Advanced Programming II

Lesson 3: Concurrent Programming viewed as an Abstraction

Grado en Ingeniería Informática

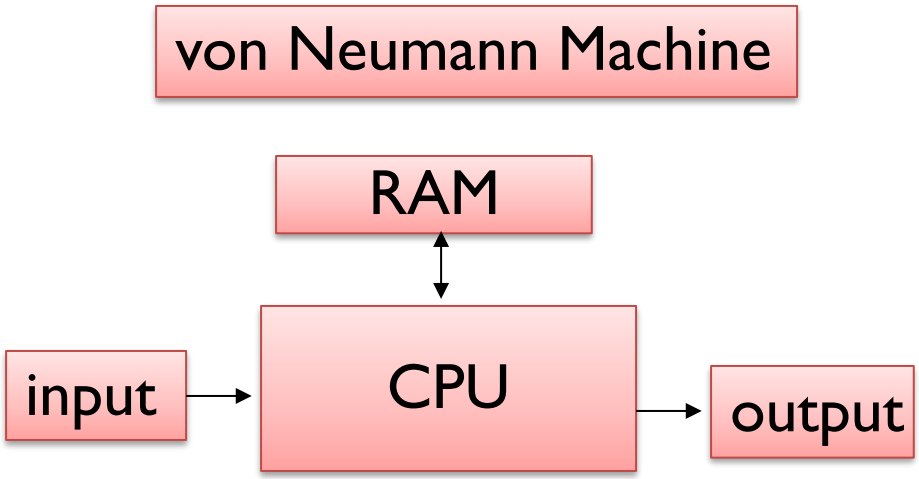
Grado en Ingeniería del Software

Contents

- From Sequential programming to Concurrent programming
- Advantages and usages of Concurrent programming
- Inherent problems in Concurrent programming:
  - Atomic sentences
  - Critical Sections
  - Mutual Exclusion
- Platforms for concurrent execution

Sequential Programming

- To take the most of the computer's power by means of Sequential Programming languages, a well-suited program must be explicitly created
- For each clock cycle:
  - An instruction is fetched from memory
  - It is decodified and executed by sending several electronic signals to the corresponding system components




```
{
    x = 0;
    y = 0;
    …
}
```

```
{
    x = …
    y = …
    if (x < y) z = x;
    else z = y;
    …
}
```

Formal Example

- If $P = p_1;p_2;\ldots;p_n$ is a sequential program
  - p_1 executes always before p_2 (in any execution),
  - P_2 executes always before p_3

  ….
  - p_{n-1} executes always before p_n

- Stated that the symbol "$\rightarrow$" means "precedes to" and $\forall e$ means "for any execution", the sequential behaviour may be formalized as $\forall e.(p_1 \rightarrow p_2) \wedge (p_2 \rightarrow p_3) \wedge \ldots \wedge (p_{n-1} \rightarrow p_n)$

- This behaviour is always the same, even if the code P includes conditional or loop sentences.

if (b) {A;} else {B;}

- $\forall e.(b \rightarrow A) \vee (b \rightarrow B)$ (depends on input data for each execution)

Formal Example

while (b) {A;}

$\forall e.b \vee (b \rightarrow A \rightarrow b) \vee (b \rightarrow A \rightarrow b \rightarrow A \rightarrow b) \vee \ldots$

(The number of times **b** is executed depends on input)

Let's say P = $p_1;p_2;\ldots;p_n$

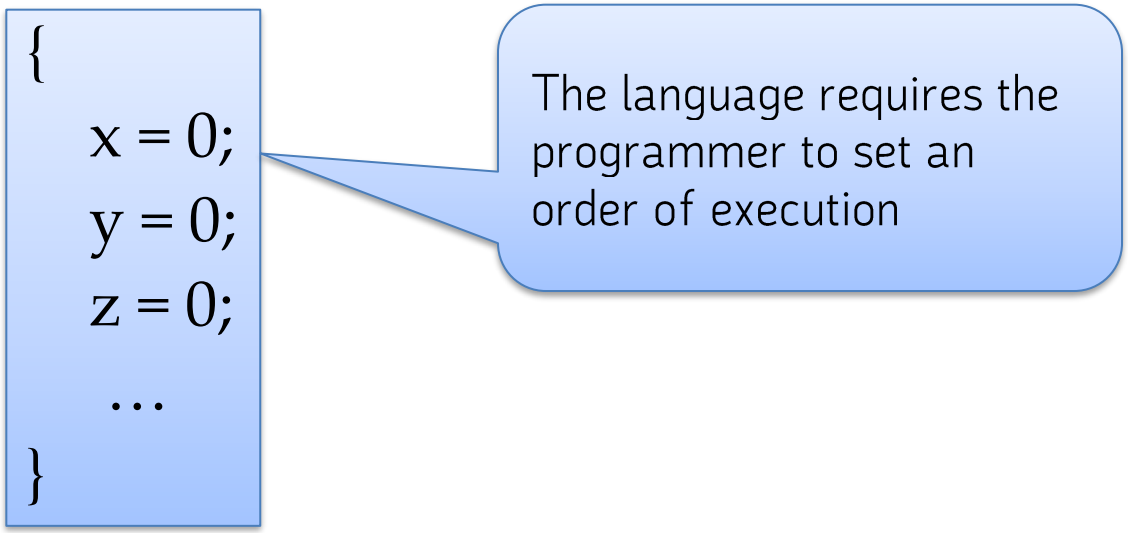
assuming $0<i<j\leq n$ then $\forall e.p_i \rightarrow p_j$

The program is deterministic. Given a program and an input data set, it can be predicted which is the next sentence to execute

Sentences are strictly ordered

Concurrent Programming

- Not all the sentences of a program **have to** be executed sequentially.



