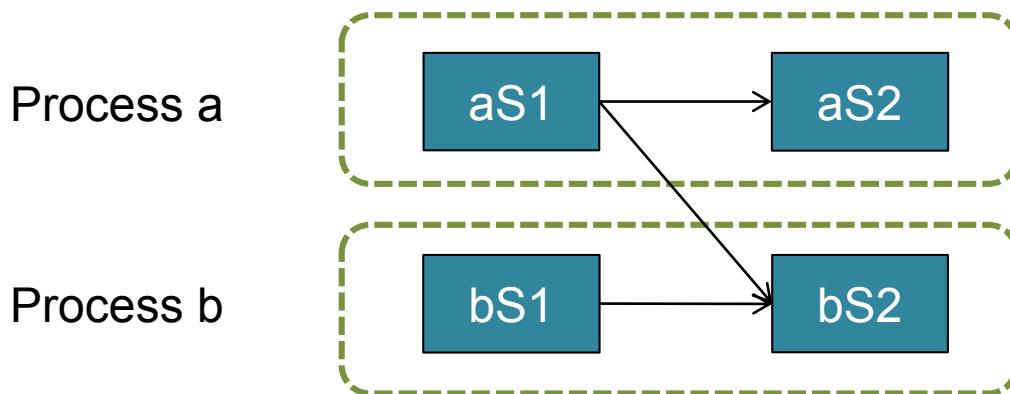
#### **2.3.6 Several Critical Sections**

### **2.4 Busy wait VS. passive wait**

### **2.5 Conclusions**

## 2.2 Condition Synchronization

- One process halts waiting for a condition to be met thanks to the action of another process.
- Let us consider the following precedence diagram of the statements of 2 processes:



## 2.2 Condition Synchronization

- Assume that the last instruction in each block of statements is to print the id of such block (in theory, `System.out.print` in Java provides atomic access to screen).

Some possible outputs:

aS1 aS2 bS1 bS2

aS1 bS1 aS2 bS2

bS1 aS1 aS2 bS2

bS1 aS1 aS2 bS2

...think more...

bS1 aS1 bS2 aS2

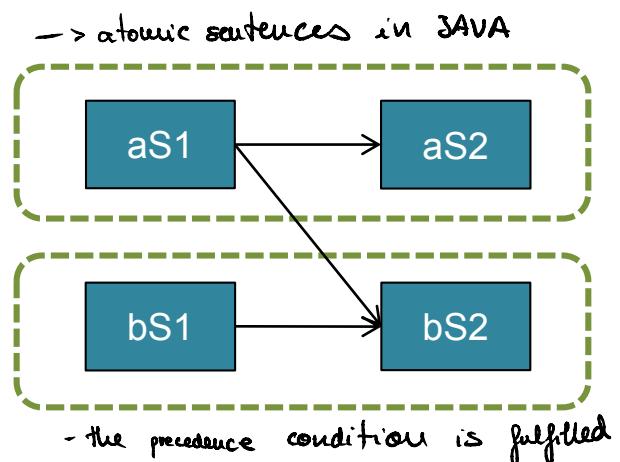
An impossible output:

bS1 bS2 aS1 aS2

→ precedence condition

Process a

Process b



# 2.2 Condition Synchronization

## Pascal-FC code

```
program conditSynch;

var
  continue: boolean;

process ProcessA;
begin
  write('aS1 ');
  continue := true;
  write('aS2 ');
end;

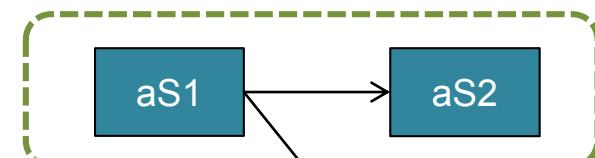
process ProcessB;
begin
  write('bS1 ');
  while not continue do;
  write('bS2 ');
end;

begin
  continue := false;
  cobegin
    ProcessA;
    ProcessB;
  coend
end.
```

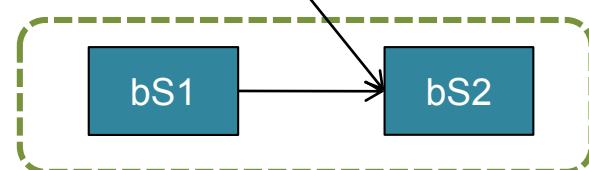
- Shared variable: *continue*
- The work done by *Process b* waiting for the condition to be set, is called **busy wait**. That is, waste processor time doing *nothing* until the other process sets the condition.

nothing  
busy wait/  
espera activa }  
The process is checking  
the value of a variable  
while doing nothing.

Process a



Process b



## 2.2 Condition Synchronization

- **Interleaving** of instructions to get the output:

aS1 bS1 aS2 bS2

processor  
time slice  
↑

|   | Process A         | Process B          | continue |
|---|-------------------|--------------------|----------|
| 1 | write('aS1');     |                    | false    |
| 2 |                   | write('bS1');      | false    |
| 3 | continue := true; |                    | true     |
| 4 | write('aS2');     |                    | true     |
| 5 |                   | while not continue | true     |
| 6 |                   | write('bS2');      | true     |

## 2.2 Condition Synchronization

- Interleaving of instructions to get the output:

bS1 aS1 aS2 bS2

|   | Process A                     | Process B                       | continue           |
|---|-------------------------------|---------------------------------|--------------------|
| 1 |                               | <code>write('bS1 ')</code>      | <code>false</code> |
| 2 |                               | <code>while not continue</code> | <code>false</code> |
| 3 |                               | <code>while not continue</code> | <code>false</code> |
| 4 |                               | <code>while not continue</code> | <code>false</code> |
| 5 | <code>write('aS1 ')</code>    |                                 | <code>false</code> |
| 6 | <code>continue:= true;</code> |                                 | <code>true</code>  |
| 7 | <code>write('aS2 ')</code>    |                                 | <code>true</code>  |
| 8 |                               | <code>while not continue</code> | <code>true</code>  |
| 9 |                               | <code>write('bS2 ')</code>      | <code>true</code>  |

busy  
wait

## 2.2 Condition Synchronization

- Interleaving of instructions to get the output: *complete the table*

| aS1 | aS2 | bS1 | bS2 |
|-----|-----|-----|-----|
|-----|-----|-----|-----|

|   | Process A         | Process B            | continue |
|---|-------------------|----------------------|----------|
| 1 | .write('aS1');    |                      | false    |
| 2 | continue := true; |                      | true     |
| 3 | .write('aS2');    |                      | true     |
| 4 |                   | write('bS1');        | true     |
| 5 |                   | .while not continue; | true     |
| 6 |                   | write('bS2');        | true     |

# 2.2 Condition Synchronization

JAVA code

```
class ThreadA extends Thread{  
  
    SharedObject s;  
  
    public ThreadA(SharedObject shared) {  
        s=shared;  
    }  
  
    public void run() {  
        System.out.println("aS1");  
        s.setGo();  
        System.out.println("aS2");  
    }  
}
```

```
class ThreadB extends Thread{  
  
    SharedObject s;  
  
    public ThreadB(SharedObject shared) {  
        s=shared;  
    }  
  
    public void run() {  
        System.out.println("bS1");  
        while(!s.getGo());  
        System.out.println("bS2");  
    }  
}
```

```
class SharedObject{  
  
    boolean go=false;  
  
    public boolean getGo() {  
        return go;  
    }  
  
    public void setGo() {  
        go=true;  
    }  
}
```

*Why isn't it necessary to add the volatile modifier to boolean go?*

```
public class CondSynch {  
  
    public static void main(String args[]) {  
        SharedObject s=new SharedObject();  
        (new ThreadA(s)).start();  
        (new ThreadB(s)).start();  
  
    }  
}
```

## **2.1 Introduction**

## **2.2 Condition Synchronization**

## **2.3 Mutual Exclusion**

**2.3.1 First Attempt**

**2.3.2 Second Attempt**

**2.3.3 Third Attempt**

**2.3.4 Fourth Attempt**

**2.3.5 Dekker's Algorithm**

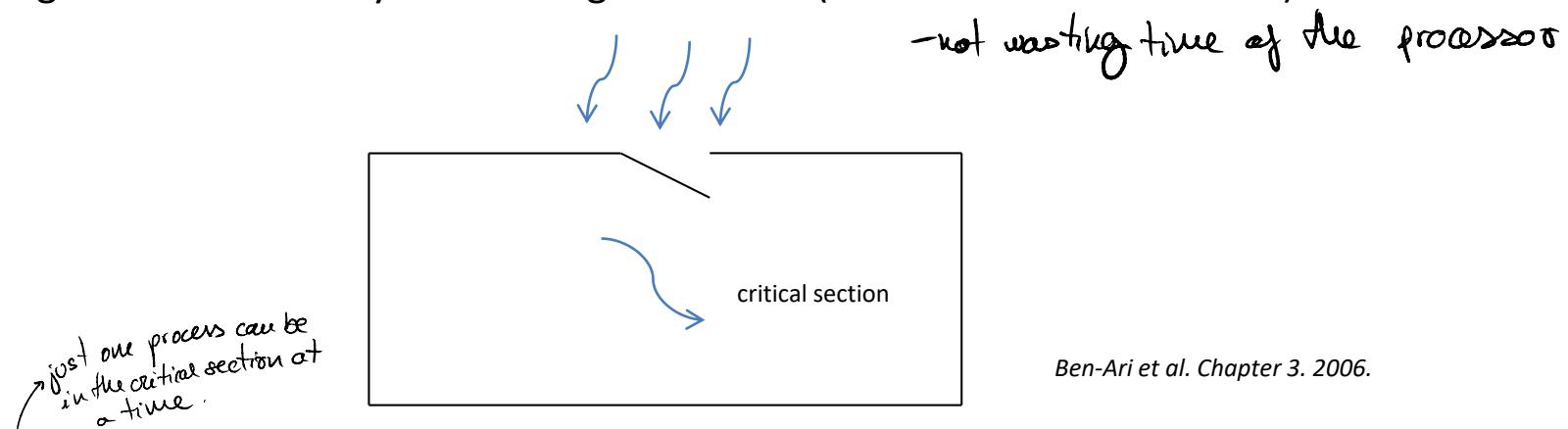
**2.3.6 Several Critical Sections**

## **2.4 Busy wait VS. passive wait**

## **2.5 Conclusions**

## 2.3 Mutual Exclusion

- When two or more processes need to access a shared variable, object or set of statements which require exclusive access, **processes need to be synchronized** in order to guarantee that only 1 of them gains access (enters the critical section).



Ben-Ari et al. Chapter 3. 2006.

- The **synchronization by mutual exclusion** is the execution of a set of instructions before entering the critical section (**pre-protocol**) and another set of instructions immediately after leaving the critical section (**post-protocol**).
- The **pre-protocol** guarantees mutual exclusion in the access.
- The **post-protocol** communicates to the other processes that the entrance to the critical section is now open.

## 2.3 Mutual Exclusion

- A **correct solution** of mutual exclusion fulfills:
  - Mutual exclusion is granted. → thread safety (avoid deadlocks)
  - Avoids livelock and starvation of processes trying to enter the critical section.
    - ↳ a process is doing something but it is not useful (! liveness) → avoid starvation
    - ↳ avoid losing time, not using unnecessary waiting
- And, it would be good that:
  - No variables used in the critical and non-critical sections are used in the protocols. That is, variables used in protocols are created for their **exclusive use** in protocols.
  - Pre and post-protocols should use **little** memory and CPU clock-time.
    - ↳ protocols should be as quick as we can.

## 2.3 Mutual Exclusion

- Solutions using protocols assume that the only atomic instructions available are read and write on primitive variables.
- In 1965, Dijkstra published a solution for mutual exclusion in the case of 2 processes. In order to explain it, he first presents 4 wrong approaches or attempts in which the most common errors of concurrent programming appear.
- The correct solution is based on a mathematician called Dekker, so Dijkstra called it Dekker's algorithm.
- Dijkstra improved Dekker's Algorithm for  $n > 1$  processes: Dijkstra's algorithm.
- The Eisenberg-McGuire's algorithm is an optimization of Dijkstra's algorithm.

- 2.1 Introduction**
- 2.2 Condition Synchronization**
- 2.3 Mutual Exclusion**
  - 2.3.1 First Attempt**
  - 2.3.2 Second Attempt**
  - 2.3.3 Third Attempt**
  - 2.3.4 Fourth Attempt**
  - 2.3.5 Dekker's Algorithm**
  - 2.3.6 Several Critical Sections**
- 2.4 Busy wait VS. passive wait**
- 2.5 Conclusions**

## 2.3 Mutual Exclusion

### 1<sup>st</sup> attempt (Pascal-FC)

- Based on busy-wait: processes share 1 variable to tell which process may enter in CS

```
program FirstAttempt;
var turn:Integer;

process P1;
begin
  repeat
    while turn <> 1 do;
      writeln('P1 is in CS');
      writeln('P1 is leaving CS');
      turn:=2;
      writeln('P1 is in non-CS');
    forever
  end;

process P2;
begin
  repeat
    while turn <> 2 do;
      writeln('P2 is in CS');
      writeln('P2 is leaving CS');
      turn:=1;
      writeln('P2 is in non-CS');
    forever
  end;
```

*critical section*

*pre-protocol*  
every process ask for the value of turn

```
begin
  turn:=1;
  cobegin
    P1;
    P2;
  coend
end.
```

*to know which process can go into the critical section*

Identify the pre-protocol, the Critical Section and the post-protocol in both processes

- > if both processes are running -> no starvation
- > problem: unnecessary waiting and mandatory alternance between processes

# 2.3 Mutual Exclusion

## 1<sup>st</sup> attempt

- Commonly, this approach would work but...
- The **1<sup>st</sup> attempt is not correct because:**
  - it is not free of starvation if one process fails.

|   | P1                                     | P2                                     | turn |
|---|--|--|------|
| 1 | <code>while turn &lt;&gt; 1 do;</code> |  | 1    |
| 2 | <code>writeln('P1 is in CS');</code>   |  | 1    |
| 3 |  | <code>while turn &lt;&gt; 2 do;</code> | 1    |
| 4 | <b><i>process 1 crashes!</i></b>       |  | 1    |
|   |  | <i>Remains forever in busy-wait</i>    | 1    |

- **Alternation is mandatory (it should not):** access to CS is granted in turns, so if a process is very slow (long non-CS) the other cannot enter the CS until the other changes the value of *turn*.

## 2.3 Mutual Exclusion

### 1<sup>st</sup> attempt (Java)

```
class CS1 {  
    int turn = 1;  
  
    public void enterCS(int ID) {  
        // pre-protocol  
        while (turn != ID);  
        // CS  
        System.out.println("P" + ID + " is in CS");  
        System.out.println("P" + ID + " is leaving CS");  
        //non-CS  
        // post-protocol  
        turn = 1-ID ;  
  
        System.out.println("P" + ID + " is in non-CS");  
    }  
}
```

```
class MyThread extends Thread{  
    CS1 sharedCS;  
    int ID;  
  
    MyThread(CS1 cs, int id){  
        sharedCS=cs;  
        ID=id;  
    }  
  
    public void run(){  
        while(true){  
            sharedCS.enterCS(ID);  
        }  
    }  
}
```

```
public class FirstAttempt {  
  
    public static void main(String args[]){  
        CS1 criticalSection=new CS1();  
  
        (new MyThread(criticalSection, 0)).start();  
        (new MyThread(criticalSection, 1)).start();  
    }  
}
```

- 2.1 Introduction**
- 2.2 Condition Synchronization**
- 2.3 Mutual Exclusion**
  - 2.3.1 First Attempt**
  - 2.3.2 Second Attempt**
  - 2.3.3 Third Attempt**
  - 2.3.4 Fourth Attempt**
  - 2.3.5 Dekker's Algorithm**
  - 2.3.6 Several Critical Sections**
- 2.4 Busy wait VS. passive wait**
- 2.5 Conclusions**

## 2.3 Mutual Exclusion

### 2<sup>nd</sup> attempt (Pascal-FC)

- In order to solve the problem of having just one variable which leads to mandatory alternation: when the flag of process B is not set, then process A uses its own flag to indicate it is entering in CS

→ I go into CS because you're not in it.

```
program SecondAttempt;
var flag1,flag2:Boolean;

process P1;
begin
  repeat
    while flag2 do;
    flag1:=true; the processor can be lost in the middle of execution
    writeln('P1 is in CS');
    writeln('P1 is leaving CS');
    flag1:=false;
    writeln('P1 is in non-CS');
  forever
end;

process P2;
begin
  repeat
    while flag1 do;
    flag2:=true;
    writeln('P2 is in CS');
    writeln('P2 is leaving CS');
    flag2:=false;
    writeln('P2 is in non-CS');
  forever
end;
```

But the 2<sup>nd</sup> attempt is not correct because

- Mutual Exclusion is not guaranteed.

P2 is in CS

P1 is in CS

- Moreover, starvation may occur if one process repeats its loop and sets its flag before the other process leaves its busy-wait.

Think what interleaving of statements leads to common access to CS

→ when losing the processor after setting the flag to true

# 2.3 Mutual Exclusion

## 2<sup>nd</sup> attempt

Interleaving which leads to common access to CS:

|   | P1                                   | P2                                   | flag1              | flag2              |
|---|--------------------------------------|--------------------------------------|--------------------|--------------------|
| 1 | <code>while flag2 do</code>          |                                      | <code>false</code> | <code>false</code> |
| 2 |                                      | <code>while flag1 do</code>          | <code>false</code> | <code>false</code> |
| 3 | <code>flag1:=true</code>             |                                      | <code>true</code>  | <code>false</code> |
| 4 |                                      | <code>flag2:=true;</code>            | <code>true</code>  | <code>true</code>  |
| 5 |                                      | <code>writeln('P2 is in CS');</code> | <code>true</code>  | <code>true</code>  |
| 6 | <code>writeln('P1 is in CS');</code> |                                      | <code>true</code>  | <code>true</code>  |

# 2.3 Mutual Exclusion

## 2<sup>nd</sup> attempt

Starvation does not happen on the fail of one process in its non-CS because its flag would remain false.

|   | P1                         | P2   | flag1        | flag2        |
|---|----------------------------|--|--------------|--------------|
| 1 | ... flag1:= <b>false</b> ; | → no one changes process 1 flag                  |              |              |
| 2 | <b>P1 halts forever!</b>   |  | <b>false</b> | <b>false</b> |
| 3 |                            | <b>while</b> flag1 <b>do</b>                     | <b>false</b> | <b>false</b> |
| 4 |                            | flag2:= <b>true</b> ;<br>writeln('P2 is in CS'); | <b>false</b> | <b>true</b>  |
|   |                            | writeln('P2 is leaving CS');                     | <b>false</b> | <b>true</b>  |
|   |                            | flag2:= <b>false</b> ;                           | <b>false</b> | <b>false</b> |
|   |                            | writeln('P2 is in non-CS');                      | <b>false</b> | <b>false</b> |
|   |                            | <b>while</b> flag1 <b>do</b>                     | <b>false</b> | <b>false</b> |
|   |                            | flag2:= <b>true</b> ;                            | <b>false</b> | <b>true</b>  |
|   |                            | writeln('P2 is in CS');...                       | <b>false</b> | <b>true</b>  |
|   |                            | <i>Keeps entering in CS forever</i>              |              |              |

## 2.3 Mutual Exclusion 2<sup>nd</sup> attempt (Java)

```
class CS2 {  
  
    boolean[] flag={false, false};  
  
    public void enterCS(int ID) {  
        // pre-protocol  
        while (flag[1-ID]);  
        flag[ID]=true;  
        // CS  
        System.out.println("P" + ID + " is in CS");  
        System.out.println("P" + ID + " is leaving CS");  
        // non-CS  
        // post-protocol  
        flag[ID]=false;  
  
        System.out.println("P" + ID + " is in non-CS");  
    }  
}
```

```
class MyThread2 extends Thread{  
    CS2 sharedCS;  
    int ID;  
  
    MyThread2(CS2 cs, int id){  
        sharedCS=cs;  
        ID=id;  
    }  
  
    public void run(){  
        while(true){  
            sharedCS.enterCS(ID);  
        }  
    }  
}
```

```
public class SecondAttempt{  
  
    public static void main(String args[]){  
        CS2 criticalSection=new CS2();  
  
        (new MyThread2(criticalSection, 0)).start();  
        (new MyThread2(criticalSection, 1)).start();  
  
    }  
}
```

- 2.1 Introduction**
- 2.2 Condition Synchronization**
- 2.3 Mutual Exclusion**
  - 2.3.1 First Attempt**
  - 2.3.2 Second Attempt**
  - 2.3.3 Third Attempt**
  - 2.3.4 Fourth Attempt**
  - 2.3.5 Dekker's Algorithm**
  - 2.3.6 Several Critical Sections**
- 2.4 Busy wait VS. passive wait**
- 2.5 Conclusions**

## 2.3 Mutual Exclusion

### 3<sup>rd</sup> attempt

I want to enter into the CS but I w8 if you want to go into the CS too.