Concurrent Programming

- Not all the sentences of a program have to be executed sequentially
  - This code can be executed following **6** different orders, and any of them is considered as valid

```
{
    x = 0;
    y = 0;
    z = 0;
    …
}
```

```
{
    x = 0;
    z = 0;
    y = 0;
    …
}
```

```
{
    y = 0;
    x = 0;
    z = 0;
    …
}
```

…….

Concurrent Programming

- Not all the sentences of a program have to be executed sequentially
- This code can be executed following 6 different orders, and any of them is considered valid
- Assume we define the **new operator ||** to express formally this behaviour

$$
\begin{aligned}
x = 0 \,||\, y = 0 \,||\, z = 0 \quad = \{ & x = 0 \rightarrow y = 0 \rightarrow z = 0, \\
& x = 0 \rightarrow z = 0 \rightarrow y = 0, \\
& y = 0 \rightarrow x = 0 \rightarrow z = 0, \\
& \ldots \\
& \}
\end{aligned}
$$

Concurrent Programming

- Assume we define the new operator || to express formally this behaviour.

But, what happens if we have 3 processors?

- Each block of code (each sentence) can be assigned to a different processor, achieving a possibly higher throughput and a valid execution.
- The meaning of **P || Q is extended** to express that there is a valid execution of P and Q where both blocks of code overlap in time:

$$\exists\ e.\ \neg\ (P \rightarrow Q) \wedge \neg\ (Q \rightarrow P)$$

Concurrent Programming

- How to understand well the meaning of an execution like **P || Q**?
  - We may require two real processors in order to have a true time overlapping processing of the code
    - It is not a good idea because, in such case, the behaviour of the code depends on the hardware architecture
  - Instead of this, we will assume that there are a couple of **virtual processors**, and each of them executes a block of code

    In the real world, we could assign a real processor to each virtual one, but what happens if there are not enough real processors?

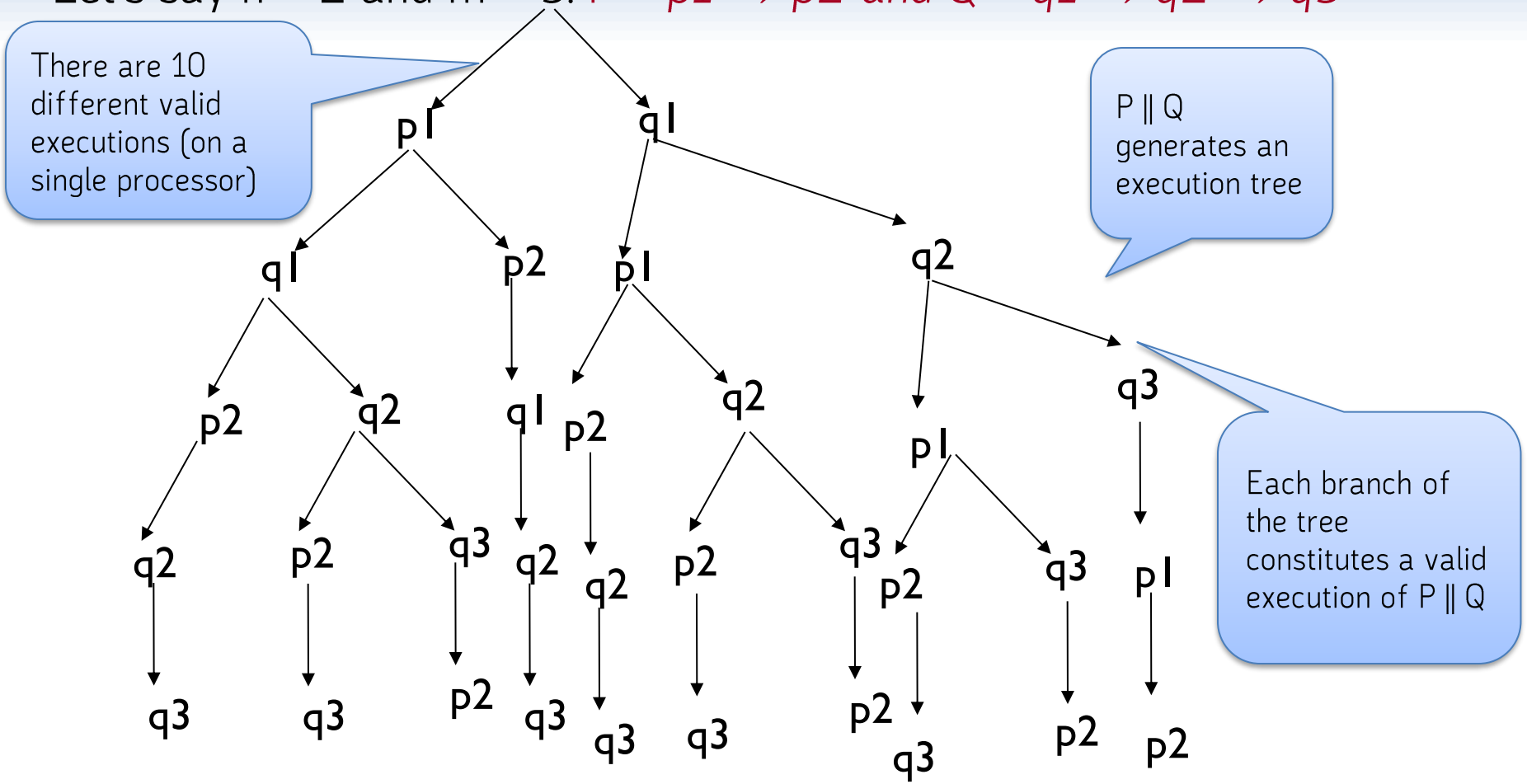
Let's say $P = p_1 \rightarrow p_2 \rightarrow \ldots \rightarrow p_n$ and

$Q = q_1 \rightarrow q_2 \rightarrow \ldots \rightarrow q_m$

How can be interpreted **P || Q** when there is only a single real processor?

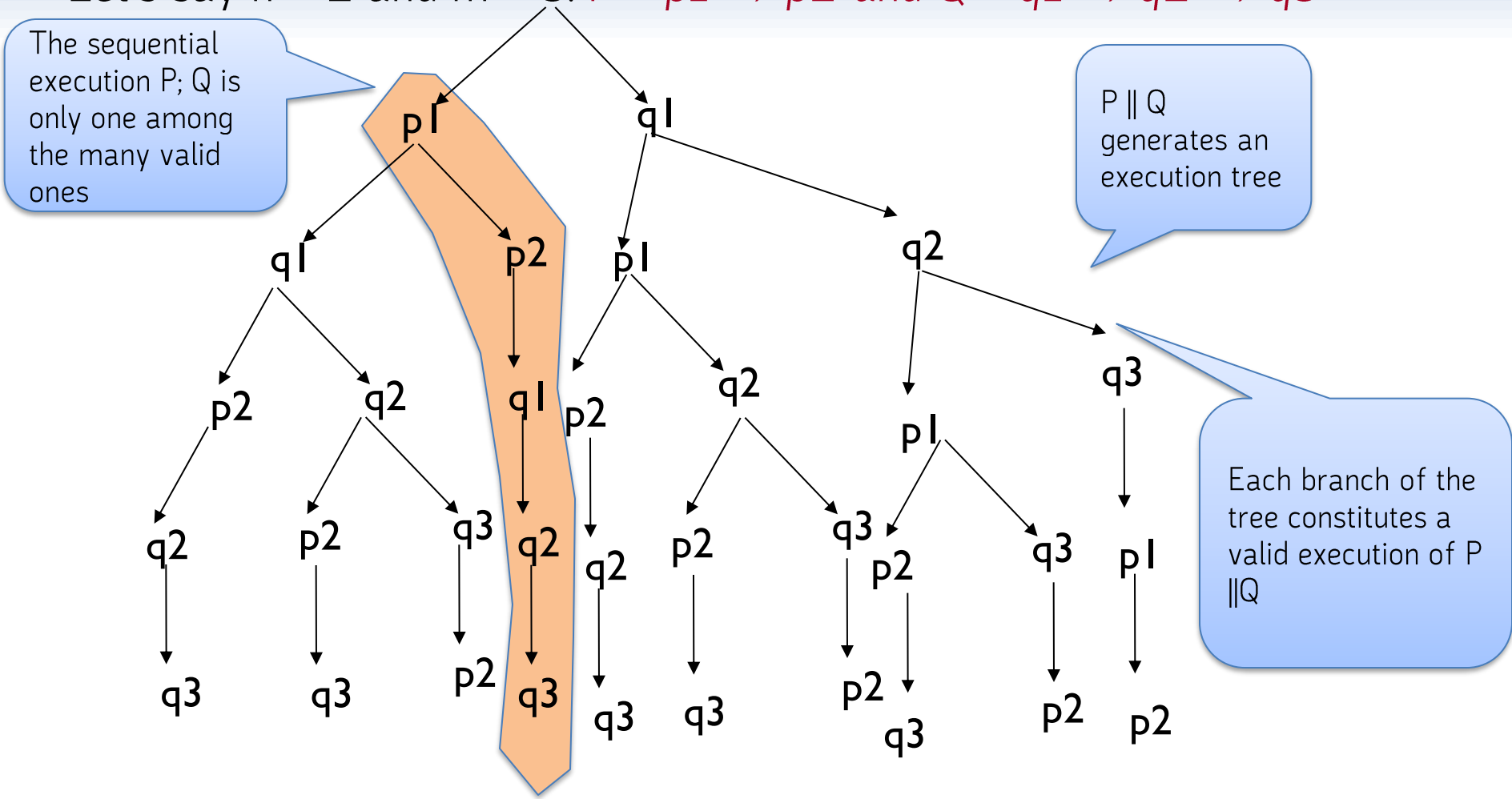
Interleaving

- Let's say n = 2 and m = 3: $P = p1 \rightarrow p2$ and $Q = q1 \rightarrow q2 \rightarrow q3$