- 1<sup>st</sup> attempt → because of the use of 1 variable, it forces alternation, and one process starves when the other halts.
- 2<sup>nd</sup> attempt → mutual exclusion is not granted because one process may check the status of the other process before it is updated.
- So now, we still use **2 variables** but they do not indicate the status of being inside the CS or not, but the will to enter **before trying to enter**.

## 2.3 Mutual Exclusion

### 3<sup>rd</sup> attempt (Pascal-FC)

```
program ThirdAttempt;

var wantsCS1,wantsCS2:Boolean;

process P1;
begin
  repeat
    wantsCS1:=true;
    while wantsCS2 do;
    writeln('P1 is in CS');
    writeln('P1 is leaving CS');
    wantsCS1:=false;
    writeln('P1 is in non-CS');
  forever
end;

process P2;
begin
  repeat
    wantsCS2:=true;
    while wantsCS1 do;
    writeln('P2 is in CS');
    writeln('P2 is leaving CS');
    wantsCS2:=false;
    writeln('P2 is in non-CS');
  forever
end;
```

```
begin
  wantsCS1:=false;
  wantsCS2:=false;
  cobegin
    P1;
    P2;
  coend
end.
```

I do not enter in CS if the other is willing to enter

*Besides global variables, Pascal-FC allows sharing information by passing **variables per reference** (next slide)*

## 2.3 Mutual Exclusion

### 3<sup>rd</sup> attempt (Pascal-FC)

- Same solution but passing per reference a record which holds the 2 variables

```
program ThirdAttemptPerRef;

type willsRecord = record
    wantsCS1,wantsCS2: boolean;
end;

process P1(var w: willsRecord);
begin
repeat
    w.wantsCS1:=true;
    while w.wantsCS2 do;
    writeln('P1 is in CS');
    writeln('P1 is leaving CS');
    w.wantsCS1:=false;
    writeln('P1 is in non-CS');
forever
end;

process P2(var w: willsRecord);
begin
repeat
    w.wantsCS2:=true;
    while w.wantsCS1 do;
    writeln('P2 is in CS');
    writeln('P2 is leaving CS');
    w.wantsCS2:=false;
    writeln('P2 is in non-CS');
forever
end;
```

```
var wills:willsRecord;
begin
wills.wantsCS1:=false;
wills.wantsCS2:=false;
cobegin
    P1(wills);
    P2(wills);
coend
end.
```

The 3<sup>rd</sup> attempt may fall in a **livelock**.  
Can you guess the interleaving which leads to that situation?

## 2.3 Mutual Exclusion

### 3<sup>rd</sup> attempt

|   | P1                                  | P2                                  | wantsCS1    | wantsCS2     |
|---|-------------------------------------|-------------------------------------|-------------|--------------|
| 1 | w.wantsCS1:= <b>true</b> ;          |                                     | <b>true</b> | <b>false</b> |
| 2 |                                     | w.wantsCS2:= <b>true</b> ;          | <b>true</b> | <b>true</b>  |
| 3 |                                     | <b>while</b> w.wantsCS1 <b>do</b> ; | <b>true</b> | <b>true</b>  |
| 4 | <b>while</b> w.wantsCS2 <b>do</b> ; |                                     | <b>true</b> | <b>true</b>  |
|   | <i>livelock</i>                     |                                     | <b>true</b> | <b>true</b>  |

## 2.3 Mutual Exclusion

### 3<sup>rd</sup> attempt (Java)

```
class CS3 {  
  
    boolean[] wantsCS={false, false};  
  
    public void enterCS(int ID) {  
        // pre-protocol  
        wantsCS[ID]=true;  
        while (wantsCS[1-ID]);  
        // CS  
        System.out.println("P" + ID + " is in CS");  
        System.out.println("P" + ID + " is leaving CS");  
        // non-CS  
        // post-protocol  
        wantsCS[ID]=false;  
  
        System.out.println("P" + ID + " is in non-CS");  
    }  
}
```

- 2.1 Introduction**
- 2.2 Condition Synchronization**
- 2.3 Mutual Exclusion**
  - 2.3.1 First Attempt**
  - 2.3.2 Second Attempt**
  - 2.3.3 Third Attempt**
  - 2.3.4 Fourth Attempt**
  - 2.3.5 Dekker's Algorithm**
  - 2.3.6 Several Critical Sections**
- 2.4 Busy wait VS. passive wait**
- 2.5 Conclusions**

## 2.3 Mutual Exclusion

### 4<sup>th</sup> attempt

- 1<sup>st</sup> attempt → because of the use of 1 variable, it forces alternation, and one process starves if the other halts.
- 2<sup>nd</sup> attempt → mutual exclusion is not granted because one process may check the status of the other process before it is updated.
- 3<sup>rd</sup> attempt → if both processes want to enter (**contention for access**), none of them renounces its will. So both will wait forever.
- The fourth attempt resolves contentions by making a process renounce, during a short period of time, its will to enter if the other process wants to enter.

## 2.3 Mutual Exclusion 4<sup>th</sup> attempt (Pascal-FC)

```
program FourthAttempt;

type willsRecord = record
    wantsCS1,wantsCS2: boolean;
  end;
process P1(var w: willsRecord);
begin
  repeat
    w.wantsCS1:=true;
    while w.wantsCS2 do
      begin
        w.wantsCS1:=false;
        (*do anything, e.g. sleep*)
        w.wantsCS1:=true;
      end;
    writeln('P1 is in CS');
    writeln('P1 is leaving CS');
    w.wantsCS1:=false;
    writeln('P1 is in non-CS');
    forever
  end;

process P2(var w: willsRecord);
begin
  repeat
    w.wantsCS2:=true;
    while w.wantsCS1 do
      begin
        w.wantsCS2:=false;
        (*do anything, e.g. sleep*)
        w.wantsCS2:=true;
      end;
    writeln('P2 is in CS');
    writeln('P2 is leaving CS');
    w.wantsCS2:=false;
    writeln('P2 is in non-CS');
    forever
  end;
```

```
var wills:willsRecord;
begin
  wills.wantsCS1:=false;
  wills.wantsCS2:=false;
  cobegin
    P1(wills);
    P2(wills);
  coend
end.
```

Both processes may give way to each other during a long period. Livelock and starvation will not last forever, because a process will eventually gain access to CS.  
This solution is **correct but lacks efficiency**.

## 2.3 Mutual Exclusion

### 4<sup>th</sup> attempt (Java)

```
class CS4 {  
  
    boolean[] wantsCS={false,false};  
  
    public void enterCS(int ID) {  
        // pre-protocol  
        wantsCS[ID]=true;  
        while (wantsCS[1-ID]){  
            wantsCS[ID]=false;  
            try {  
                Thread.sleep(100);  
            } catch (InterruptedException e) {  
                e.printStackTrace();  
            }  
            wantsCS[ID]=true;  
        }  
        // CS  
        System.out.println("P" + ID + " is in CS");  
        System.out.println("P" + ID + " is leaving CS");  
        // non-CS  
        // post-protocol  
        wantsCS[ID]=false;  
  
        System.out.println("P" + ID + " is in non-CS");  
    }  
}
```

- 2.1 Introduction**
- 2.2 Condition Synchronization**
- 2.3 Mutual Exclusion**
  - 2.3.1 First Attempt**
  - 2.3.2 Second Attempt**
  - 2.3.3 Third Attempt**
  - 2.3.4 Fourth Attempt**
  - 2.3.5 Dekker's Algorithm**
  - 2.3.6 Several Critical Sections**
- 2.4 Busy wait VS. passive wait**
- 2.5 Conclusions**

## 2.3 Mutual Exclusion Dekker's Algorithm

- Dekker decided to join the first, third and fourth attempts:
  - each process has a *flag* to announce its will to enter in CS.
  - when there is a contention, a common variable *turn* decides which must give way to the other.
- So Dekker's solution uses 3 shared variables: two boolean flags (one per process) and an integer for *turn*.
- It fulfills all requirements to be a correct solution using protocols:
  - Mutual exclusion is assured
  - Livelock does not happen
  - Starvation does not happen
- And it is efficient! (except for the use of busy-wait)

# 2.3 Mutual Exclusion

## Dekker's Algorithm (Pascal-FC)

```
program Dekker;

type willsRecord = record
    wantsCS1,wantsCS2: boolean;
    turn:integer;
  end;

process P1(var w: willsRecord);
begin
repeat
    w.wantsCS1:=true;
    while w.wantsCS2 do
        if w.turn = 2 then
        begin
            w.wantsCS1:=false;
            while w.turn = 2 do;
            w.wantsCS1:=true;
        end;
    writeln('P1 is in CS');
    writeln('P1 is leaving CS');
    w.turn:=2;
    w.wantsCS1:=false;
    writeln('P1 is in non-CS');
forever
end;
```

```
process P2(var w: willsRecord);
begin
repeat
    w.wantsCS2:=true;
    while w.wantsCS1 do
        if w.turn = 1 then
        begin
            w.wantsCS2:=false;
            while w.turn = 1 do;
            w.wantsCS2:=true;
        end;
    writeln('P2 is in CS');
    writeln('P2 is leaving CS');
    w.turn:=1;
    w.wantsCS2:=false;
    writeln('P2 is in non-CS');
forever
end;

var wills:willsRecord;
begin
wills.wantsCS1:=false;
wills.wantsCS2:=false;
wills.turn:=1;
cobegin
    P1(wills);
    P2(wills);
coend
end.
```

## 2.3 Mutual Exclusion Dekker's Algorithm (Pascal-FC)

*Rewrite Dekker's algorithm using global variables instead of a record passed by reference.*

## 2.3 Mutual Exclusion Dekker's Algorithm (Java)

```
class CSdekker {  
  
    boolean[] wantsCS={false, false};  
    volatile int turn=0;  
  
    public void enterCS(int ID) {  
        // pre-protocol  
        wantsCS[ID]=true;  
        //if the other process does not  
        //want to enter, I enter even  
        //if it is not my turn  
        while (wantsCS[1-ID]) {  
            if(turn==1-ID){  
                wantsCS[ID]=false;  
                while(turn==1-ID);  
                wantsCS[ID]=true;  
            }  
        }  
        // CS  
        System.out.println("P" + ID + " is in CS");  
        System.out.println("P" + ID + " is leaving CS");  
        // non-CS  
        // post-protocol  
        turn=1-ID;  
        wantsCS[ID]=false;  
  
        System.out.println("P" + ID + " is in non-CS");  
    }  
}
```

Do you think the *volatile* modifier  
Is necessary in variable *turn*?

## 2.3 Mutual Exclusion

- Other algorithms :
  - Peterson's (1981) developed an easier pre-protocol to grant exclusive access to CS.
  - Dijkstra's (1965) is an extension of Dekker's algorithm for  $n$  processes.
  - Eisenber-McGuire's (1972) improved the efficiency of Dijkstra's algorithm.
  - Lamport's algorithm, also known as the Bakery algorithm (1974), was developed for  $n$  processes running in distributed systems, where there is only read-access for shared variables which belong to other processes
- Hardware solutions: there exist processors which provide special atomic instructions, which grant mutual exclusion avoiding the use of protocols.

Increment & Decrement instructions (also Fetch-and-Add)

```
INC(int x) { int v = x; x = x + 1; return v }
```

*Machine instruction from processor IA32*

- 2.1 Introduction**
- 2.2 Condition Synchronization**
- 2.3 Mutual Exclusion**
  - 2.3.1 First Attempt**
  - 2.3.2 Second Attempt**
  - 2.3.3 Third Attempt**
  - 2.3.4 Fourth Attempt**
  - 2.3.5 Dekker's Algorithm**
  - 2.3.6 Several Critical Sections**
- 2.4 Busy wait VS. passive wait**
- 2.5 Conclusions**

## 2.3 Mutual Exclusion Several CSs

- There exist two kinds of **atomic execution**:
  - **Fine-grained**
    - Provided by the programming language to the developer.
    - They are compiled to atomic machine instructions executed by the processor.
  - **Coarse-grained**
    - Set of instructions executed without interleaving of other processes.
    - There exist programming (protocols, semaphores,...) and hardware solutions which provide tools to make a set of sentences be executed in an atomic manner.
- Given these definition, we can say that the **Critical Section** instructions together are a **coarse-grained instruction** because two processes cannot interleave critical section statements (they can mix one CS statements with non-CS from other process).

## 2.3 Mutual Exclusion Several CSs

- The work done inside a critical section usually performs changes in shared variables. If this change is done only in one piece of code, then the program only has 1 CS.
- But the same **shared variable** may be accessed/changed **in several situations** inside the same program:
  - Each piece of code which makes access counts as 1 CS.
  - **Flags and variables used in protocols are not replicated, they are used in access to all CSs.**
- E.g.: increments and decrements of the same variable

## 2.3 Mutual Exclusion Several CSs

- Variable  $x$  is used in 2 critical sections. Protocols may be any which is correct, e.g. Dekker's algorithm.

```
program incdec;

process type inc(var x:integer);
begin
  (*preprotocol(x)*)
  x:=x+1;
  (*postprotocol(x)*)
end;

process type dec(var x:integer);
begin
  (*preprotocol(x)*)
  x:=x-1;
  (*postprotocol(x)*)
end;
```

```
var
  x:integer;
  pInc:inc; pDec:dec;
begin
  x:=0;
  cobegin
    pInc(x);
    pDec(x);
  coend;
  writeln(x)
end.
```

- 2.1 Introduction**
- 2.2 Condition Synchronization**
- 2.3 Mutual Exclusion**
  - 2.3.1 First Attempt**
  - 2.3.2 Second Attempt**
  - 2.3.3 Third Attempt**
  - 2.3.4 Fourth Attempt**
  - 2.3.5 Dekker's Algorithm**
  - 2.3.6 Several Critical Sections**
- 2.4 Busy wait VS. passive wait**
- 2.5 Conclusions**

## 2.4 Busy wait VS. passive wait

- As we saw, busy-wait is the execution of instructions used to make the process wait for a condition to be fulfilled in order to go on doing actual progress in the program.

**while not continue do;**
- So the process uses processor time by interleaving statements which “do nothing”.
- Thus, busy-wait is known to be a very inefficient way to make a process wait.
- **Problems of busy-wait:**
  - Processes doing busy-wait are wasting processor time which could be used by other processes willing to do useful work.
  - A processor working consumes energy and generates heat.
- It is necessary to find another approach to make processes wait

## 2.4 Busy wait VS. passive wait

- A process which uses **passive-wait** (also called **blocked-wait**) enters in state *blocked* (or similar). In that state, the process does not execute any instruction.
- A process exits the *blocked* state due to an action of another process (probably change of one condition variable).
- Clearly, passive-wait is more efficient than busy-wait.
- **Busy-wait**, as a means to achieve synchronization, is **recommended only** when the programming environment does not provide passive-wait tools or primitives.

## 2.4 Busy wait VS. passive wait

- Primitive calls, methods or objects provided by some languages allow a **greater abstraction for synchronization** than using busy-wait-based protocols.
- Among these programming tools, some of them may still make our code confusing and error-prone.
- Not all synchronization tools are available in all programming languages.
- **Synchronization tools** are available for two models of communication:
  - Shared-memory
    - Semaphores
    - Critical Regions
    - Conditional Critical Regions (CCR)
    - Monitors
  - Message Passing
    - (A)synchronous message passing
    - Remote invocation

- 2.1 Introduction**
- 2.2 Condition Synchronization**
- 2.3 Mutual Exclusion**
  - 2.3.1 First Attempt**
  - 2.3.2 Second Attempt**
  - 2.3.3 Third Attempt**
  - 2.3.4 Fourth Attempt**
  - 2.3.5 Dekker's Algorithm**
  - 2.3.6 Several Critical Sections**
- 2.4 Busy wait VS. passive wait**
- 2.5 Conclusions**

## 2.5 Conclusions

- Processes may need to be synchronized in order to:
  - start/end an action (Condition Synchronization)
  - access a shared resource (mutual exclusion)
- In any programming environment, **mutual exclusion** access to a critical section can be achieved using **protocols** which make use of shared variables and busy-wait.
- In order to **say these protocols are correct**, they must:
  - Grant mutual exclusion
  - Avoid livelock
  - Avoid starvation
  - Avoid deadlock.

## 2.5 Conclusions

- **Passive-wait** primitives are available in **some concurrent programming languages** to achieve a more efficient execution, and to make our code easier to read and less error-prone.
- **You are lucky**, we will study these tools in the following units!

# Potential Midterm Exam Questions

1. What interactions between processes need synchronization?

Read and write

2. What do we mean when we say that a process running a busy-wait is *doing nothing*?

It means it is waiting for a condition to be fulfilled

3. In program *ConditSynch* (section 2.2), what interleaving of instructions leads to output “aS1 as2 bS1 bS2”?

|   | Process A        | Process B             | continue |
|---|------------------|-----------------------|----------|
| 1 | write('aS1')     |                       | false    |
| 2 | continue := true |                       | true     |
| 3 | write('as2')     |                       | true     |
| 4 |                  | write('bS1')          | true     |
| 5 |                  | while not continue do | true     |
| 6 |                  | write('bS2')          | true     |

# Potential Midterm Exam Questions

4. What are the requirements for a correct solution of mutual exclusion synchronization?

The implementation of a pre-protocol (to indicate a process is entering the CS) and a post-protocol (to announce the process is leaving the CS) allowing just one single process to enter the CS at a time.