Interleaving

- Let's say n = 2 and m = 3: $P = p1 \rightarrow p2$ and $Q = q1 \rightarrow q2 \rightarrow q3$



Partial Order

- A valid execution of P || Q
  ($P = p1 \rightarrow p2 \rightarrow \ldots \rightarrow pn$, $Q = q1 \rightarrow q2 \rightarrow \ldots \rightarrow qm$)
  is obtained by interspersing P and Q sentences,
  but not all merges are valid

- The internal order of the sentences in P and Q must remain,

  Given two indexes i, j,

  - *if* $i < j$ *then* $\forall e.\ (pi \rightarrow pj) \wedge (qi \rightarrow qj)$
  - *if* $i < j$ *then* $\forall e.\ (pi \rightarrow qj) \vee (qj \rightarrow pi)$

The order inside P and Q is kept up

But there is no need of any order among the sentences of P and Q

Partial Order

- A valid execution of P || Q
  ($P = p1 \rightarrow p2 \rightarrow \ldots \rightarrow pn$, $Q = q1 \rightarrow q2 \rightarrow \ldots \rightarrow qm$)
  is obtained by interspersing P and Q sentences,
  but not all merges are valid

- The internal order of the sentences in P and Q must remain,
  Given two indexes i, j,
  - *if* $i < j$ *then* $\forall e.\ (pi \rightarrow pj) \wedge (qi \rightarrow qj)$
  - *if* $i < j$ *then* $\forall e.\ (pi \rightarrow qj) \vee (qj \rightarrow pi)$

At any time, different sentences are available to be executed and one of then is randomly selected. Therefore, a parallel execution is always NON DETERMINISTIC

The sentences of P||Q are **partially ordered**

Concurrent Programming

- How to understand well the meaning of an execution like P || Q?
  - We may require two real processors in order to have a true time overlapping processing of the code
    - It is not a good idea because, in such a case, the behaviour of the code depends on the hardware architecture
  - Instead of this, we will assume that here are a couple of **virtual processors**, and each of them executes a block of code
  - A virtual processor may correspond with a real one, but
    what happens if there are not enough real processors?
  - The sentences of P and Q are interspersed in a processor
    - (semantics of interleaving)
- This may be seen as if the virtual processors executing P and Q are working with different speeds, so
  - We can not take for granted anything about the relative speed of virtual processors, or about the exact time of execution taken by any block of code

Concurrent Programming

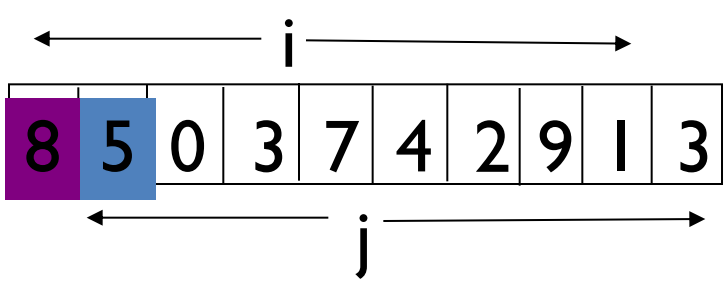
Summary,

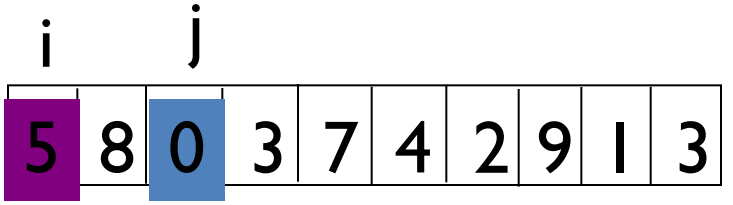
- P || Q stands for a concurrent execution of P and Q
- P and Q are named processes/threads because they are executed concurrently with other processes
- Nonetheless, the code of P and Q are executed sequentially
- If P = $p1 \rightarrow p2 \rightarrow \ldots \rightarrow pn$ and Q = $q1 \rightarrow q2 \rightarrow \ldots \rightarrow qm$, then the total number of valid executions of **P||Q** (assuming no overlapping in time when two sentences are executed) is

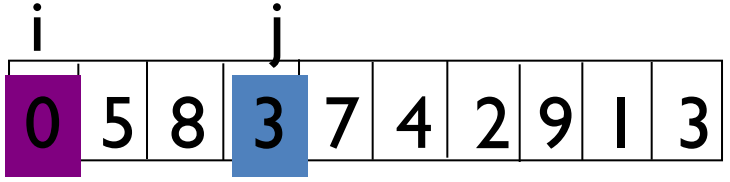
  (m+n)!/m!*n!
- Each valid execution is named trace (a branch of the tree)
- A program **P||Q** is **correct** when all its traces are correct

Advantages

- Improve the throughput of processors
- Exploit multiprocessor architectures
- Make easier to model systems where the coherence is inherent
- Complete a task quickly
  - Example: Selection Sort







Example (Selection Sort)