5. What problems can you find in this attempt of achieving mutual exclusion?

Livelock  $\rightarrow$  starvation

They'll be in the loop forever  $\rightarrow$

?

```
program FirstAttempt;
var turn:Integer;

process P1;
begin
  repeat
    while turn <> 1 do;
      writeln('P1 is in CS');
      writeln('P1 is leaving CS');
      turn:=1; 1
      writeln('P1 is in non-CS');
  forever
end;

process P2;
begin
  repeat
    while turn <> 2 do;
      writeln('P2 is in CS');
      writeln('P2 is leaving CS');
      turn:=2; 2
      writeln('P2 is in non-CS');
  forever
end;
```

# Potential Midterm Exam Questions

6. From the point of view of atomic execution, what can we say about critical sections?

*They're coarse-grained instruction because two processes cannot interleave critical section statements.*

7. What is the upper bound of the number of critical sections in a program?



***Why did the concurrent chicken cross the road?***

*the side! other To to get*



# Keywords phonetics

- attempt /ə'tempt/ 
- renounce /rɪ'naʊns/ 
- mutual /'mju:tʃuəl/ 
- exclusion /ɪk'sklu:ʒən/ 
- coarse /kɔ:s/ 
- critical /'krɪtɪkəl/ 
- section /'sekʃən/ 
- contention /kən'tenʃən/ 

# Unit 3

## Shared-memory Communication

### 3.1 SEMAPHORES

harto  
aqui 1<sup>er</sup>  
examen }

- 3.1 Semaphores
  - 3.1.1 Introduction
  - 3.1.2 Mutual Exclusion
  - 3.1.3 Condition Synchronization
  - 3.1.4 Producer-Consumer Problem
  - 3.1.5 The Dining Philosophers

- 3.2 CCR
- 3.3 Monitors

### **3.1 Semaphores**

**3.1.1 Introduction**

**3.1.2 Mutual Exclusion**

**3.1.3 Condition Synchronization**

**3.1.4 Producer-Consumer Problem**

**3.1.5 The Dining Philosophers**

**3.2 CCR**

**3.3 Monitors**

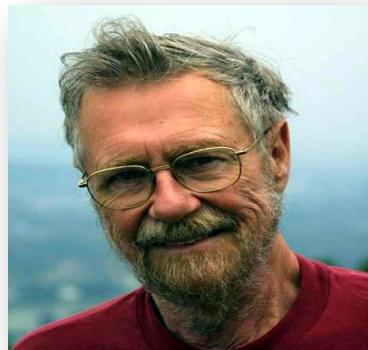
# 3 Shared-memory Communication

- In Unit 2, we learned techniques based on busy-wait which allowed the programmer to synchronize processes so that:
  - Processes do not execute some action if a condition is not met
  - Processes access shared resources under mutual exclusion
- We learned that passive-wait (blocked process) is much more efficient than using busy-wait and protocols.
- In multiprogramming, processes share memory. This fact has been used to develop tools which make synchronization easier and less error-prone.
- The most important and most frequently found in concurrency languages, are:
  - Semaphores
  - Conditional Critical Regions
  - Monitors

ordered increasingly by level of abstraction.

## 3.1.1 Introduction to semaphores

- Dijkstra (1968) creates the first synchronization primitive tool using passive-waiting: semaphores.
- A **semaphore** is a low level tool which helps the programmer achieve conditional synchronization among processes and access to critical sections under mutual exclusion.



Dijkstra  
(1930-2002)

## 3.1.1 Introduction to semaphores

- A semaphore may be implemented as a structured data type or as an object, depending on the programming language.

- In order to **implement a semaphore from scratch**, we need:

**counter** – A private counter of permits. —> how many processes can access the critical section

**queue** – A private set of *blocked processes*

**methods** **Public methods** which change the counter and, according to the new value, may make changes in the state of processes (block / unblock).

- **Permissible values** for the counter of permits depends of the **type of semaphore**:

- General or counting semaphore: non-negative integer value.
- Binary semaphore: 0 or 1. Only 1 bit of storage is required!

(only one process can enter the critical section)

- In practice, we refer to the current value of the counter of permits by saying “the value of the semaphore”. So, given semaphore  $s$ , “ $s > 0$ ” stands for “the counter of permits in  $s$  is greater than 0”. —> if the value of the counter is different from 0 the process can continue.

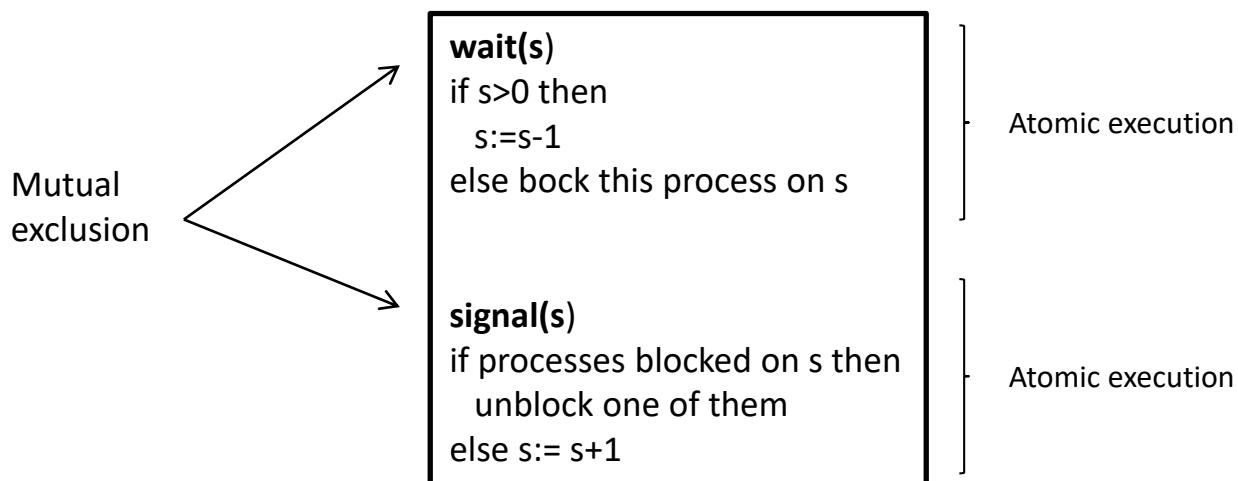
## 3.1.1 Introduction to semaphores

( toilet w/ key example )

- **Permissible operations.** There are only 2 operations which processes can carry on semaphores: *wait(s)* and *signal(s)*.  
[ Binary semaphore  $\rightarrow$  **MUTEX** ]  
 $\hookrightarrow$  access to a critical section
- **wait(s):** decreases  $s$  or blocks the process.
  - If  $s > 0$ ,  $s := s - 1$
  - If  $s = 0$ , block the process and queue it in blocked processes set.
- **signal(s):** increases  $s$  or unblocks a process.  $\rightarrow$  atomic action
  - If there are blocked processes in  $s$ , unblock one process.
  - else,  $s := s + 1$
- A call to any of these operations involves testing the value of  $s$  or the set of *blocked processes* and, when appropriate, perform a modification. Semaphores grant that all this is executed as **an indivisible action**. Internally, **this is not implemented with busy-wait** but using OS functions to block and grant mutual exclusion of both operations.

## 3.1.1 Introduction to semaphores

- Pascal uses a random algorithm } to unblock a process
- C uses FIFO
- Having 2 operations called on a semaphore, op1 and op2, both are to be executed but we cannot predict the order. We just know that they will be run under mutual exclusion.



- In the signal operation, if there is more than 1 blocked process, the way to **select the process to be unblocked** depends on the implementation of the semaphore: FIFO , priorities, random (Pascal-FC),...

## 3.1.1 Introduction to semaphores

- If **signal** is called on a **binary semaphore** which already has a **value of 1**, in most programming languages this is tackled by becoming this call a no-operation; that is, it has no effect.
- **Caution**, the name of operation *wait* maybe a bit misleading. A process calling *wait* only does wait (blocks) if  $s=0$ . That is, if it cannot take any permit.
- In Java:
  - keywords *wait* and *signal*, without parameters, are reserved for processes signaling. They can be regarded as a semaphore with 0 permits by default in any execution. We will work with them in lab assignments.
  - Object *java.util.concurrent.Semaphore* provides the corresponding methods *acquire* and *release*.  
*wait()*  
*signal()*

# 3.1.1 Introduction to semaphores

- Pascal-FC provides non-negative general semaphores. So if you want a binary semaphore, it is your responsibility to correctly deploy wait and signal operations.
- Examples for declaring variables, an array and a record using type **semaphore**.

```
var
  s1,s2:semaphore;
  semArray:array [1..5] of semaphore;
  semRec:record
    i:integer;
    s:semaphore;
  end;
```

- Restrictions in declaring semaphores:
  - they may be declared
    - in the global declaration space right before the main block and pass them by reference .
    - In the global declaration space of the program.
  - In order to pass them by reference, do not forget to use the var modifier.

## 3.1.1 Introduction to semaphores

- Why do you think semaphores need to be passed as reference?  
Each process must access the semaphore itself not just a copy.
- Procedure **initial(s, value)** initiates semaphore s to the given value. MUTEX(1), o.w. (#processes that can access the critical section)
- Semaphores must be **initiated in the main block** and out of the **cobeing-coend keywords**.  
↳ semaphore must be initialized outside of cobeing-coend
- Pascal provides the 2 semaphore operations, implemented as procedures:
  - **wait(s)** and **signal(s)**,
  - Where s is of type **semaphore**.
  - **signal(s)** is never a null operation
- Procedure **write** can use a semaphore as argument to output its value on screen.
- **Java** provides class `java.util.concurrent.Semaphore`.  
`Semaphore mutex=new Semaphore(1); // binary semaphore`

### **3.1 Semaphores**

**3.1.1 Introduction**

**3.1.2 Mutual Exclusion**

**3.1.3 Condition Synchronization**

**3.1.4 Producer-Consumer Problem**

**3.1.5 The Dining Philosophers**

**3.2 CCR**

**3.3 Monitors**

## 3.1.2 Mutual Exclusion

- By using semaphores, we can **forget about using protocols** to access and exit critical sections.
  - Given a critical section CS, we achieve mutual exclusion access by:
    - using a binary semaphore  $s$
    - calling  $wait(s)$  right before entering CS
    - calling  $signal(s)$  right after exiting CS
  - If  $s=1$ , CS is free.
  - If  $s=0$ , a process is inside CS.
- If we have **several critical sections**, we need the same binary semaphore for each subset of critical sections which need mutual exclusion among them.

## 3.1.2 Mutual Exclusion (Pascal-FC)

```
program mutexSem;
var
  cont : integer;

process type MyProcess(var sem: semaphore; id:integer);
begin
  repeat
    wait(sem);
    (* CS *)           ↗ value the previous process has written
    cont := cont +1;   ↗ numbers in the output are in order
    writeln('[' ,id, '] ',cont);

    signal(sem); —> finish
    (* non-CS*)

  forever
end;

var
  p1,p2:MyProcess;
  mutex:semaphore; —> outside (global)

begin
  initial(mutex,1); —> outside (global)
  cont := 0;
  cobegin
    p1(mutex,1);
    p2(mutex,2);
  coend;
end.
```

What do you think the output is?

id + count of each process

What may happen if we move the write command out of CS?

- number not in order
- a process can write a number more than once



## 3.1.2 Mutual Exclusion (Java)

```
class Counter {  
  
    int cont = 0;  
    Semaphore mutex = new Semaphore(1);  
  
    void increment(int id) {  
        try {  
            mutex.acquire();  
        } catch (InterruptedException e) {  
            e.printStackTrace();  
        }  
        cont++;  
        System.out.println("[" + id + "] " + cont);  
        mutex.release();  
    }  
}  
  
class counterThread extends Thread {  
    Counter c;  
    int ID;  
  
    counterThread(Counter counter, int id) {  
        c = counter;  
        ID = id;  
    }  
  
    public void run() {  
        for (int i = 0; i < 100; i++) {  
            c.increment(ID);  
        }  
    }  
}
```

```
public class mutexSem {  
    public static void main(String args[]) {  
        Counter counter = new Counter();  
        for (int i = 0; i < 5; i++)  
            (new counterThread(counter, i)).start();  
    }  
}
```

*Do you think the same reference to mutex is shared among processes?*

## 3.1.2 Mutual Exclusion

- **General semaphores** can be used for sharing a given number of available instances of a **resource**; or to provide **multiple exclusion** (more than 1 process in CS).
- E.g.: number of tickets for a concert; number of chairs in a room;...
- Thus:
  - Binary semaphore: mutual exclusion  
e.g. pascal-FC    initial (mutex, 1) ;
  - General semaphore:
    - Share or allocate resources.  
e.g. pascal-FC    initial (tickets, 5) ;
    - Multiple exclusion: N>1 processes in CS  
e.g. pascal-FC    initial (multiplex, 3) ;

## 3.1.2 Mutual Exclusion

- When a general semaphore is used to **share resources** (e.g. chair in waiting room), once one instance of a resource is returned the process must call *signal(s)*.
- If a general semaphore is used to **allocate resources** which are never returned (e.g. books for sale), then *signal(s)* is never called. If there are more processes than the initial value of the semaphore, some processes will never get the resource and fall in deadlock.



<http://adit.io/>

# HAMSTERS LIFE - GENERAL SEMAPHORE

- eat and play
- we have 5 hamsters
  - 1 plate : 3 hamsters at a time
  - 1 wheel : 1 hamster at a time

Process hamster {

```
while (true) {  
    wait (sem_eat);  
    eat();  
    signal (sem_eat);  
    wait (sem_play);  
    play();  
    signal (sem_play);  
}
```

init (sem\_play, 1)  
init (sem\_eat, 3)

1 hamster can access to the  
wheel

3 hamsters can access to  
the plate

↓ control access to the critical section  
and processes synchronization

7

7

### **3.1 Semaphores**

**3.1.1 Introduction**

**3.1.2 Mutual Exclusion**

**3.1.3 Condition Synchronization**

**3.1.4 Producer-Consumer Problem**

**3.1.5 The Dining Philosophers**

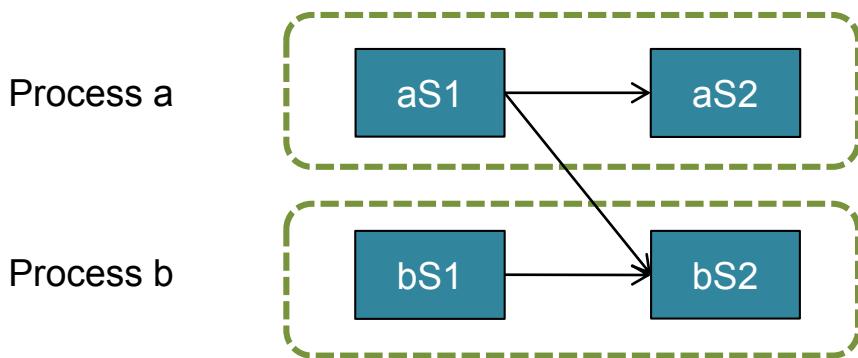
**3.2 CCR**

**3.3 Monitors**

### 3.1.3 Condition Synchronization

- Remember from Unit 2 that a condition synchronization happens when one process halts waiting for a condition to be met thanks to the action of another process.

A possible solution for this precedence diagram  
is the use of busy-wait:



```
program conditSynch;  
  
var  
    continue: boolean;  
  
process ProcessA;  
begin  
    write('aS1 ');  
    continue := true;  
    write('aS2 ');\nend;  
  
process ProcessB;  
begin  
    write('bS1 ');\n    while not continue do;  
        write('bS2 ');\n    end;  
  
begin  
    continue := false;  
    cobegin  
        ProcessA;  
        ProcessB;  
    coend  
end.
```

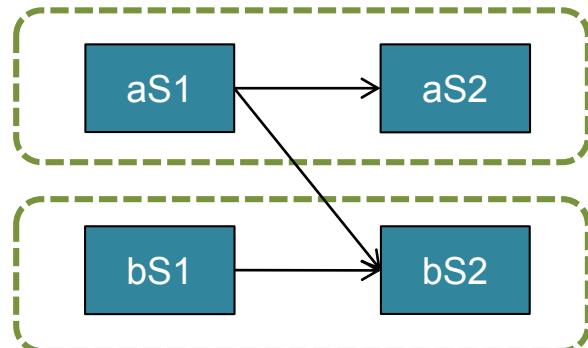
### 3.1.3 Condition Synchronization

*Solution using a binary semaphore in Pascal-FC*

```
program synchronization;  
  
var aS1done: semaphore;  
  
process ProcessA;  
begin  
    write('aS1 ');  
    signal(aS1done);  
    write('aS2 ');\br/>end;  
  
process ProcessB;  
begin  
    write('bS1 ');\br/>    wait(aS1done);  
    write('bS2 ');\br/>end;  
  
begin  
    initial(aS1done, 0);  
    cobegin  
        ProcessA;  
        ProcessB;  
    coend  
end.
```

use semaphores for synchronization

Process a



Process b

unblock process A  
waits for the other  
process to send the signal

Try to code this solution in Java at home!

### 3.1.3 Condition Synchronization

- A **rendezvous** is a meeting between 2 processes: both wait for each other

two processes  
must stop at a  
point and then  
they continue  
together

Process a

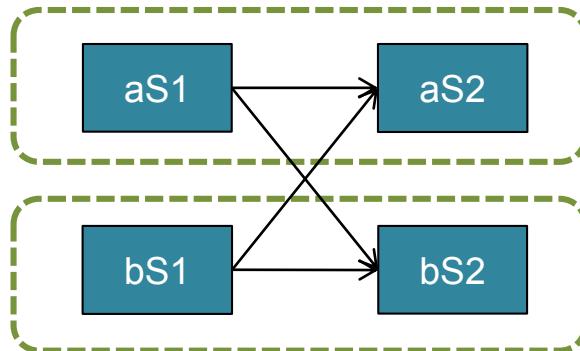
Process b

law free then a w8 4u

```
program rendezvous;

var aS1done: semaphore;
var bS1done: semaphore;

process ProcessA;
begin
    write('aS1 ');
    signal(aS1done);
    wait(bS1done);
    write('aS2 ');
end;
```



```
process ProcessB;
begin
    write('bS1 ');
    signal(bs1done);
    wait(aS1done);
    write('bS2 ');
end;

begin
    initial(aS1done,0);
    initial(bS1done,0);
    cobegin
        ProcessA;
        ProcessB;
    coend
end.
```

aS2 and bS2 will not  
be written until aS1  
and bS1 are.

To achieve a rendezvous,  
each process must *signal*  
its semaphore (announce  
itself) and then *wait* on  
the semaphore of the  
other process.

*Java code this solution  
at home!*

### 3.1.3 Condition Synchronization

→ turnstile

- A **barrier** → as a rendezvous but with more processes
  - is a type of condition synchronization in which processes are blocked in the same point until all of them reach that point.
  - Is a generalization of rendezvous, which only works for 2 processes.
- Pascal-FC does not have a primitive type for barriers.
- A common way to implement a barrier using semaphores is using the **turnstile solution**: once all processes are blocked in the same statement, pairs of wait-signal are executed to let all processes go through a binary semaphore.
- E.g.: 5 processes write 'A'. When all of them finish, then they write B.