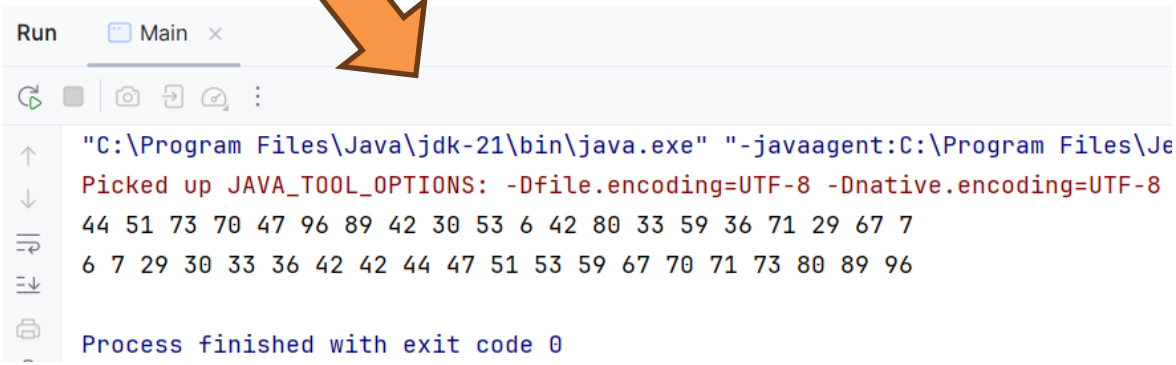
Implementing the Selection sort algorithm

```
def selectionSort(lista: Array[Int]) =
    for (i<-0 until lista.length)
        for (j<-i+1 until lista.length)
            if (lista(i) > lista(j))
                val aux = lista(i)
                lista(i) = lista(j)
                lista(j) = aux
```

Example (Selection Sort). Option 1

Sequential case: ordering an array by calling the previous algorithm

```
import scala.util.Random
object Main extends App:
    val a = new Array[Int](20)
    for (i <- 0 until a.length)
        a(i) = Random.nextInt(100)
    println(a.mkString(" "))
    selectionSort(a)
    println(a.mkString(" "))
```



Example (Selection Sort). Option 2

Use the divide and conquer technique: the array is divided into two halves. Then, we call the algorithm twice to sort each half. Finally, we need a method that merges the two halves already ordered into the definitive sorted array.

```
def mergeIt(l1: Array[Int],l2: Array[Int],lista: Array[Int]) =
   var i=0
   var j=0
   var k=0
   while i<l1.length && j<l2.length do
      if (l1(i)<=l2(j)) then
         lista(k) = l1(i); i += 1
      else
         lista(k) = l2(j); j += 1
      k += 1
   while i < l1.length do
      lista(k) = l1(i); i += 1; k += 1
   while j < l2.length do
      lista(k) = l2(j); j += 1; k += 1
```

Example (Selection Sort). Option 2

Use the divide and conquer technique: the array is divided into two halves. Then, we call the algorithm twice to sort each half. Finally, we need a method that merges the two halves already ordered into the definitive sorted array.

```
import scala.util.Random
object Main extends App:
   val a = new Array[Int](20)
   for (i <- 0 until a.length)
      a(i) = Random.nextInt(100)
   println(a.mkString(" "))
   val (l1, l2) = a.splitAt(a.length / 2)

   selectionSort(l1)
   selectionSort(l2)
   mergeIt(l1, l2, a)

   println(a.mkString(" "))
```

```
"C:\Program Files\Java\jdk-21\bin\java.exe" "-javaagent:C:\I
Picked up JAVA_TOOL_OPTIONS: -Dfile.encoding=UTF-8 -Dnative
48 68 66 81 9 95 55 39 63 87 95 49 13 82 62 24 92 68 15 38
9 13 15 24 38 39 48 49 55 62 63 66 68 68 81 82 87 92 95 95

Process finished with exit code 0
```

Example (Selection Sort). Option 3

With parallelism: using threads + divide and conquer. Two threads are used simultaneously to sort the halves at the same time. Finally, both halves are merged.

```
import scala.util.Random
object Main extends App:
   val a = new Array[Int](20)
   for (i <- 0 until a.length)
      a(i) = Random.nextInt(100)
   println(a.mkString(" "))
   val (a1, a2) = a.splitAt(a.length / 2)
   val h1 = new MyThread(a1)
   val h2 = new MyThread(a2)
   h1.start();
   h2.start()
   h1.join();
   h2.join()

   mergeIt(a1, a2, a)
   println(a.mkString(" "))
```

```
"C:\Program Files\Java\jdk-21\bin\java.exe" "-javaagent:C:\
Picked up JAVA_TOOL_OPTIONS: -Dfile.encoding=UTF-8 -Dnative
42 58 81 77 44 19 73 94 87 35 73 25 71 59 72 12 10 8 16 61
8 10 12 16 19 25 35 42 44 58 59 61 71 72 73 73 77 81 87 94

Process finished with exit code 0
```

Example

- Studying complexities

```
def selectionSort(lista: Array[Int]) =
    for (i<-0 until lista.length)
        for (j<-i+1 until lista.length)
            if (lista(i) > lista(j))
                val aux = lista(i)
                lista(i) = lista(j)
                lista(j) = aux
```

If ***n*** is the number of items in the vector to sort

$$T(n) = \sum_{i=0}^{n-1} \sum_{j=i+1}^{n} 1 = \sum_{i=0}^{n-1} (n-i) = \frac{n(n+1)}{2} \approx \frac{n^2}{2}$$