*Draw the precedence diagram*

### 3.1.3 Condition Synchronization (Pascal-FC)

```
program turnstileBarrier;

const NPR=5;
var
  mutex : semaphore;
  Sbarrier: semaphore;
  pCounter: integer;

process type writer;
begin
  write('A');
  wait(mutex); (*mutex to increment counter*)
  pCounter := pCounter + 1;
  signal (mutex);
  if pCounter = NPR then signal(Sbarrier); -> unlock next process
    (*turnstile: once a process is unblocked from wait, the rest will be unblocked one after the other*)
  NO wait(Sbarrier);
  signal(Sbarrier); -> unlock next process
  write('B');
end;
```

→ control the access to the critical section when we have a global variable modified by various processes

```
var
  i:integer;
  writers:array [1..5] of writer;

begin
  pCounter := 0;
  initial(Sbarrier, 0); control that everybody arrives to this point
  initial(mutex, 1);
  cobegin
    for i:=1 to 5 do
      writers[i];
  coend
end.
```

### 3.1.3 Condition Synchronization (Java)

```
import java.util.concurrent.CyclicBarrier ;  
  
class Writer extends Thread{  
    CyclicBarrier barrier;  
  
    Writer(CyclicBarrier b){  
        barrier=b;  
    }  
    public void run(){  
        System.out.println("A");  
        try {  
            barrier.await();  
        } catch (Exception e) {  
            e.printStackTrace();  
        }  
        System.out.println("B");  
    }  
}  
  
public class Barrier {  
    static final int NPR=5;  
  
    public static void main(String args[]){  
        CyclicBarrier cb=new CyclicBarrier(NPR);  
        for(int i=0;i<NPR;i++) (new Writer(cb)).start();  
    }  
}
```

*The concurrent package of Java Provides an object CyclicBarrier which abstracts us from the barrier implementation*

### 3.1.3 Condition Synchronization (Java)

- A **cyclic barrier** allows to re-use the barrier again after the waiting threads are released.
- For example, it can be used in a loop to repeat the AAAAABBBBB printing 2 times.

```
public void run() {  
    for (int loop = 0; loop < 2; loop++) {  
        System.out.println("A");  
        try {  
            barrier.await();  
            System.out.println("B");  
            barrier.await();  
        } catch (Exception e) {  
            e.printStackTrace();  
        }  
    }  
}
```

*This code gives a new precedence restraint: the second set of 'A' cannot be printed until the previous set of 'B' are written. Otherwise, we would only get the first iteration right.*

Let's KAHOOT

(go to [kahoot.it](https://kahoot.it))

### 3.1.3 Condition Synchronization (Pascal-FC)

- Implementing a cyclic barrier in Pascal-FC is trickier because we need to build the barrier using semaphores.
- We need:
  - 2 semaphore-barriers
  - 2 turnstile protocols
  - Block the first semaphore-barrier at start.
  - Block the second semaphore-barrier before unblocking the first.
  - Decrease the counter before going through the second turnstile protocol

### 3.1.3 Condition Synchronization (Pascal-FC)

```
program cyclicBarrier;
const NPR=5;ITERATIONS=3;
var
  mutex,Sbarrier1,Sbarrier2 : semaphore;
  pCounter: integer;

process type rewriter;
  var it:integer;
begin
  for it:=1 to ITERATIONS do
    begin
      write('A');
      wait(mutex); (*mutex to increment counter*)
      pCounter := pCounter + 1;
      if pCounter = NPR then
        begin
          wait(Sbarrier2);{block Sbarrier2}
          signal(Sbarrier1);{unblock Sbarrier1}
        end;
      signal (mutex);

      wait(Sbarrier1);(*1st turnstile*)
      signal(Sbarrier1);

      write('B');
      wait(mutex); (*mutex to decrement counter*)
      pCounter := pCounter - 1;
      if pCounter = 0 then
        begin
          wait(Sbarrier1);{block Sbarrier1}
          signal(Sbarrier2);{unblock Sbarrier2}
        end;
      signal(mutex);

      wait(Sbarrier2); (*2nd turnstile*)
      signal(Sbarrier2);
    end;
end;
```

```
var
  i:integer;
  rewriters:array [1..5] of rewriter;
begin
  pCounter := 0;
  initial(Sbarrier1,0);
  initial(Sbarrier2,1);
  initial(mutex,1);
  cobegin
    for i:=1 to 5 do
      rewriters[i];
  coend
end.
```

*Why do you think mutex is used for both the increment and decrement?*

*Why do you think each mutex controls the access to six lines of code?*

## 3.1.3 Condition Synchronization

- Thus, semaphores can be used to provide:
  - Mutual exclusion in access to shared resources
  - Synchronize processes
- There exist several classic Concurrent Programming problems which need to be solved using both mutual exclusion and synchronization.
- We will solve 2 problems:
  - Producer-Consumer
  - The dining philosophers

## [Steps]

- 1) number and type of processes
- 2) Code in sequential way for each process
- 3) Look for synchronization parts:
  - Conditional exclusion
  - Mutual
- 4) Number of semaphores needed.
- 5) wait/signal
- 6) initial

①

program seqex;

Using semaphore

var P1.isDone P2.isDone semaphore

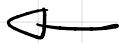
process P1

begin

write ('P1 is Done');

signal (P1.isDone);

end



process P2

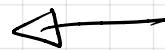
begin

wait (P1.isDone);

write ('P2 is done');

signal (P2.isDone);

end



process P3

begin

wait (P2.isDone)

write ('P3 is done');

end

begin

initial (P1.isDone, 0);

initial (P2.isDone, 0);

cobegin

P1

P2

P3

coend

end

P3 can go ahead  
when P2 has finished

(2)

program seges;

var P1.isDone P2.isDone P3.isDone semaphore

process P1

begin

wait (P2.isDone);

write ('P1 is done');

signal (P1.isDone);

end

process P3

begin

wait (P1.isDone);

write ('P3 is done');

signal (P3.isDone);

end

process P2

begin

wait (P3.isDone);

write ('P2 is done');

signal (P2.isDone);

end

begin

initial (P1.isDone, 0);

initial (P2.isDone, 1);

initial (P3.isDone, 0);

ojo

cobegin

P1;

P3;

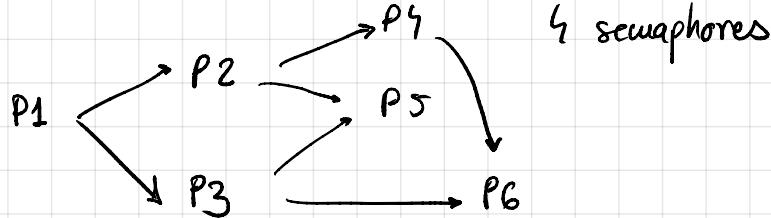
P2;

coend;

end;

(3)

WITHOUT WHILE



Var P1, P2, P3, P4: semaphore

process P1

begin

code P1;

signal(P2)

end

while loop | P1 P2 P3 ✓  
P1 P2 P2 -> starvation => P3 X

we do not know which process is going to take the signal

Signal(P2) → mutex

if P3 takes the signal the P2  
would never execute

process P2

begin

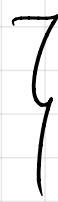
wait(P1)

code P2;

signal(P2);

end

Signal(P2);



initial (P1, 0);

initial (P2, 0);

initial (P3, 0);

initial (P4, 0);

process P3

begin

wait(P1)

code (P3);

signal(P3);

end.

Signal(P3);



process P4;

wait(P2)

code P4

signal(P4)

end

The solution is ok when the  
processes are not in while loop

process P5

begin

wait(P2)

wait(P3)

code P5

process P6

begin

wait(P3)

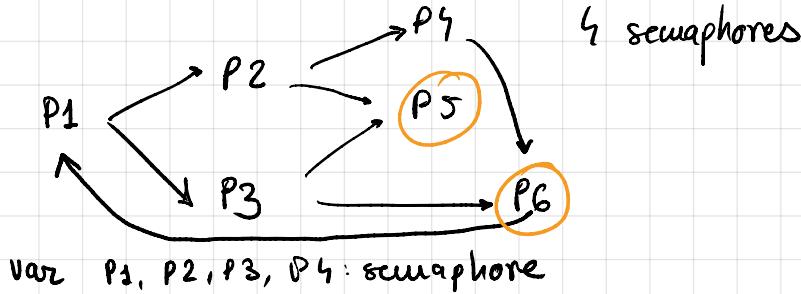
wait(P4)

code P6

end

4

WITH WHILE

P<sub>1</sub> before P<sub>2</sub> and P<sub>3</sub>P<sub>2</sub> before P<sub>4</sub> and P<sub>5</sub>P<sub>3</sub> before P<sub>5</sub>P<sub>6</sub> after P<sub>3</sub> and P<sub>4</sub>process P<sub>1</sub>

```

begin
repeat
  code P1
  signal(P2);
  forever
end
  
```

→ mutex

if P<sub>3</sub> takes the signal the P<sub>2</sub>  
would never execute

process P<sub>2</sub>

```

begin
repeat
  wait(P2)
  code P2;
  forever
end
  
```

}

process P<sub>3</sub>

```

begin
repeat
  wait(P3)
  code (P3);
  forever
end
  
```

}

```

process P4;
begin
repeat
  wait(P4)
  code P4
  signal(P6)
  forever
end
  
```

open  
semaphore

initial (P<sub>61</sub>, 1)initial (P<sub>6</sub>, 0);

```

process P5
begin
repeat
  wait(P5)
  wait(P3)
  code P5
  forever
end
  
```

process P<sub>6</sub>

```

begin
repeat
  wait(P6)
  wait(P4)
  code P6
  forever
end
  
```

3 semaphores

initial (P<sub>12</sub>, 0);initial (P<sub>13</sub>, 0);initial (P<sub>24</sub>, 0);initial (P<sub>25</sub>, 0);initial (P<sub>35</sub>, 0);initial (P<sub>36</sub>, 0);

4

 $x = 1$   
 $y = 4$ initial ( $S_1, 1$ )  
initial ( $S_2, 0$ )initial ( $S_3, 2$ )

|                  |
|------------------|
| PA               |
| wait ( $S_2$ )   |
| wait ( $S_3$ )   |
| $x = y * 2$      |
| $y = y + 2$      |
| signal ( $S_3$ ) |

|                  |
|------------------|
| PB               |
| wait ( $S_1$ )   |
| wait ( $S_3$ )   |
| $x = x + 1$      |
| $y = x + 8$      |
| signal ( $S_2$ ) |
| signal ( $S_3$ ) |

|                  |
|------------------|
| PC               |
| wait ( $S_1$ )   |
| wait ( $S_3$ )   |
| $x = y + 1$      |
| $y = x * 4$      |
| signal ( $S_3$ ) |
| signal ( $S_1$ ) |

PB has 1st the processor

$$\begin{cases} x = 20 \\ y = 11 \end{cases}$$

PC has 1st the processor

$$\begin{cases} x = 28 \\ y = 15 \end{cases}$$

|                            |
|----------------------------|
| PA                         |
| wait ( $S_2$ )             |
| wait ( $\underline{S_3}$ ) |
| $x = y * 2$ 20             |
| $y = y + 2$ 11             |
| signal ( $S_3$ )           |

|                  |
|------------------|
| PB               |
| wait ( $S_1$ ) 0 |
| wait ( $S_3$ ) 0 |
| $x = x + 1$ 2    |
| $y = x + 8$ 10   |
| signal ( $S_2$ ) |
| signal ( $S_3$ ) |

|                  |
|------------------|
| PC               |
| wait ( $S_1$ )   |
| wait ( $S_3$ )   |
| $x = y + 1$      |
| $y = x * 4$      |
| signal ( $S_3$ ) |
| signal ( $S_1$ ) |

starvation

|                  |
|------------------|
| PA               |
| wait ( $S_2$ )   |
| wait ( $S_3$ )   |
| $x = y * 2$ 28   |
| $y = y + 2$ 15   |
| signal ( $S_3$ ) |

|                  |
|------------------|
| PB               |
| wait ( $S_1$ )   |
| wait ( $S_3$ )   |
| $x = x + 1$ 6    |
| $y = x + 8$ 14   |
| signal ( $S_2$ ) |
| signal ( $S_3$ ) |

|                  |
|------------------|
| PC               |
| wait ( $S_1$ ) 0 |
| wait ( $S_3$ ) 0 |
| $x = y + 1$ 5    |
| $y = x * 4$ 20   |
| signal ( $S_3$ ) |
| signal ( $S_1$ ) |

### **3.1 Semaphores**

**3.1.1 Introduction**

**3.1.2 Mutual Exclusion**

**3.1.3 Condition Synchronization**

**3.1.4 Producer-Consumer Problem**

**3.1.5 The Dining Philosophers**

**3.2 CCR**

**3.3 Monitors**

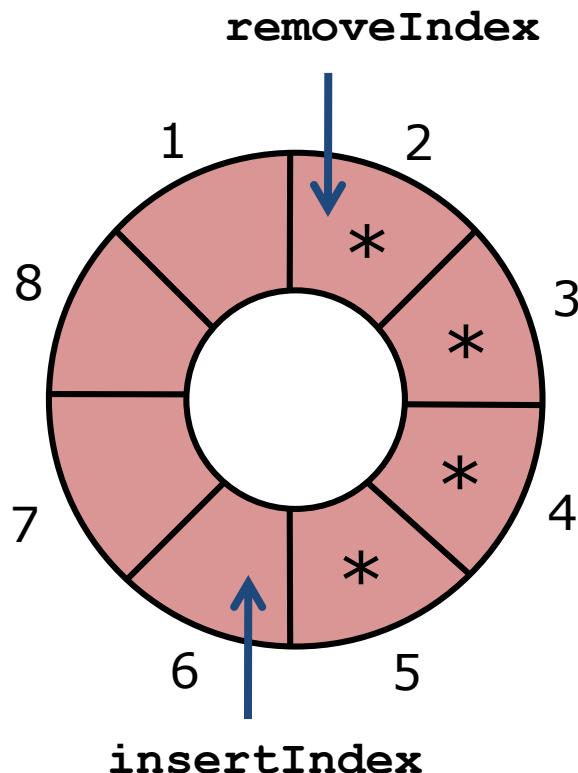
## 3.1.4 Producer-Consumer problem

- This problem can be set with different levels of difficulty. We will solve an advanced version of the problem, with 4 producers, 3 consumers and a buffer of Size 8.
- Producers insert data in a circular buffer.
  - Only 1 producer can insert an item in a given slot.
  - A position cannot be written if it is being read.
  - If the buffer is full, no item can be inserted.
- Consumers remove data from the circular buffer.
  - Remove in a FIFO manner
  - 2 consumers cannot remove an item from the same slot
  - A slot cannot be consumed if it is being written.
  - A consumer cannot remove data from an empty buffer.

Java does not have  
a FIFO queue in semaphore.  
In pseudo-code it works.

*Identify which restrictions need condition synchronization, and which need mutual exclusion.  
Do we need more than 1 semaphore for mutual exclusion?*

## 3.1.4 Producer-Consumer problem



Necessary data:

`removeIndex` → integer

`insertIndex` → integer

`buffer` → record.

`slots` → array of integer variables

Necessary semaphores:

Mutual exclusion → binary

Counter of empty slots → general

Counter of available items → general

*Why do we use semaphores to count,  
Instead of integer variables?*

# 3.1.4 Producer-Consumer problem

```

program producerConsumer;

const SIZE=8;
{new buffer type}
type tBuffer = record
  data: array [1..SIZE] of integer;
  insertIndex, removeIndex: integer;
  Sitems, Sempty, Smutex: semaphore;
end;

procedure init(var buffer:tBuffer);
begin
  buffer.insertIndex := 1;
  buffer.removeIndex := 1;
  initial(buffer.Sitems, 0); slots empty
  initial(buffer.Sempty, SIZE);
  initial(buffer.Smutex, 1);
end;

↓
when changing the value of the variables

```

What operations on buffer are protected by mutual exclusion?



```

procedure insert(item:integer; var buffer:tBuffer);
begin
  {block if buffer has no empty slots}
  wait(buffer.Sempty);
  wait(buffer.Smutex);
  buffer.data[buffer.insertIndex]:= item;
  writeln('-->',item);
  buffer.insertIndex:=buffer.insertIndex MOD SIZE + 1;
  signal(buffer.Smutex);
  {increase the counter of items}
  signal(buffer.Sitems);
end;

procedure remove(var item:integer; var
buffer:tBuffer);
begin
  {block if buffer has 0 items}
  wait(buffer.Sitems);
  wait(buffer.Smutex);
  item := buffer.data[buffer.removeIndex];
  writeln('<--',item);
  {our array starts by index 1}
  buffer.removeIndex:=buffer.removeIndex MOD SIZE + 1;
  signal(buffer.Smutex);
  {increase the counter of empty slots}
  signal(buffer.Sempty);
end;

```

procedure writer

```
begin
if (!flag)
  wait (Sempty);
  wait (wrt);
  *write*
  signal (wrt);
  signal (Sempty);
  flag = true;
end
wrt(1);
```

procedure reader

```
begin
flag = true;
wait (Sempty),
wait (wrt);
NO { * read *
  signal (wrt);
  signal (Sempty);
  flag = false;
end
```

volatile boolean flag = false;

[this solution gives priority to readers]

process type writer

```
begin
  wait (Sempty);
  wait (wrt);
  *write*
  signal (wrt);
  signal (Sempty);
end
```

| wrt(1);  
| nreaders = integer;

mutex ↑

process type reader

```
begin
  wait (Sitems);
  wait (writer);
  nreaders = nreaders + 1
  if (nreaders = 1) then
    signal (writer)
    wait (wrt) }
  → * read *
  signal (Sitems)
  wait (writer);
  nreaders = nreaders - 1;
```

if (nreaders = 0) then
 signal (writer);
 signal (wrt);

allow items to  
writer ↓

signal (Sitems);

end

### only 1 process access code

P1

```
while (true) {
    wait (mutex);
    code - P1;
    signal (mutex);
```

P2

```
while (true) {
    wait (mutex);
    code - P2;
    signal (mutex);
```

P3

```
while (true) {
    wait (mutex);
    code - P3;
    signal (mutex);
```

}

}

}

mutex (1);

### two processes access code

P1

```
while (true) {
    wait (sem);
    code - P1;
    signal (sem);
```

}

P2

```
while (true) {
    wait (sem);
    code - P2;
    signal (sem);
```

}

P3

```
while (true) {
    wait (sem);
    code - P3;
    signal (sem);
```

}

sem(2);

### order

P1

```
while (true) {
    wait (P3);
    code - P1;
    signal (P1);
```

}

P2

```
while (true) {
    wait (P1);
    code - P2;
    signal (P2);
```

}

P3

```
while (true) {
    wait (P2);
    code - P3;
    signal (P3);
```

}

P1 → P2 → P3

init(P1, 0); init(P3, 0);  
init(P2, 0);

P1

```
while (true) {
    wait (P1);
    code_P1;
    signal (P1);
}
```

?

P2

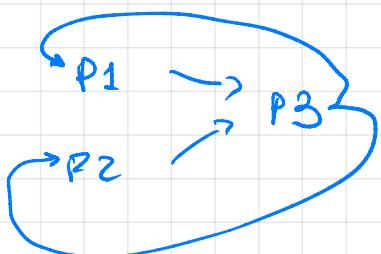
```
while (true) {
    wait (P2);
    code_P2;
    signal (P2);
}
```

?

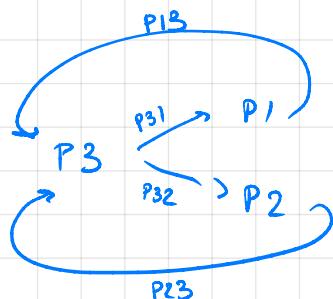
P3

```
while (true) {
    wait (P3);
    wait (P23);
    code_P3;
    signal (P3);
    signal (P23);
}
```

?