Example

If *n* is the number of items in the array to sort

Case sequential: $T(n) \sim n^2/2$

Case sequential with two invocations: $T(n) \sim n^2/4 + n$

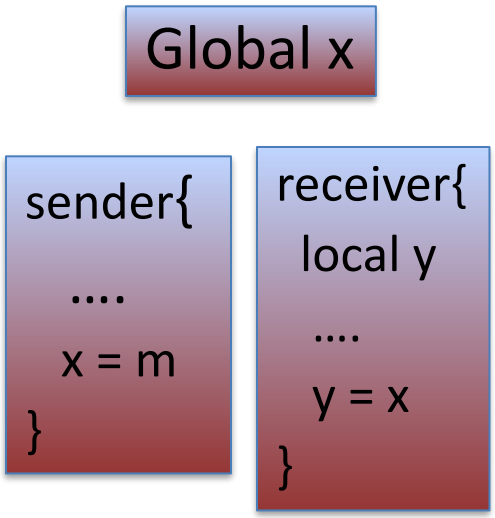
Case parallel with two invocations $T(n) \sim n^2/8 + n$

n \ T(n)	$n^2/2$	$n^2/4+n$	$n^2/8+n$
20	200	120	60
40	800	440	220
1000	500000	256000	125500

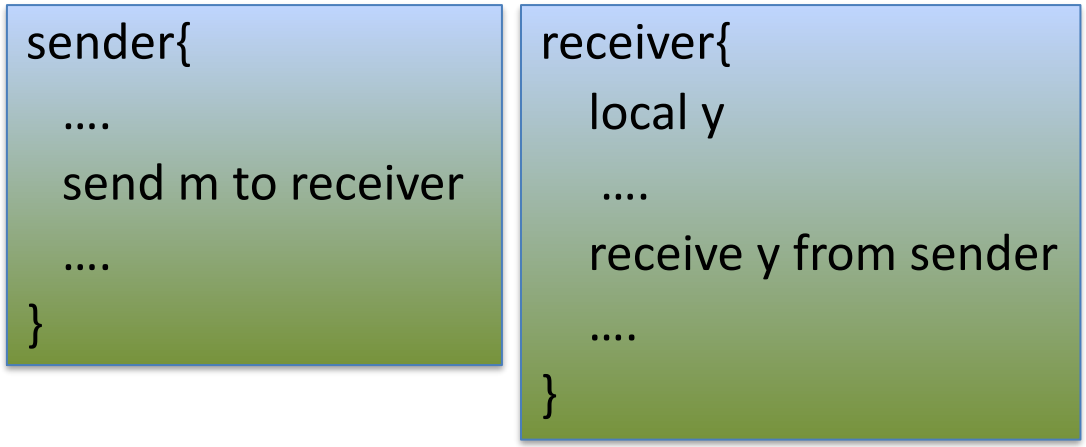
Communication and Synchronization

The processes involved in a concurrent program usually communicate among them and synchronize their actions