P13 (0);  
 P23 (0);  
 P31 (1);  
 P32 (2);



program }

P13, P12, P31, P32 : semaphore

initial (P13, 1); initial (P23, 0);  
 initial (P31, 0); initial (P32, 0);

P1

```
while (true) {
    wait (P31);
    code_P1;
    signal (P13);
}
```

P2

```
while (true) {
    wait (P32);
    code_P2;
    signal (P23);
}
```

?

P3

```
while (true) {
    wait (P13);
    wait (P23);
    code_P3;
    signal (P31);
    signal (P32);
}
```

?

# 3.1.4 Producer-Consumer problem

```

process type tProducer
    (var buffer:tBuffer);
var
  item: integer;
begin
  repeat
    item := random(200);
    insert(item,buffer);
  forever
end;

process type tConsumer
    (var buffer:tBuffer);
var
  item: integer;
begin
  repeat
    remove(item,buffer);
    writeln(item);
  forever
end;

```

```

var
  buffer:tBuffer;
  i:integer;
  prod:array [1..5] of tProducer;
  cons:array [1..3] of tConsumer;
begin
  init(buffer);
  cobegin
    for i:=1 to 5 do
      prod[i] (buffer);

    for i:=1 to 3 do
      cons[i] (buffer);
  coend;
end.

```

## 3.1.4 Producer-Consumer problem

- A similar solution can be implemented in Java using class Semaphore, and creating class Buffer instead of a record.
- In order to gain mutual exclusion, Java provides the keyword *synchronized*.
- Methods in the same class with **synchronized** modifier are executed under mutual exclusion. That is, only 1 thread can exist at a time among all synchronized methods.
- The Producer-Consumer problem is now solved using this keyword.
- In order to block and resume a thread, Java provides methods ***wait()*, *notify()*, *notifyAll()***. Do not confuse them with semaphores procedures. They are signals, which will be taught in more detail in Lab Sessions.
- We cannot know which thread is going to be resumed by *notify()*.

# 3.1.4 Producer-Consumer problem

```

class Buffer {
    static final int SIZE = 8;
    int insertIndex, removeIndex, empty, items;
    int[] data;

    Buffer() {
        data = new int[SIZE];
        insertIndex = 0;
        removeIndex = 0;
        empty = SIZE;
        items = 0;
    }

    synchronized void insert(int item)
        throws InterruptedException {

        // block if buffer has no empty slots
        while (empty == 0) {
            wait();
        }
        data[insertIndex] = item;
        //arrays start by index 0
        insertIndex = (1+insertIndex)%SIZE;

        items++; //increase counter of items
        notifyAll();
    }
}

```

```

synchronized int remove()
    throws InterruptedException {

    // block if buffer has 0 items
    while (items == 0) {
        wait();
    }
    int x = data[removeIndex];
    removeIndex = (1+removeIndex)%SIZE;

    empty++; //increase counter of empty slots

    notifyAll();
    return x;
}
}

```

*Why do we call notifyAll instead of notify?  
Why do we enclose wait in a while loop instead an if condition?*

## 3.1.4 Producer-Consumer problem

```
class Producer extends Thread {  
    Buffer buffer;  
  
    Producer(Buffer b) {  
        buffer = b;  
    }  
  
    public void run() {  
        java.util.Random r = new java.util.Random();  
        while (true) {  
            try {  
                buffer.insert(r.nextInt(200));  
            } catch (InterruptedException e) {  
                e.printStackTrace();  
            }  
        }  
    }  
  
    class Consumer extends Thread {  
        Buffer buffer;  
        Consumer(Buffer b) {  
            buffer = b;  
        }  
  
        public void run() {  
            while (true) {  
                try {  
                    int x = buffer.remove();  
                    System.out.println(x);  
                } catch (InterruptedException e) {  
                    e.printStackTrace();  
                }  
            }  
        }  
    }  
}
```

```
public class ProducerConsumer {  
  
    public static void main(String args[]) {  
  
        Buffer buffer = new Buffer();  
        for (int i = 0; i < 4; i++)  
            (new Producer(buffer)).start();  
        for (int i = 0; i < 3; i++)  
            (new Consumer(buffer)).start();  
    }  
}
```

### **3.1 Semaphores**

**3.1.1 Introduction**

**3.1.2 Mutual Exclusion**

**3.1.3 Condition Synchronization**

**3.1.4 Producer-Consumer Problem**

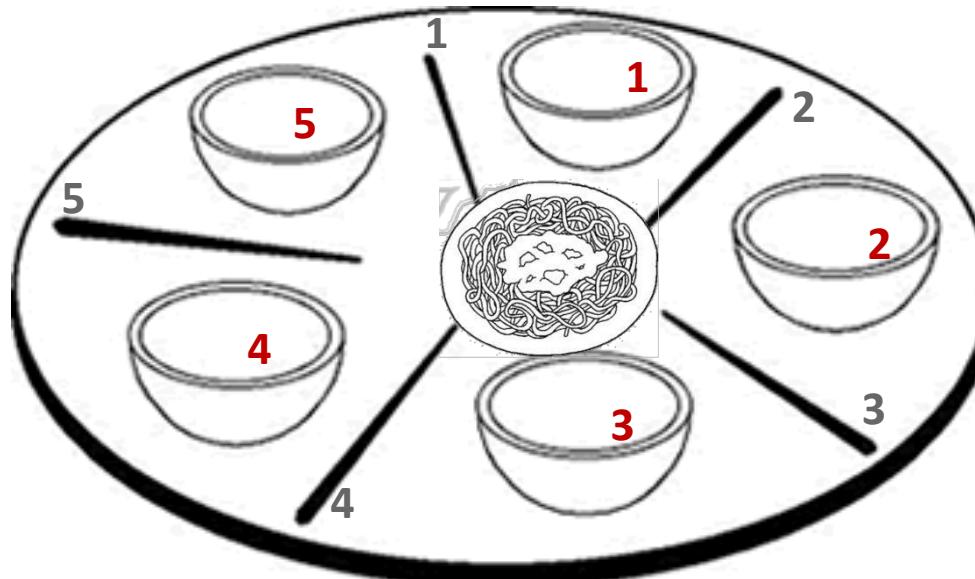
**3.1.5 The Dining Philosophers**

**3.2 CCR**

**3.3 Monitors**

## 3.1.5 The dining philosophers

- Five philosophers are engaged in only two activities: thinking and eating.
- Meals are taken at a table set with five plates and five chopsticks. In the center of the table is a bowl of spaghetti that is endlessly replenished.
- When a philosopher is hungry, he sits down and **uses the chopsticks on his left and right sides**. Thus, two philosophers sitting together cannot eat at the same time.



*Chopsticks are the shared resources*

- **Philosopher  $i$  takes chopstick  $i$  and  $i+1$**

philosopher

\* thinking \*

\* eating \*

## 3.1.5 The dining philosophers

- Being each philosopher a process, they will do the following actions:

```
process type philosopher(id:: integer);
begin
repeat
  think;
  sit;
  take left and right chopsticks
  eat;
  release chopsticks
forever
end;
```

- How can we code the *think*, *sit* and *eat* actions?
- **The core problem is how to synchronize them when taking the chopsticks**

## 3.1.5 The dining philosophers

- The given solution should meet the following criteria:
  1. One chopstick is held by just 1 philosopher: mutual exclusion
  2. One philosopher can eat only if he holds the left and right chopsticks: condition synchronization.
  3. Free from deadlock and livelock
  4. Free from starvation
  5. *If possible, be efficient: more than 1 philosopher should be able to eat at the same time.*
- We will implement 2 solutions:
  1. Mutex semaphores for chopsticks (produces deadlock)
  2. Allow only 4 philosopher to be sitting (it is not efficient)

# 3.1.5 The dining philosophers – Sol.1

```

program diningPhilosophersSol1;
{S01: mutex chopsticks}
const N=5;{5 philosophers}
var chopstick:array[1..N] of semaphore;

process type tPhilosopher(id:integer);
begin
  repeat
    sleep(random(2)); {THINK and SIT}
    wait(chopstick[id]);
    wait(chopstick[(id MOD N)+1]);
    writeln('[' , id , '] eating;');
    sleep(random(2)); {EAT}
    signal(chopstick[id]);
    signal(chopstick[(id MOD N)+1]);
  forever;
end;

var
  phils : array[1..N] of tPhilosopher;
  i: integer;
begin
  for i:=1 to N do
    initial(chopstick[i],1);
  cobegin
    for i:=1 to N do
      phils[i](i);
  coend;
end.

```

*If the interleaving of instructions  
is such that all philosophers take their  
left chopstick, they will wait forever  
eating for their right chopstick: deadlock*

*Sol.2: allow only N-1 philosophers to sit down at  
the same time.*

*What kind of semaphore do we use to implement  
this idea: only N-1 chairs available.*

# 3.1.5 The dining philosophers – Sol.2

```
program diningPhilosophersSol2;
{Sol2: mutex chopsticks, and N-1 chairs}
const N=5;{5 philosophers}
var
  chopstick:array[1..N] of semaphore;
  chairs:semaphore;

process type tPhilosopher(id:integer);
begin
  repeat
    sleep(random(2)); {THINK}
    wait(chairs); {SIT}
    wait(chopstick[id]);
    wait(chopstick[(id MOD N)+1]);
    writeln([' ',id,' ] eating;');
    sleep(random(2)); {EAT}
    signal(chopstick[id]);
    signal(chopstick[(id MOD N)+1]);
    signal(chairs);
  forever;
end;
Var

  phils : array[1..N] of tPhilosopher;
  i: integer;
begin
  for i:=1 to N do
    initial(chopstick[i],1);
  initial(chairs,N-1);
  cobegin
    for i:=1 to N do
      phils[i](i);
  coend;
end.
```

PASCAL-FC

*This solution is correct.*

*At any moment, at least 1 philosopher will be able to eat.*

*But having just 1 out of N philosophers eating is not efficient.*

*There exist more efficient solutions, which can be presented as Task 1:*

*Sol3:*

- *Odd philosophers first wait on left chopstick.*
- *Even philosophers first wait on right chopstick.*

*Sol4:*

*-The last philosopher first waits on a different chopstick than the others.*

## 3.1.5 The dining philosophers

- Next, Solution 2 is implemented in Java.
- Chopsticks are classes which provide methods *take()* and *release()*.
- Actions on chairs are performed through the methods provided in class Chairs.
- Philosophers are threads.
  
- Mutual exclusion is necessary in actions on chopsticks and chairs. This can be achieved by using object Semaphore, or with the *synchronized* modifier. We will use the latter.

# 3.1.5 The dining philosophers Sol.2

```
class Chopstick{
    boolean free=true;

    synchronized public void take() {
        while(!free) {
            try{
                wait();
            }catch(Exception e) {
                e.printStackTrace();
            }
            free=false;
        }

        synchronized public void release() {
            free=true;
            notifyAll();
            //all blocked phil. will try to
            //take it again
        }
    }
}
```

```
class Chairs{

    int max, busy;

    Chairs(int m) {
        max=m;
        busy=0;
    }

    synchronized public void sit() {
        while(busy==max) {
            try{wait();
            }catch(Exception e) {
                e.printStackTrace();
            }
            busy++;
        }

        synchronized public void standUp() {
            busy--;
            notifyAll();
        }
    }
}
```

# 3.1.5 The dining philosophers Sol.2

```

class Philosopher extends Thread{
int id;
Chopstick rightCh, leftCh;
Chairs chairs;

public Philosopher(int ID, Chairs c,
    Chopstick r, Chopstick l){
id=ID;
chairs=c;
rightCh=r;
leftCh=l;
}

public void run(){
java.util.Random r=new java.util.Random();
while(true){
try {//THINK
Thread.sleep(r.nextInt(2000));
} catch (InterruptedException e) {
e.printStackTrace();
}
chairs.sit();//SIT
rightCh.take();
leftCh.take();
System.out.println("[ "+id+"] eating.");//EAT
rightCh.release();
leftCh.release();
chairs.standUp();
}}}

```

```

public class DiningPhilosophersSol2{
public static final int N=5;

public static void main(String args[]){
Chairs chairs=new Chairs(N-1);
Chopstick chopstick[]=new Chopstick[N];

for(int i=0;i<N;i++)
chopstick[i]=new Chopstick();

for(int i=0;i<N;i++)
new Philosopher(i,chairs,chopstick[i],
chopstick[(i+1)%N]).start();
}
}

```

# Conclusions

You may have already reached the following conclusion:

- Semaphores have advantages:
  - Easy and efficient solution for mutual exclusion and synchronization
  - Their primitives are available in most of concurrent programming languages
- But they have several disadvantages:
  - Their level of abstraction is low
  - Error-prone: their use depends on the programmer (lost signal...)
  - Readability and maintenance of the code is not straightforward (they are deployed along the code without any structure).
  - At first glance, we cannot tell which semaphores are used for mutual exclusion or which for synchronization.
- These disadvantages (specially, the last one) are alleviated by using CCR and Monitors. But they are not available in all languages!

# Potential Mid-term Exam Questions

- 1. What is a cyclic barrier? How can we use it in Pascal-FC and Java?  
Is a barrier (rendez-vous with more than two processes) which can be reused after the waiting threads are released.  
we need to implement it using semaphores.
- 2. If we were to implement a semaphore, what variables and procedures do we need to code?  
wait() and signal() surrounding the CS (pre protocol and post protocol)  
declare the semaphores  
initialise the semaphores
- 3. What is the most general purpose of binary semaphores? And general?  
Providing mutual exclusion for the binary semaphores  
General are used for providing access to a resource that has multiple instances, or provide multiple-exclusion.

# Potential Mid-term Exam Questions

- 4. What effect do *wait*, *notify* and *notifyAll* signals have on threads in Java?

*wait()* -> blocks a process until a condition holds (another process)

*notify()* -> wakes one process

*notifyAll* -> unblocks all blocked processes

- 5. Read again Solution 1 of the Dining Philosophers problem. Write a possible interleaving which leads to a deadlock situation



# Keywords phonetics

- semaphore /'seməfɔ:r/ 
- procedure /prə'si:dʒər/ 
- rendezvous /'rəndeɪvu:/ 
- cyclic /'saɪklɪk/ 
- barrier /'bæriər/ 
- philosopher /fɪ'lɒsəfər/ 
- signal /'sɪgnl/ 
- binary /'bainəri/ 
- multiple /'mʌltipl/ 



(i) https://campusvirtual.udlm.es/mod/quiz/attempt.php?attempt=723454& 90% ...

Usted se ha identificado como DIANA ALONSO SÁIZ

¿Qué efecto tiene el modificador *volatile* aplicado sobre una variable en Java?

Seleccione una:

- a. Permite que una variable sea accesible por varios hilos concurrentemente.
- b. Garantiza la exclusión mútua en el acceso a la variable cuando ésta es modificada por varios hilos
- c. Permite que una variable sea modificada por varios hilos.
- d. Asegura que si varios hilos acceden a una misma variable éstos siempre tendrán el último valor actualizado de la variable aunque dicha variable haya sido modificada por otro hilo.

## Pregunta 2

Sin responder aún

Puntúa como 2,00

Marcar pregunta

✓ Describe la semántica de los operadores *wait(s)* y *signal(s)* sobre semáforos generales.

*wait()* --> Operación atómica que si el estado=0 el proceso estará atrapado en un semáforo hasta que se despierte por otro proceso. Si el semáforo no es nulo, decrementa el valor del contador en 1 y el proceso continúa con su ejecución.

*signal()* --> Operación atómica que una vez se ha terminado el uso del recurso, el proceso lo señala al semáforo y si hay algún proceso bloqueado se despierta. Y si no, incrementa el contador.



Usted se ha identificado como DIANA ALONSO SÁIZ

## Pregunta 3

Sin responder aún

Puntúa como 1,00

▼ Marcar pregunta

Imagina 5 procesos ejecutando el siguiente código en Java con una variable compartida **volatile**  
**int v=0;**

**for(int j=0; j<3;j++) v++;**

¿Podemos decir que el valor final de v es 15?

Seleccione una:

- a. No, puede haber multiples intercalaciones de instrucciones atómicas
- b. Sí porque v utiliza es del tipo simple int
- c. Sí, porque v es volátil
- d. No, el resultado de v será 3.

[Siguiente página](#)



Usted se ha identificado como DIANA ALONSO SÁIZ

En qué arquitecturas no es directamente aplicable el modelo de variables compartidas

Seleccione una:

- a. Arquitecturas multiprocesador multiprocessor / multicore
- b. Arquitecturas distribuidas distributed systems
- c. Arquitecturas monoprocesador un processor
- d. En cualquiera de ellas es aplicable

Pregunta 5

Sin responder aún

Puntúa como 1,00

▼ Marcar pregunta

? Sólo un hilo puede estar en cualquiera de los métodos synchronized de un Objeto en un momento dado.

Seleccione una:

- a. VERDADERO
- b. FALSO

Pregunta 6

Sin responder aún

Puntúa como 1,00

▼ Marcar pregunta

La programación concurrente no tiene sentido en sistemas monoprocesador

Seleccione una:

- a. FALSO
- b. VERDADERO



Usted se ha identificado como DIANA ALONSO SÁIZ

¿Cuál es la ventaja de utilizar el interface *Runnable* para crear hilos?

Seleccione una:

- a. Podemos extender otra clase.
- b. Los Hilos se ejecutan más rápido
- c. Podemos tener varios métodos run()
- d. Ninguna es correcta
- e. Código más sencillo

Pregunta 8

Sin responder aún

Puntúa como 1,00

▼ Marcar pregunta

La unión de varias instrucciones atómicas también es atómica.

Seleccione una:

- a. Verdadero si sólo aparecen variables simples
- b. verdadero si son operaciones de asignación
- c. Verdadero
- d. Falso

Usted se ha identificado como DIANA ALONSO SÁIZ

bloqueo

```
program ProdCons2;

var producto : integer;
    producido, consumido : boolean;

process productor;
var i : integer;
begin
  for i := 1 to 100 do
    begin
      producto := i;
      producido := true; {-1-}
      consumido := false;{-2-}
      while (NOT consumido) do NULL;
    end;
end;

process consumidor;
begin
  repeat
    while (NOT producido) do NULL;
    writeln(' Consumo');
    consumido := true;
    producido := false;
  forever;
end;

begin
  producido := false;
  producto := -1;
  cobegin
    productor;
    consumidor;
  coend;
end.
```

Seleccione una:

- a. 3-4-1-2
- b. 1-3-2-4
- c. 3-1-2-3-4



Usted se ha identificado como DIANA ALONSO SÁIZ

## Pregunta 10

Sin responder aún

Puntúa como 2,00

▼ Marcar pregunta

Qé es una condición de carrera?

Competence (with resources available  
in memory)

Múltiples procesos se encuentran en condición de carrera si el resultado de los mismos dependen del orden de su ejecución. Si los procesos que están en condición de carrera no son correctamente sincronizados puede producirse una corrupción de datos.

## Pregunta 11

Sin responder aún

Puntúa como 1,00

▼ Marcar pregunta

Una cita doble es un tipo de:

rendez-vous

Respuesta: Sincronización condicional

condition synchronization



Usted se ha identificado como DIANA ALONSO SÁIZ

Sin responder aún

Puntúa como 2,00

▼ Marcar pregunta

✓ En Java ¿cuál es el efecto de `wait()`, `notify()` y `notifyAll()` en los hilos?

`wait()` --> Se usa con el fin de, parar el hilo según la condición.

`notify ()` --> Se usa para mandar una señal al hilo, despertando a un proceso que estaba en espera.

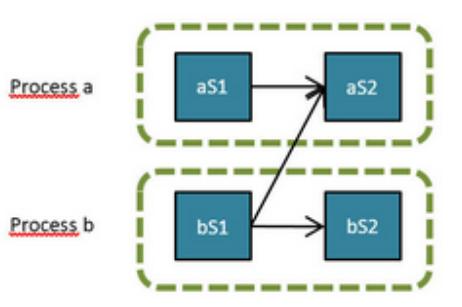
`notifyAll()` --> Se usa para despertar a todos los hilos que estaban esperando.



Usted se ha identificado como DIANA ALONSO SÁIZ

Puntuación 1,00

▼ Marcar pregunta



Una posible salida de estos procesos según este gráfico sería:

Seleccione una:

- a. aS1 bS1 aS2 bS2
- b. Todas son posibles
- c. bS1 bS2 aS1 aS2
- d. aS1 bS1 bS2 aS2



Usted se ha identificado como DIANA ALONSO SÁIZ

¿Cuál es el número mínimo de hilos que contiene un programa en Java?

Seleccione una:

- a. Un Hilo  
 b. Puede no contener ningún hilo  
 c. Dos

Pregunta 15

Sin responder aún

Puntúa como 2,00

▼ Marcar pregunta

✓ ¿Qué es la espera activa? ¿Cuál es la diferencia con la espera pasiva?

Espera activa --> Es una técnica en la cual un proceso comprueba continuamente si se cumple una condición o si produce un evento.

Se diferencia en espera pasiva porque ésta es cuando un proceso no puede continuar ejecutando las sentencias porque está bloqueado y deja de ejecutarlas hasta que otro proceso lo desbloquea.



Usted se ha identificado como DIANA ALONSO SÁIZ



¿Por qué las sentencias wait(s) y signal(s) garantizan la exclusión mútua en el acceso al semáforos ?

Seleccione una:

- a. No la garantizan
- b. Porque se usan en un mutex
- c. Sólo si el semáforo se inicializa a 1
- d. Porque son atómicas

#### Pregunta 17

Sin responder aún

Puntúa como 2,00

Marcar pregunta

Define

- a) Exclusión mútua
- b) Sincronización condicional

 Exclusión mutua: Algoritmo que evita que entre más de un proceso a la vez en la sección crítica, es decir, dos procesos no pueden usar el mismo recurso a la vez.

Sincronización condicional: Se produce cuando un proceso debe esperar a que se cumpla una cierta condición para proseguir su ejecución, ésto solo puede ser activada por otro proceso.



Usted se ha identificado como DIANA ALONSO SÁIZ

muestren su salida de forma ordenada:

A, B, B, A, B, B, A, B, B, A, B, B, A, B, B...

Debes indicar qué código pondrías en cada apartado {puede ser más de una instrucción, algunos apartados pueden estar vacíos}



- |                     |                                |
|---------------------|--------------------------------|
| {-3}                | signal (sB);                   |
| {-1- Declaraciones} | var sA, sB : semaphore;        |
| {6}                 | signal(sA);                    |
| {4}                 | wait (sA);                     |
| {-2-}               | initial(sA,1); initial (sB,0); |
| {5}                 | wait (sB);                     |



Usted se ha identificado como DIANA ALONSO SÁIZ

**Pregunta 19**

Sin responder aún

Puntúa como 1,00

Marcar pregunta



La sincronización necesaria para que varios procesos no accedan al mismo tiempo a un recurso compartido se denomina :

Respuesta: Exclusión mutua

**Pregunta 20**

Sin responder aún

Puntúa como 2,00

Marcar pregunta



El método interrupt() de la clase Thread interrumpe y para un hilo en ejecución.

Seleccione una:

- a. FALSO  
 b. VERDADERO



Usted se ha identificado como DIANA ALONSO SÁIZ

**Pregunta 21**

Sin responder aún

Puntúa como 2,00

Marcar pregunta

 ¿Cuál es la diferencia entre los métodos **start** y **run** en un Thread Java?

**run()** --> Ejecuta el bloque del código escrito en el método **run()**.

**start()** --> Inicializa el hilo y ejecuta lo que tiene el método **run()**.



Usted se ha identificado como DIANA ALONSO SÁIZ

## Pregunta 22

Sin responder aún

Puntúa como 1,00

▼ Marcar pregunta



Asocia los posibles estados de un hilo en Java con su significado

Runnable

Cuando el método start() es llamado el hilo entra en este estado. Se está ejecutando.

Running

El scheduler ha seleccionado el hilo para ejecutarlo

Waiting/Blocked  
/Sleeping

El Hilo está vivo pero no es elegible para ejecutarse

Terminated  
(Dead)

El Hilo ha terminado de ejecutar su método run()

New

En este estado un hilo aún no ha sido iniciado. (Su método start() no ha sido ejecutado)



Usted se ha identificado como DIANA ALONSO SÁIZ

Marcar pregunta

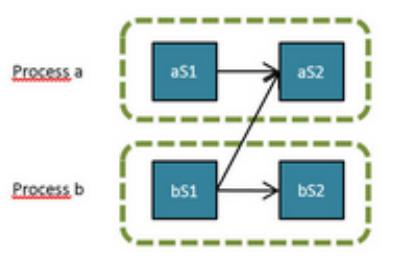
?

El siguiente código:

```
var  
    continue: boolean;  
  
process ProcessA;  
begin  
    write('aS1');  
    continue := true;  
    write('aS2');  
end;
```

```
process ProcessB;  
begin  
    write('bS1');  
    while not continue do;  
        write('bS2');  
    end;
```

```
begin  
    continue := true;  
cobegin  
    ProcessA;  
    ProcessB;  
coend  
end.
```



Seleccione una:

- a. Sincroniza el diagrama propuesto
- b. Bloquea el proceso B esperando a que A termine.
- c. Ninguna es correcta
- d. Produce un DeadLock

Pregunta **24**

Sin responder aún

Puntúa como 1,00

 Marcar pregunta

En una ejecución paralela..

Seleccione una:

-  a. Hay 2 o más procesos ejecutándose al mismo tiempo usando una CPU por proceso
- b. Hay cualquier número de procesos ejecutándose en una CPU
- c. Todas son correctas
- d. Hay múltiples procesos ejecutándose en una o varias CPUs



# PROGRAMACIÓN CONCURRENTE Y TIEMPO REAL

## Pregunta 25

Sin responder aún

Puntúa como 2,00

▼ Marcar pregunta

?

Sólo podemos utilizar notify(), notifyAll() y wait() en métodos *synchronized*.

Seleccione una:

- a. FALSO  
 b. VERDADERO

# Unit 3

## Shared-memory Communication

### 3.2 CONDITIONAL CRITICAL REGIONS

3.1 Semaphores

3.2 CCR

- 3.2.1 Introduction
- 3.2.2 Critical Region
- 3.2.3 Conditional Critical Region
- 3.2.4 Producer-Consumer problem
- 3.2.5 Readers-Writers problem
- 3.2.6 The dining philosophers problem

3.3 Monitors

### **3.1 Semaphores**

### **3.2 CCR**

#### **3.2.1 Introduction**

**3.2.2 Critical Region**

**3.2.3 Condition Critical Region**

**3.2.4 Producer-Consumer problem**

**3.2.5 Readers-Writers problem**

**3.2.6 The dining philosophers problem**

### **3.3 Monitors**

## 3.2.1 Introduction

- Semaphores:
  - are low level abstraction tools for mutual exclusion and synchronization.
  - their syntax is the same when they are used for both kinds of interaction among processes.
  - It is easy to forget one wait or signal operation.
- In 1972, Hoare and Brinch Hansen proposed and made popular the notion of Conditional Critical Region (CCR).
- CCRs tell the compiler where the mutual exclusion (CR) and condition synchronization (CCR) statements are, and it deals with the deployment of wait and signal operations.

### **3.1 Semaphores**

### **3.2 CCR**

#### **3.2.1 Introduction**

#### **3.2.2 Critical Region**

#### **3.2.3 Condition Critical Region**

#### **3.2.4 Producer-Consumer problem**

#### **3.2.5 Readers-Writers problem**

#### **3.2.6 The dining philosophers problem**

### **3.3 Monitors**

## 3.2.2 Critical Region

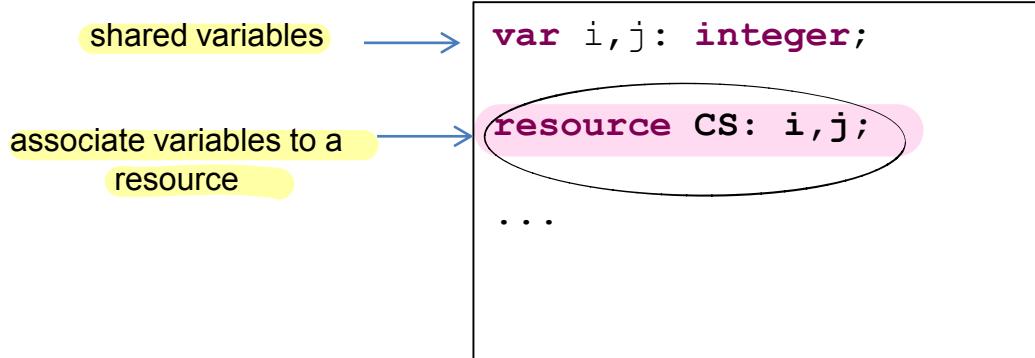
- A critical section is a piece of code that should be executed under mutual exclusion. It depends on the programmer that this happens.
- A **critical region (CR)** is a piece of code that is executed under mutual exclusion. The programmer does not need to take care of protocols nor correct use of wait/signal calls.



- CR and CCRs can be seen as precursor of monitors, and they are available just in a few programming languages (e.g. Ada 9X).
- CR and CCR are not available in Pascal-FC nor Java. We will learn its use using Pascal-FC syntax and reserved word *resource*, but it must be read just as pseudo-code.

## 3.2.2 Critical Region

- Shared variables must be declared as usual, and then associate them to a **resource**.



- Since **variables** are attached to a resource, they **can only be accessed** by explicitly using the keyword **region**, and the name of the resource which holds the variable.

```
region CS do  
begin  
  i:= i+1;  
  ...  
end;
```

## 3.2.2 Critical Region

- All regions with the same resource name will be executed under mutual exclusion.
- If a variable associated to a shared resource is accessed directly from code, the compiler flags an error.
- One variable can only be associated to 1 resource.
- Nested CRs may lead to a deadlock situation:

```
process p1:  
...  
region A do  
  region B do  
    S;
```

```
process p2:  
...  
region B do  
  region A do  
    S;
```

### **3.1 Semaphores**

### **3.2 CCR**

#### **3.2.1 Introduction**

#### **3.2.2 Critical Region**

#### **3.2.3 Conditional Critical Region**

#### **3.2.4 Producer-Consumer problem**

#### **3.2.5 Readers-Writers problem**

#### **3.2.6 The dining philosophers problem**

### **3.3 Monitors**

### 3.2.3 Conditional Critical Region

- So CRs provide mutual exclusion by making the compiler be responsible of the use of semaphores.
- A **Conditional Critical Region (CCR)** is an extension of CR which lets express a condition to gain access to the resource: condition synchronization.
- Resources are declared the same way than CRs.
- Access to resources now includes a condition:

```
var i,j: integer;  
  
resource CCS i,j;  
...  
region CCS when <condition> do  
    begin  
    ...  
end;
```

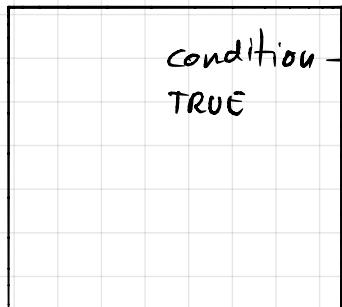
### 3.2.3 Conditional Critical Region

- Thus, a process trying to enter a CCR runs as follows:
  1. A process remains blocked in the **main queue** until it gains mutual exclusion access to the region.
  2. Once access is obtained,
    1. if *condition* is true, statements in the CCR are executed.
    2. else, the process releases the mutual exclusion access and is blocked in the **events queue**.
  3. Once the execution of statements finishes:
    1. Processes in the events queue are allowed to test the condition again. If the condition is met, the first process is unblocked.
    2. Else, the first process in the main queue is unblocked.
- This means that a process which has gained mutual exclusion access once is not required to gain it again. That is, it has a higher priority than new incoming processes: **EVENTS QUEUE > MAIN QUEUE**

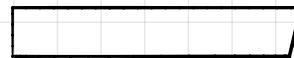
main queue



critical  
region



FALSE



events queue

(higher priority)

1st checks the condition

### 3.2.3 Conditional Critical Region

- In order to give a full example, let's see how to implement semaphores using CCR:
  - One shared variable to be used as counter of permits (mutual exclusion)
  - Wait can only be executed under a given condition if we do not allow negative semaphores.
  - Signals do not need condition

```
program semaphore;  
var  
  s: integer;  
  resource sem : s;  
  
procedure wait;  
begin  
{if s=0, process is blocked in the events queue}  
  region sem when s>0 do  
    s:= s-1;  
  end;
```

```
procedure signal;  
begin  
  region sem do  
    s:= s+1;  
  end;
```

### **3.1 Semaphores**

### **3.2 CCR**

**3.2.1 Introduction**

**3.2.2 Critical Region**

**3.2.3 Conditional Critical Region**

**3.2.4 Producer-Consumer problem**

**3.2.5 Readers-Writers problem**

**3.2.6 The dining philosophers problem**

### **3.3 Monitors**

## 3.2.4 Producer-Consumer problem

- Again, we instantiate the problem with a finite buffer, and several producers and consumers.
- Critical sections are those in which the following variables are accessed:  
*insertIndex, removeIndex, numItems*
- We define 2 condition synchronizations between processes:
  - Producers cannot insert items if buffer is full.
  - Consumers cannot remove items if buffer is empty.

## 3.2.4 Producer-Consumer problem

```
program producerConsumerCCR;
const SIZE=8;
var
numItems, insertIndex,
      removeIndex, i: integer;
data: array[1..SIZE] of integer;
resource buffer: insertIndex,
      removeIndex, numItems;
```

```
process type tProducer;
var
item: integer;

begin
repeat
  item := random(200);
  region buffer when numItems < SIZE do
    begin
      data[insertIndex]:=item;
      insertIndex:=insertIndex MOD SIZE + 1;
      numItems:=numItems+1;
    end;
  forever
end;
```

```
process type tConsumer;
begin
var
item: integer;
repeat
region buffer when numItems>0 do
begin
  item := data[removeIndex];
  removeIndex := removeIndex MOD SIZE+ 1;
  numItems:=numItems-1;
end;
forever
end;

var
prod:array [1..5] of tProducer;
cons:array [1..3] of tConsumer;
begin
numItems:=0;
insertIndex:=0;
removeIndex:=0;
cobegin
  for i:=1 to 5 do
    prod[i];
  for i:=1 to 3 do
    cons[i];
coend;
end.
```

### **3.1 Semaphores**

### **3.2 CCR**

- 3.2.1 Introduction**
- 3.2.2 Critical Region**
- 3.2.3 Conditional Critical Region**
- 3.2.4 Producer-Consumer problem**
- 3.2.5 Readers-Writers problem**
- 3.2.6 The dining philosophers problem**

### **3.3 Monitors**

## 4.2.5 Readers/Writers problem

- In this problem, several readers and writers access to the same resource (e.g. database).
- Several readers can read at the same time if no writers are writing.
- If one writer is writing, no reader nor extra writer can access.
- The solution is slightly different depending on which kind of process has priority when both are waiting to access:
  - Readers first
  - Writers first
- In this example, we decide to give priority to writers:
  - if a writer wants to access, it waits until current readers stop reading. Then no reader may enter until the waiting writer finishes its action.
  - We need 3 variables:
    - numReaders: number of processes of type Reader reading
    - numWaiting: num of processes of type Writer waiting to access
    - isWriting: true when a writer is accessing it

## 4.2.5 Readers/Writers problem

```
program readersWriters;
{PRIORITY WRITERS}
var
  numReading, numWwaiting: integer;
  isWriting: boolean;
resource data: numReading, numWwaiting,
           isWriting;

process type tReader;
begin
  (*readers wait if a writer is
  writing or waiting*)
  region data when not isWriting and
    numWwaiting=0 do (priority to writers)
    numReading:=numReading+1;
    writeln('Reading...');
  region data do
    numReading:=numReading-1;
end;
```

Thanks to the unique FIFO policy of the events queue, starvation among readers is not possible (it is using semaphores). It is possible if there are always writers willing to write.

```
process type tWriter;
begin
  {announce a new writer is waiting}
  region data do
    numWwaiting:=numWwaiting+1;
  {wait if a writer or reader is in}
  region data when numReading=0 and
    not isWriting
  begin
    isWriting:=true;
    numWwaiting := numWwaiting-1;
  end;
  writeln('Writing...');
  region data do isWriting:=false;
  end;

var
  read:array[1...5] of tReader;
  wri:array[1...3] of tWriter;
  i:integer;
begin
  numReading:=0;
  isWriting:=false;
  cobegin
    for i:=1 to 5 do
      read[i];
    for i:=1 to 3 do
      wri[i];
  coend
end.
```

### **3.1 Semaphores**

### **3.2 CCR**

- 3.2.1 Introduction**
- 3.2.2 Critical Region**
- 3.2.3 Conditional Critical Region**
- 3.2.4 Producer-Consumer problem**
- 3.2.5 Readers-Writers problem**
- 3.2.6 The dining philosophers problem**

### **3.3 Monitors**

# 3.2.6 The dining philosophers problem

- Critical sections: those in which chopsticks are taken.
- Cond. Synch: philosopher  $i$  cannot eat if he cannot take chopstick  $i$  and  $i+1$ .

```
program diningPhilosophers;
const N=5;
var
{true when they are available}
chopstick:array[1..N] of boolean;
resource chopsCCR:chopstick;

process type tPhilosopher(id:integer);
begin
repeat
  sleep(random(2)); {THINK and SIT}
  region chopsCCR when chopstick[id] and
    chopstick[(id+1) MOD N] do
    begin
      chopstick[id]:=false;
      chopstick[(id+1) MOD N]:=false;
    end;
  sleep(random(2)); {EAT}
  region chopsCCR do
    begin
      chopstick[id]:=true;
      chopstick[(id+1) MOD N]:=true;
    end;
  forever
end;
```

```
var
phils : array[1..N] of tPhilosopher;
i: integer;
begin
for i:=1 to N do
  chopstick[i]:=true; initialize
  cobegin
    for i:=1 to N do
      philis[i](i);
  coend;
end.
```

Using semaphores (solution 1) deadlock could happen.  
Using CCRs this is not possible, since a philosopher only enters in the CCR when both chopsticks are available.

# Conclusions

- CCRs let us distinguish between critical sections and condition synchronization.
- They may be difficult to implement (two process queues)
- Deadlock risk when embedding CCRS
- They still keep one problem of semaphores: their use is widespread along the code: difficult to maintain.

# Potential Mid-term Exam Questions

1. Which processes queue has higher priority in CCR?

Events queue

2. What correctness problem do CCRs solve in the Dining Philosophers problem which needed a special solution when using semaphores?