- Communication may be performed by means of shared memory



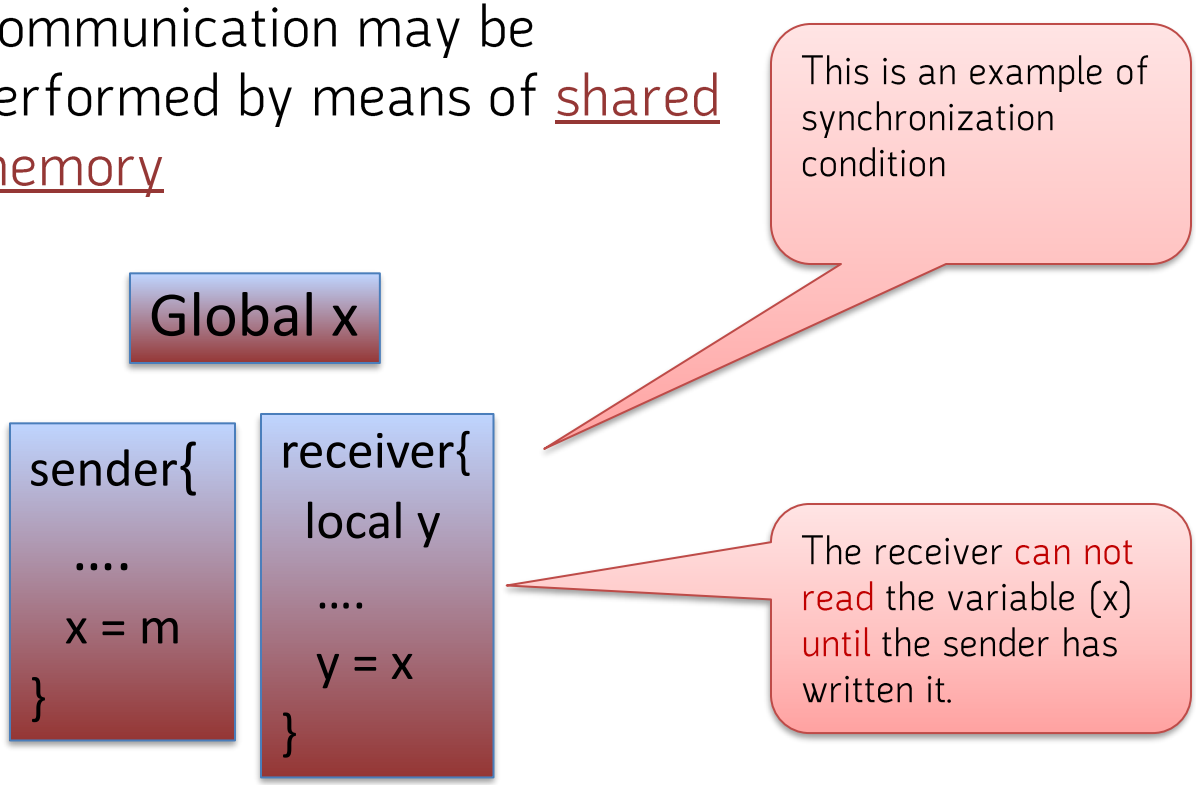
- ... or by some way of message passing



Communication and Synchronization

The processes involved in a concurrent program usually communicate among them and synchronize their actions

- Communication may be performed by means of shared memory

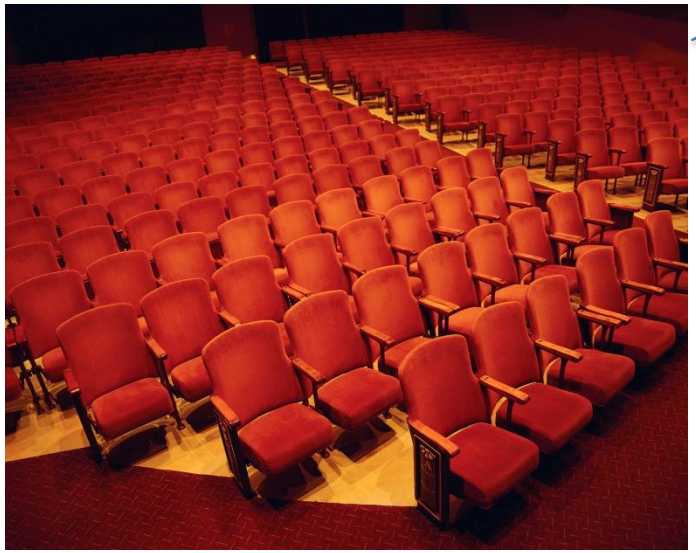


Problems with Concurrent Programming

Atomic sentences

- Critical section
- Mutual exclusion

- Example: Tickets vending machine



Tickets can be bought at the same time in different ticket offices

Problems with Concurrent Programming

- Each ticket office is a process T_i
- All the ticket offices work concurrently
- All the ticket offices execute the same code
- The seats states of the theatre are stored into an array of booleans (taken or not)

`seats= new Array[Boolean](200)`

T1 || T2 || ...|| Tn

Code of T1

```
while (true){
  Show untaken seats to the user
  a = ... // selected seat
  seats[a] = true
  print the ticket
}
```

....

Code of Tn

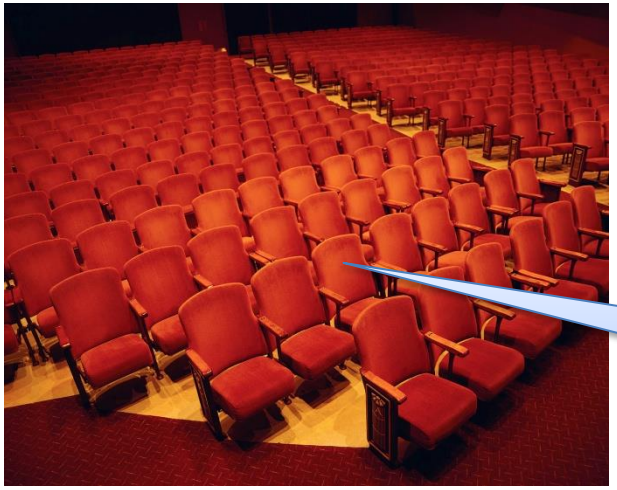
```
while (true){
  Show untaken seats to the user
  a = ... // selected seat
  seats[a] = true
  print the ticket
}
```

Example: Tickets Vending Machine

- Such a solution is incorrect. May happen two customers buy tickets for the same seat

Ti: show untaken seats to the user
Tj: show untaken seats to the user
Ti: a = ...; // selected seat in office i, for example a = 22
Tj: a = ...; // selected seat in office j, for example a = 22
Ti: seats[22] = true
Tj: seats[22] = true
Ti: print the ticket
Tj: print the ticket

Interleaving: This is an execution trace, i.e. a branch of the tree which behaves incorrectly. A trace like this is useful to show that a program is **incorrect**



The seat nº 22 has been assigned to two different customers

Code of each office

```
while (true){
   Show untaken seats to the user
   a = ... // selected seat
   seats[a] = true
   print the ticket
}
```

Example: Tickets Vending Machine

We try to debug the code

Code of Ti

```
while (true){
  success= false;

  while (!success){
     Show untaken seats to the user
     a=…; // selected seat
     if (! seats[a]) {
              seats[a] = true ;
              print the ticket
              success= true;
     } else {
              display an error message
     }
   }
}
```