Deadlock, because a philosopher only enters the CR when both chopsticks are available.

3. What is the difference between CR and CCR?

- CR provides mutual exclusion by making the compiler be responsible for the use of the semaphores. Programmer is not responsible for protocols, signal/w8.
- CCR is an extension of a CR in which we express a condition that needs to be fulfilled to gain access to the resource, then it also provides condition synchronization).

## The elevator problem

program elevator;

constant N = 4;

var

up: boolean;

level, inside : integer;

resource data: up, level, inside;

process type Person;

begin;

# Keywords phonetics

- region /'ri:dʒən/ 
- resource /rɪ'zɔ:s/ 
- priority /prai'ɒriti/ 

# Unit 3

## Shared-memory Communication

### 3.3 Monitors

3.1 Semaphores

3.2 CCR

**3.3 Monitors**

    3.3.1 Introduction

    3.3.2 Monitors

        -Mutual Exclusion

        -Condition Synchronization

    3.3.3 Producer-Consumer problem

    3.3.4 Monitors in Java

**3.1 Semaphores**

**3.2 CCR**

**3.3 Monitors**

**3.3.1 Introduction**

**3.3.2 Monitors**

-Mutual Exclusion

-Condition Synchronization

**3.3.3 Producer-Consumer problem**

**3.3.4 Monitors in Java**

### 3.3.1 Introduction

- We learned that semaphores are concurrent programming solutions which have several problems. Can you remember a few of them?
- CCR solved some of these problems:
- But still provided a resulting code with sparse use of global variables aimed to control the concurrent execution.
- Solution: Monitors

### 3.3.1 Introduction

- Monitors were proposed by C.A.R. Hoare in 1974.
- A monitor is an encapsulation of resources, and operations which can be applied on them.
- Variables declared in a monitor can be accessed only from a procedure exported from the same monitor.
- All procedures are executed under mutual exclusion; the programmer does not need to explicitly manage access to critical sections anymore.
- Processes are queued inside a Condition queue<sup>FIFO</sup>, when a condition is not met. It will wait until another process makes the necessary change.
  - Problems detected in compilation time

### 3.3.1 Introduction

- Consequently, a **monitor** solves all problems mentioned when using semaphores and CCR:
  - **Modular**: the code written to control concurrent processes is inside a unique structure.
  - Local variables: variables used are local in the monitor. **The programmer will not find global variables along the code.**
  - The programmer cannot access, by error, shared variables.
  - **Meaningful**: the code inside the monitor helps the programmer to know what is the **aim** of the monitor (mutual exclusion or synchronization).
    - ↳ monitor associated to a resource
- **Active process**: an outer process which calls a procedure inside the monitor. We say the process is *inside* the monitor. (use)
- If the monitor creates processes inside it, these are called **passive processes**.  
(implement)      ↳ wait other processes (active) uses the monitor

**3.1 Semaphores**

**3.2 CCR**

**3.3 Monitors**

**3.3.1 Introduction**

**3.3.2 Monitors**

**-Mutual Exclusion**

**-Condition Synchronization**

**3.3.3 Producer-Consumer problem**

**3.3.4 Monitors in Java**

## 3.3.2 Monitors

- Monitors are available as primitive in Pascal-FC, but not in Java. The generic structure in Pascal-FC is:

```
monitor name;  
  export exported_procedures; (interface)  
  var local_variables;  
  
  procedure P1 (parameters);  
  var local_variables;  
  begin  
    {some code}  
  end;  
  ...  
  procedure Pn (parameters);  
  var local_variables;  
  begin  
    {some code}  
  end;  
begin  
  {initiation code}  
end;
```

- A set of local variables called **permanent** variables. They indicate the state of the resource represented by the monitor.
- An **initiation code**. This code is executed once, when the monitor is created. It is used to initiate the permanent variables.
- One or more procedures which manage the value of the permanent variables.
- A list of **exported procedures** which can be accessed from outside the monitor by the programmer. The **export** keyword must be the first to appear in the monitor code.
- Procedures which are not exported are inner procedures.

## 3.3.2 Monitors

- When an active process needs to use a shared resource which is represented/controlled by a monitor, the process must call an exported procedure from the monitor.
- This call is performed in Pascal-FC as follows:  
*monitorName.procedure(arguments)*
- That is, the name of the monitor followed by the name of the exported procedure (and arguments, if required).
- Let us see how the monitor provides:
  - mutual exclusion access to procedures inside the monitor, and
  - processes synchronization

## 3.3.2 Monitors – mutual exclusion

- A monitor uses a processes queue called **monitor queue**.
- The monitor queue is managed as follows:
  - When an active process is inside the monitor, and **another active process tries to enter** the monitor through another (or the same) exported procedure than the former, the latter is blocked in the monitor queue (which has a FIFO behaviour).
  - When an **active process finishes** the execution of an exported procedure, it leaves the monitor. Then:
    - If the monitor queue is empty, the first active process which tries to gain access will enter.
    - Else, the process in the head of the monitor queue is released and gains access to the monitor.

## 3.3.2 Monitors – mutual exclusion

- Write a Pascal-FC monitor which lets us increment the value of a variable and printing its value. The access must be performed under mutual exclusion.

```
program incrementing;

monitor monit;
export inc, value;
var i:integer;

procedure inc;
begin
i:=i+1;

end;

procedure value;
begin
writeln('---->',i)
end;

begin
i:=0;
end;
```

```
process type P;
begin
repeat
monit.inc;
monit.value;
forever
end;

var
p1,p2,p3:P;
begin
cobegin
p1;
p2;
p3;
coend
end.
```

## 3.3.2 Monitors – condition synch.

- The previous example allows us to increment and print the value of a variable under mutual exclusion. But the increment&printing operations are not synchronized, so the output may be a little ‘dumb’.
- We need to learn how to synchronize processes under a given condition inside monitors.
- This is performed by using **condition variables**: variables inside a monitor which represent FIFO queues.
- An active process is **blocked in a condition variable** when it cannot continue its execution (e.g.: producer blocked because buffer is full). It will resume its execution when the situation changes (e.g.: the consumer takes an item from the buffer).

## 3.3.2 Monitors – condition synch.

- Declaration in Pascal-FC of 2 condition variables and an array of condition variables:

```
var  
cond_A, cond_B: condition;  
conditions: array [1..5] of condition;
```

- **Disambiguation** of term “condition”:
  1. The situation which makes a process block until it is changed by another process.
  2. A type of variable which represents a FIFO queue. If there are several kinds of processes, then each kind of process is queued in a different condition variable. For example, in the Producer/Consumer problem, producers are blocked in condition variable A, and consumers in condition variable B.

## 3.3.2 Monitors – condition synch.

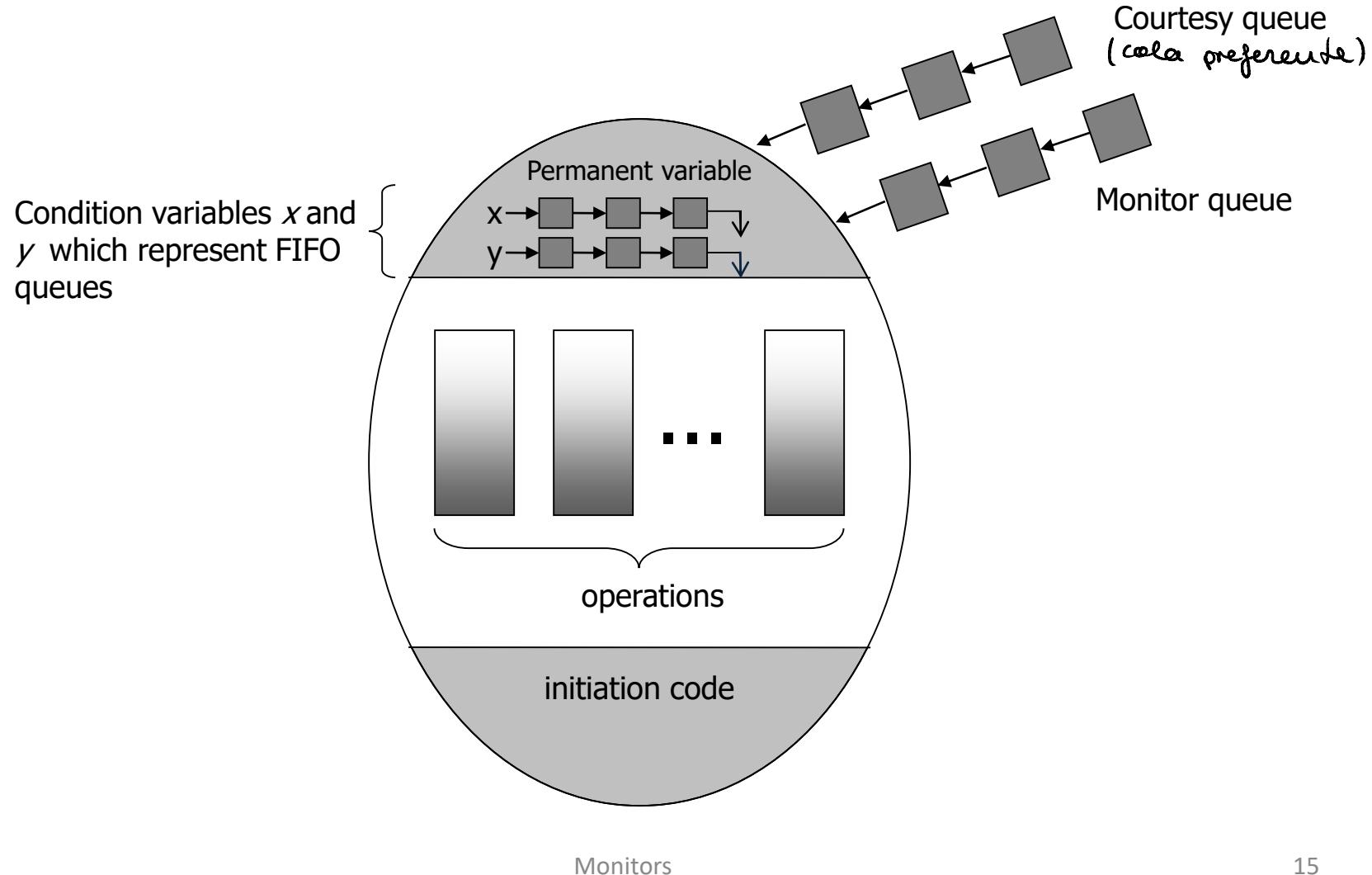
- Pascal-FC provides three necessary operations on a condition variable C.
  - **empty(C)**: it returns a boolean value. True if there are not active processes blocked in C, False otherwise.
  - **delay(C)**: the active process which executes this operation **releases the mutual exclusion** hold on the monitor, **and is queued** in condition C. This is different from *wait(semaphore)* because the active process is always blocked in the condition, while semaphores only block processes when their value is 0.
  - **resume(C)**: the active **process blocked in the head** of the queue in C is **set ready** for execution. If the queue is empty, this operation is null (does nothing); this is different from *signal(semaphore)* because signal always increases the value of the semaphore so it is never a null operation.

## 3.3.2 Monitors – condition synch.

- When an active process P1 executes *resume(C)* and then P2 is set ready for execution, which processes goes on running: P1 or P2? (both together cannot because this violates the mutual exclusion guaranteed by the monitor).
- The possible solutions are known as **semantic of the resume operation**.
  - Resume and Continue (RC):** P2 is inserted back in the monitor queue. The situation which blocked P2 needs to be reevaluated in a while loop because once it re-enters the monitor, the situation might have changed again. Java wait-notify signals use the RC semantic.  
*P2 goes to monitor queue*
  - Resume and Exit (RE):** P1 returns from the monitor (never comes back) and P2 resumes its execution inside the monitor. Thus, P2 does not need to reevaluate the situation because it was just changed. Concurrent Pascal uses RE.  
*P1 goes out the monitor and never return*
  - Resume-and-Wait (RW):** P1 returns from the monitor and is inserted again in the monitor queue. P2 resumes its execution. Modula-2, Concurrent Euclid.  
*P1 goes to the monitor queue*
  - Resume-and-Urgent-Wait (RUW):** the same than RW, but now P1 is inserted in the **courtesy queue**, which has higher priority than the monitor queue when trying to gain access to the monitor. This is the semantic of resume in Pascal-FC: → **code below 'delay(C)' is run by P2 before than code below 'resume(C)' is run by P1**, which changed conditions to release P2 → so that is why it is a good practice to call *resume* in the **last line** of a process' code.

## 3.3.2 Monitors – condition synch.

- Thus, the final structure of a Pascal-FC monitor (with RUW semantic) is this:



**3.1 Semaphores**

**3.2 CCR**

**3.3 Monitors**

**3.3.1 Introduction**

**3.3.2 Monitors**

**-Mutual Exclusion**

**-Condition Synchronization**

**3.3.3 Producer-Consumer problem**

**3.3.4 Monitors in Java**

### 3.3.3 Producer-Consumer problem

Let us solve the problem with limited buffer and several producers and consumers

- What resource is to be represented by the monitor?
- What are the permanent variables?
- What are the active processes?
- Do we need condition variables to queue active processes?
- What operations should the monitor export?

### 3.3.3 Producer-Consumer problem

```
program ProducerConsumer;

monitor buffer;
{list of operations to export}
export insert, remove;
{permanent variables}
const SIZE=10;
var
  numItems, insertIndex, removeIndex:integer;
  consumersC, producersC: condition;
  data: array[1..SIZE] of integer;

{operations}
procedure insert(item:integer);
begin
  if numItems=SIZE then delay(producersC);
  data[insertIndex]:=item;
  writeln('--->',item);
  insertIndex:= (insertIndex MOD SIZE) + 1;
  numItems:=numItems+1;
  resume(consumersC); → end of the code
end;
```

```
procedure remove(var item:integer);
{the item is passed by reference}
begin
  if numItems=0 then delay(consumersC);
  item:=data[removeIndex];
  writeln('<---',item);
  removeIndex:=(removeIndex MOD SIZE)+1;
  numItems:=numItems-1;
  resume(producersC);
end;

begin {initiation code}
insertIndex:=1;
removeIndex:=1;
numItems:=0;
end;
```

### 3.3.3 Producer-Consumer problem

Now write the code necessary to run the solution with 2 consumers and 2 producers

#### ACTIVE PROCESSES

```
process type tProducer;  
var  
    item: integer;  
begin  
repeat  
    item := random(200);  
    buffer.insert(item);  
forever  
end;  
  
process type tConsumer;  
var  
    item: integer;  
begin  
repeat  
    buffer.extract(item);  
forever  
end;
```

- In Parallel FC, it depends  
the number of queues  
(Cond. variables are queues)

```
var  
{*the buffer monitor is a global variable  
we do not need to declare it*}  
consumer1, consumer2:tProducer;  
producer1,producer2: tConsumer;  
begin  
cobegin  
consumer1;  
consumer2;  
producer1;  
producer2;  
coend  
end.
```

Always > 2 queues

**3.1 Semaphores**

**3.2 CCR**

**3.3 Monitors**

**3.3.1 Introduction**

**3.3.2 Monitors**

**-Mutual Exclusion**

**-Condition Synchronization**

**3.3.3 Producer-Consumer problem**

**3.3.4 Monitors in Java**

### 3.3.4 Monitors in Java

- Java language does not provide monitors as built-in objects.
- Think of a class with a set of **private variables, and several public synchronized methods** which operate on that variable:
  - that is a monitor which guarantees mutual exclusion operations on such variables.
  - Why?
- But, what about condition synchronization?
  - You can use signals *wait* and *notify* to block processes but...
  - Can you see a problem here?

### 3.3.4 Monitors in Java

- By default, signals block and resume threads on the same queue.
- If we want to imitate the behaviour of **condition variables**, we can call these **signals on different Object objects**.

Thread 1   Thread 2   Thread 3

```
synchronized public void queueMe() throws Exception{  
    wait();  
}
```

Thread 1   Thread 2   Thread 3

```
public void queueMe(Object ob) throws Exception{  
    //imagine each thread passes a different Object instance  
    synchronized(ob){ob.wait();}  
}
```



- Since we are using more than 1 condition variable, the synchronization must be done per block instead of the whole method. And then each block is synchronized on the Object on which a wait or signal may be called.
- But now there is not a general mutual exclusion... you'd better use the *ReentrantLock* object which provides condition queues.

# Potential Mid-term Exam Questions

1. What is the structure of a monitor which provides condition synchronization?
2. What can you say about a Java class with a private variable and a set of public and synchronized methods?
3. What do we mean by ‘semantic’ of the resume operation on blocked processes?
4. How many queues do we have in a monitor in Pascal-FC?

# Keywords phonetics

- active /'æktɪv/ 
- disambiguation /dɪsæmbɪgjʊ'eɪʃən/ 
- resume /rɪ'zju:m/ 
- courtesy /'kɔ:təsi/ 
- private /'praɪvət/ 



# 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**

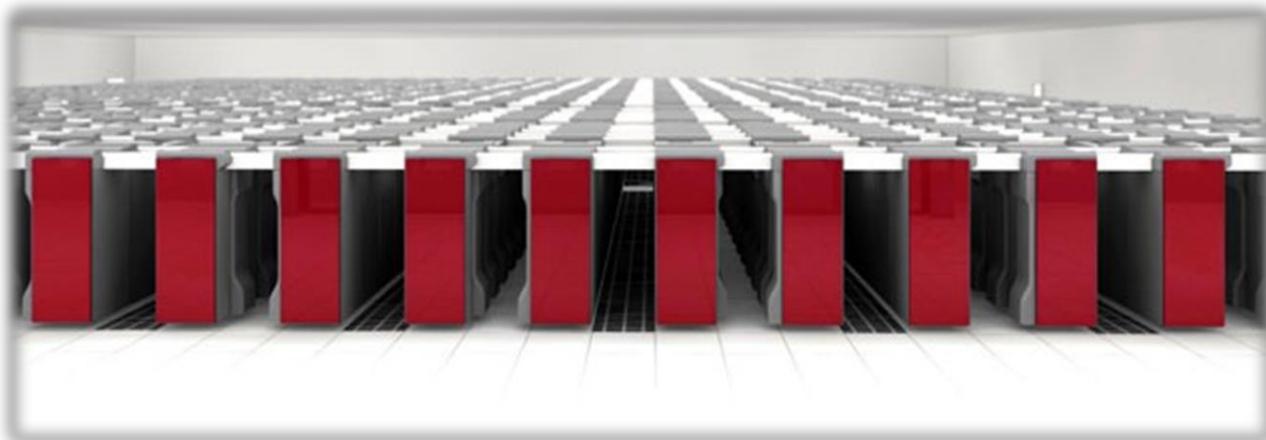
**4.3.3 Guarded selective waiting**

**4.3.4 Selective waiting – terminate**

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

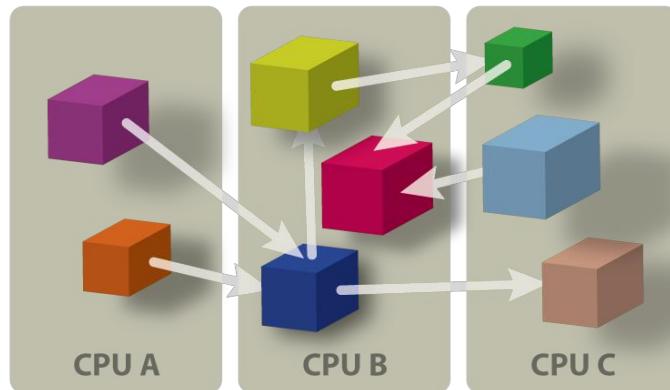
# 4.1 Introduction

- 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

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



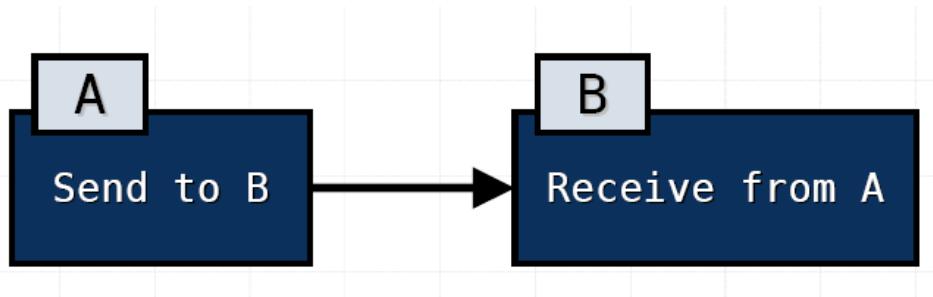
- A taxonomy of the communication models can be depicted attending to 3 different aspects:
  - Addressing method
  - Synchronization method
  - Channel characteristics

# 4.2 Message passing models

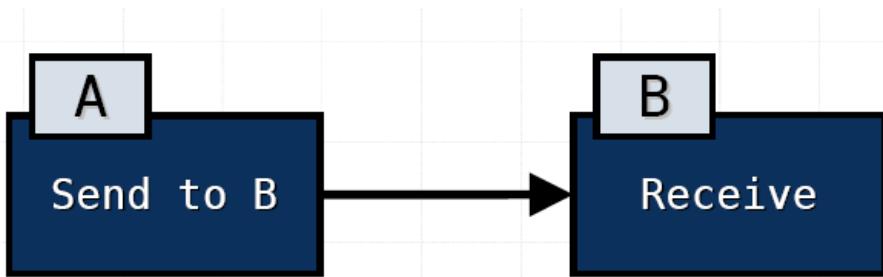
Depending on the **addressing methods**, message passing can be:

- **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.

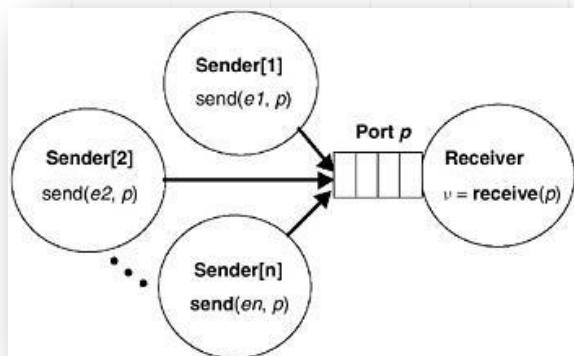
## 4.2 Message passing models



Direct symmetric communication



Direct asymmetric communication



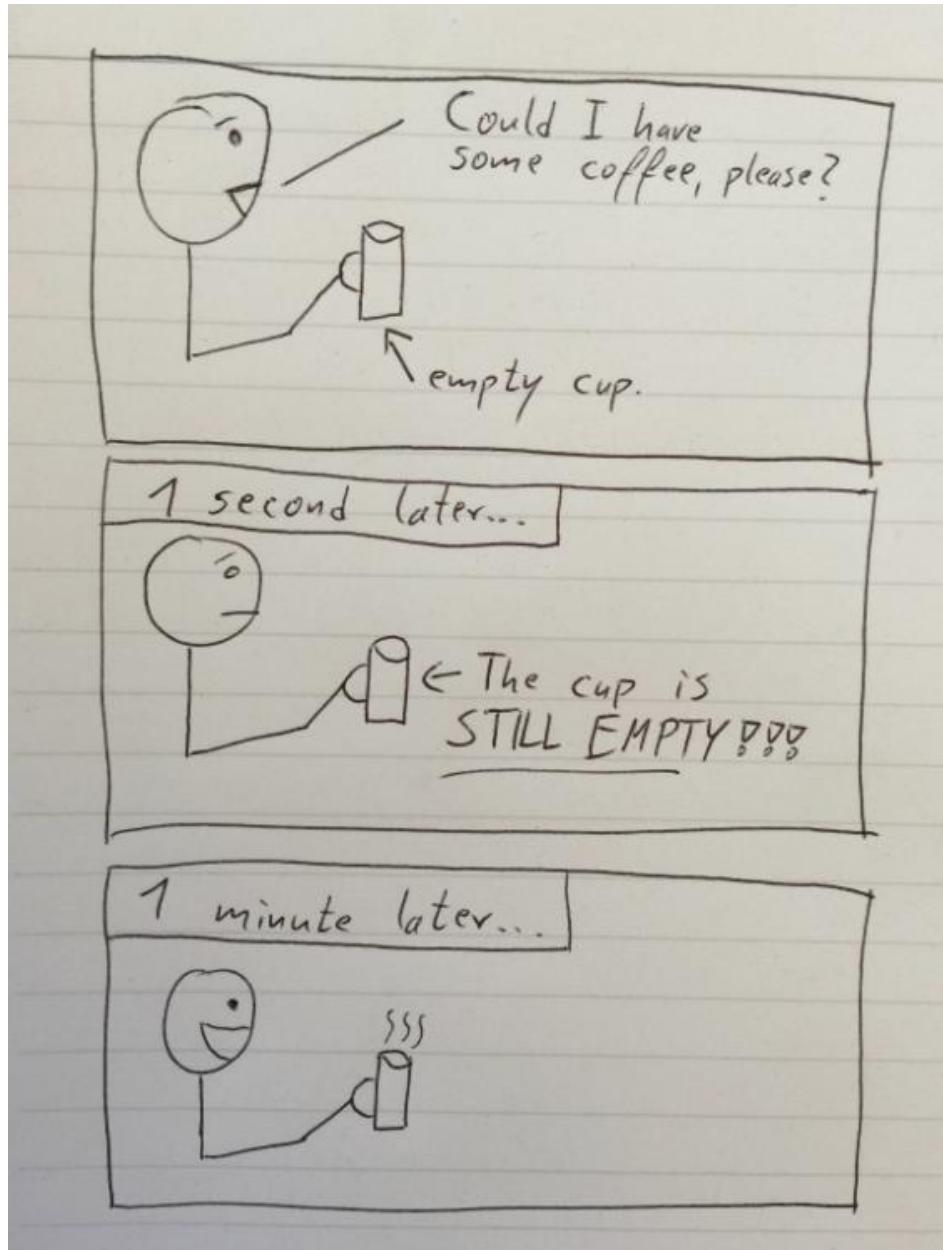
Indirect communication

# 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?*

*So synchronous communication is great for the sake of synchronization itself,  
but the in-practice problem is...*



stackoverflow.com

*but if you do not act synchronized ...*



*Missed*

*or*

*buffered*



# 4.2 Message passing models

Models provided by some **programming languages**:

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

Depending on the **channel characteristics**:

- **Data flow**: the sense in which data is sent. Unidirectional (e-mail; Pascal-FC) 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? Pascal-FC

**4.1 Introduction**

**4.2 Message passing models**

**4.3 Synchronous communication**

**4.3.1 Intro to synch. comm.**

**4.3.2 Selective waiting**

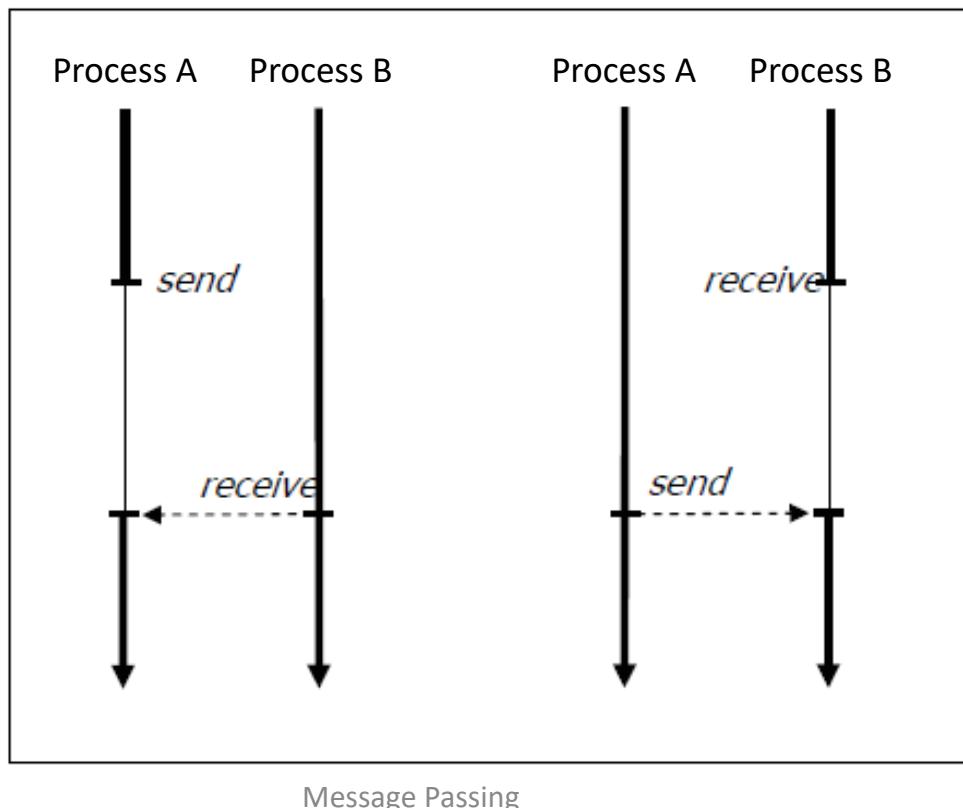
**4.3.3 Guarded selective waiting**

**4.3.4 Selective waiting – terminate**

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

## 4.3.1 Intro to Synch. Comm.

- Pascal-FC provides synchronous communication. 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 processes may want to send messages (each one through 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;           no message
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 to communicate each producer with the buffer, and the same 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; → kill process
  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.

both  
conditions  
necessary

## 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;
```



not going  
to happen

```
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.
```

*pass by reference*

*same output than previous one*

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, priority

```
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 consumer had send signal?
      chSyn ? any;
      stop:=true;
    else
      ch1 ! i; {continues sending
      i:=i+1;
    end;
  writeln('producer finished.');
end;
```

```
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
```

# Dining Philosophers

- There is one kind of process: philosopher. And:
- DATA IS NOT SHARED. So a manager process needs to be designed to control the state of the resource of interest. In this case, chopsticks.

# HOW PHILOSOPHERS USE THE MANAGER

```

program diningPhilosophers;

const PHI=4; (*philosophers - 1,
because next pos to last pos is
0*)

var takeChopsticks:
array[0..PHI] of channel of
synchronous;
var releaseChopsticks:
array[0..PHI] of channel of
synchronous;
var mutex: semaphore;

process type tPhilosopher(ID:
integer);
begin
repeat
wait(mutex);
writeln(ID, ': is thinking... ');
signal(mutex);
sleep(1);

wait(mutex);
writeln(ID, ' is hungry... ');
signal(mutex);
takeChopsticks[ID] ! any;

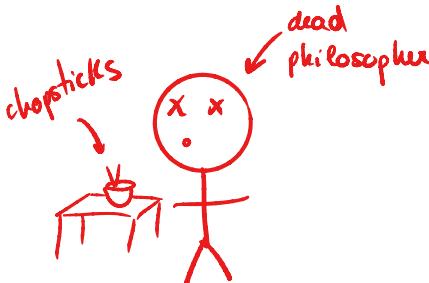
```

```

wait(mutex);
writeln(ID, ' is eating... ');
signal(mutex);

releaseChopsticks[ID] ! any;
wait(mutex);
writeln(ID, ' has finished
eating. ');
signal(mutex);
forever
end;

```



condition: both chopsticks must be free in order to take them and eat

organize how chopsticks can be used

```
process Manager;
```

...

...

```

var
pPhilosophers:
array[0..PHI] of
tPhilosopher;
i: integer;

begin
initial(mutex,1);
cobegin
for i:=0 to PHI do
pPhilosophers[i](i)
;
Manager;
coend
end.

```

## IMPORTANT

```
process Manager; } both chopsticks  
var are free  
chopsticks: array[0..PHI] of boolean; (* true means free *)  
i: integer;  
begin  
for i:=0 to PHI do  
chopsticks[i]:= true;  
repeat  
select  
for i:=0 to PHI replicate  
when  
(chopsticks[i] AND  
chopsticks[(i+1) mod PHI])  
=>then  
takeChopsticks[i] ? any; ✓  
chopsticks[i]:=false;  
chopsticks[(i+1) mod PHI]:=false;  
or  
for i:=0 to PHI replicate  
releaseChopsticks[i] ? any;  
chopsticks[i]:=true;  
chopsticks[(i+1) mod PHI]:=true;  
or  
terminate;  
end;  
forever  
end;
```

# Potential Mid-term Exam Questions

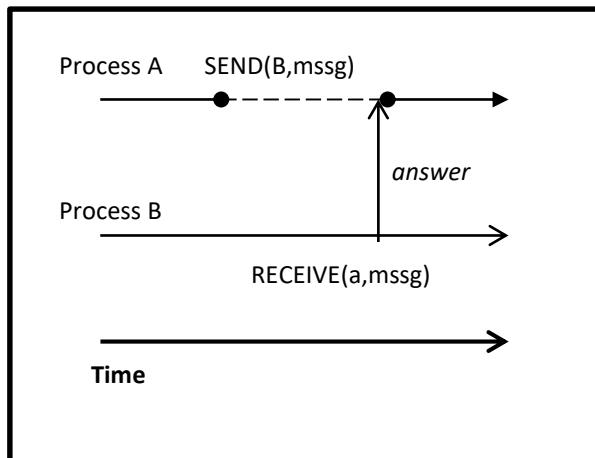
1. What is the difference between the **else** and **terminate** alternatives?

- **terminate**: kills process  → cannot be guarded
- **else**: tells the process to do another thing

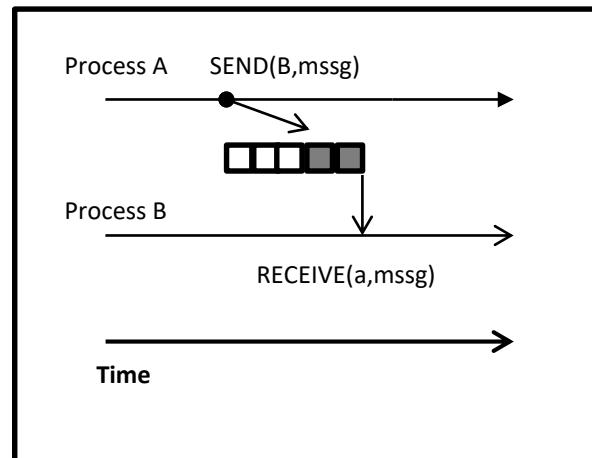
2. When can a process be blocked inside a **select** which uses an **else** alternative?

NEVER, because we have else to go out the select

3. Which kind of communication do you think these schemes represent? (from the point of view of synchronization)



direct synchronization



asynchronous  
communication

① select and else  
if all previous channels are empty I can use  
the else

terminate

- all channels are empty
- all active processes are dead  
or blocked in a channel

# Keywords phonetics

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

# 2nd problem June 2015

(Written and solved by *Miguel Ángel Galdón Romero* )

## **Sleeping Barber (Message passing):**

There is a barbershop with a waiting room which has 5 chairs. Up to 10 clients enter to be shaved by the barber in his armchair.

The barber does nothing when there is no client to shave. It takes him 5 seconds to shave a client.

When a client comes in:

If there is a free chair, he/she sits down and wait for the barber to be free.

Else: waits for a free chair outside .

    The client leaves after being shaved.

    When there are no clients, all processes are finished.

    Clients might enter while the barber is shaving.

**[4 points]** Solve this problem

**[1 point]** Make clients show the number of free chairs at every moment.

# 2nd problem June 2015

OUTPUT EXAMPLE:

[5] sits down. Free chairs: 4  
[2] sits down. Free chairs: 3  
    [5] is with barber. Free chairs: 4  
[3] sits down. Free chairs: 3  
[9] sits down. Free chairs: 2  
[8] sits down. Free chairs: 1  
[4] sits down. Free chairs: 0  
    [5] leaves  
    [9] is with barber. Free chairs: 1  
[10] sits down. Free chairs: 0  
    [9] leaves  
    [4] is with barber. Free chairs: 1  
[1] sits down. Free chairs: 0  
    [4] leaves  
    [3] is with barber. Free chairs: 1

[7] sits down. Free chairs: 0  
[3] leaves  
    [1] is with barber. Free chairs: 1  
[6] sits down. Free chairs: 0  
    [1] leaves  
    [2] is with barber. Free chairs: 1  
    [2] leaves  
    [8] is with barber. Free chairs: 2  
    [8] leaves  
    [6] is with barber. Free chairs: 3  
    [6] leaves  
    [10] is with barber. Free chairs: 4  
    [10] leaves  
    [7] is with barber. Free chairs: 5  
    [7] leaves

# 2nd problem Final Junio 2015

- There are several rendezvous:
  - Client to take chair
  - Client and barber
  - Barber finishes shaving
- There are 2 kinds of processes (client and barber). And:
- the manager process which will handle the resources of interest, in this case the number of free chairs.

```

program sleepingBarber;
const NCLI = 10;

type intchan = channel of integer;
synchan = channel of synchronous;

var CgetChair,CputChair : array [1..NCLI] of
intchan;
  CfreeBarber : array [1..NCLI] of synchan;
  CjobOver: array[1..NCLI] of synchan;

process type Tclient(id : integer);
var nchairs : integer;
begin
  sleep(random(3));
  CgetChair[id] ? nChairs; {request chair
rendezvous}
  writeln('[' ,id, '] sits down. Free chairs:
',nChairs);
  CfreeBarber[id] ? any; {next client rendezvous}
  CputChair[id] ? nChairs; {release chair
rendezvous}
  writeln('[' ,id, '] is with barber. Free chairs:
',nChairs);

  CjobOver[id] ? any; {wait until barber finishes
rendezvous}
  writeln('[' ,id, '] leaves');

end;

```

```

process Barber;
var i : integer;
begin
repeat
select
  for i := 1 to NCLI replicate
    CfreeBarber[i] ! any; {next client
rendezvous}
    sleep(5);{shaving}
    CjobOver[i] ! any; {wait until barber
finishes rendezvous}
  or terminate;
end;{select}
forever
end;

```

Manager process....

```

process BarberShop; (*manager process*)
var nChairs,i : integer;

begin
nChairs :=5;
repeat
select
  for i := 1 to NCLI replicate
    when (nChairs > 0) =>
      Cgetchair[i] ! nChairs-1; {request chair
rendezvous}
      nChairs := nChairs-1;
    or
    for i := 1 to NCLI replicate
      Cputchair[i] ! nChairs +1; {release chair
rendezvous}
      nChairs := nChairs+1;
    or
    terminate;
end;{select}

forever;
end;

```

```

var client : array [1..NCLI] of Tclient;
c : integer;

begin
cobegin
  for c := 1 to NCLI do
    client[c] (c);
    BarberShop;
    Barber;
  coend;
end.

```