We include this code to be sure of a seat is actually untaken when selected

Example: Tickets Vending Machine

Code of Ti

```
while (true){
  success= false;

  while (!success){
    Show untaken seats to the user
    a=...; // selected seat
    if (! seats[a]) {
            seats[a] = true ;
            print the ticket
            success= true;
    } else {
            display an error message
    }
  }
}
```

Execution TRACE which shows the error

Ti: check seats[22] as untaken
Tj: check seats[22] as untaken
Ti: seats [22] = true
Tj: seats [22] = true
Ti: print the ticket
Tj: print the ticket

We have the same problem as before, but now we have found out the lines of code where the problem is

Example: Tickets Vending Machine

- **Atomic sentences**: Those that are executed with **no interruption**
- Only machine instructions are really atomic

```
while (true){
  success= false;

  while (!success){
     Show untaken seats to the user
     a=...; // selected seat
     if (! seats[a]) {
              seats[a] = true ;
              print the ticket
              success= true;
     } else {
              display an error message
     }
   }
}
```

```
load seats[a] to CPU register
test the value of CPU register
jump to L1 if true
set seats [a] to true
code for "print the ticket"
set success to true
L1: code for "error message"
L2: code following if statement
```

```
>
load seats[a] to CPU register
test the value of CPU register
jump to L1 if true
set seats[a] to true
>
code for "print the ticket"
set success to true
L1: code for "error message"
L2: code following if statement
```

What would happen if the highlighted code were atomic?

Example: Tickets Vending Machine

- Atomic sentences: Those executed with no interruption

Only machine instructions are really atomic

If the highlighted code were atomic, then an erroneous trace could not happen

>
load seats[a] to CPU register
test the value of CPU register
jump to L1 if true
set seats [a] to true
>
code for "print the ticket"
set success to true
L1: code for "error message"
L2: code following if statement

Execution TRACE which shows the error

Ti: check seats[22] as untaken
Tj: check seats[22] as untaken
Ti: seats [22] = true
Tj: seats [22] = true
Ti: print the ticket
Tj: print the ticket

Example: Tickets Vending Machine

- Atomic sentences: Those executed with no interruption

  Only machine instructions are really atomic

Valid interleavings are only those which happens **BEFORE OR AFTER** Ti has executed the highlighted code

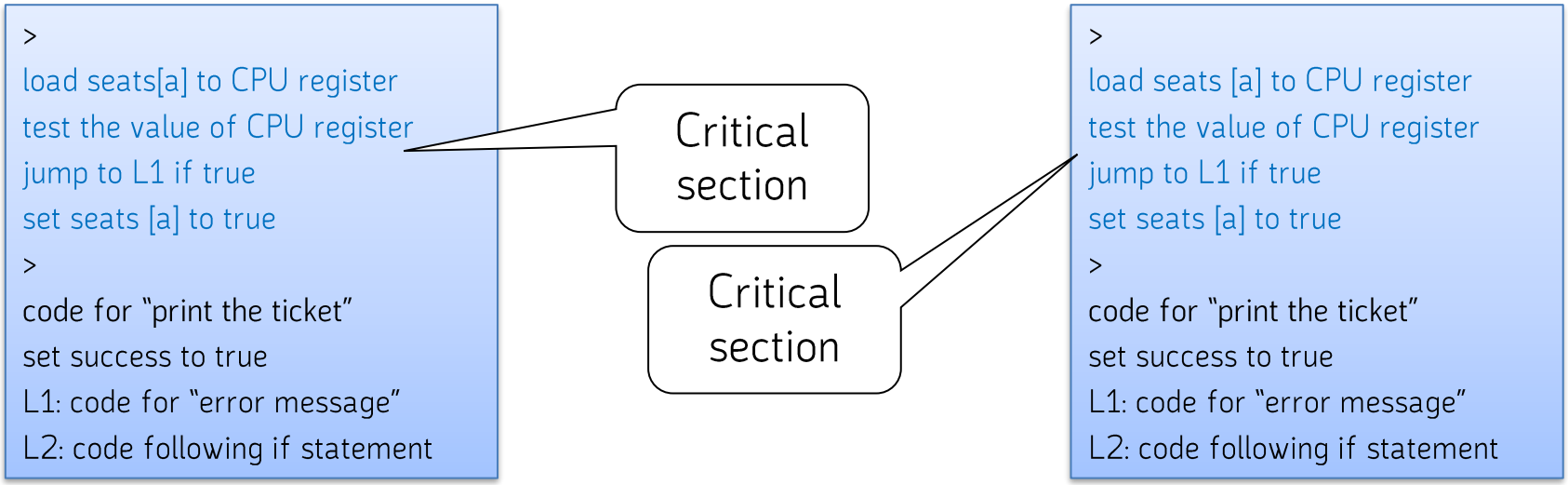
```
>
load seats[a] to CPU register
test the value of CPU register
jump to L1 if true
set seats [a] to true
>
code for "print the ticket"
set success to true
L1: code for "error message"
L2: code following if statement
```

Valid execution TRACE

Ti: check seats[22] as untaken
Ti: seats[22] = true
Tj: check seats[22] as **taken**
...

Example: Tickets Vending Machine

- Atomic sentences: those executed with no interruption
  - Only machine instructions are really atomic
- **Critical section**: a block of code in a process that ought to be executed as atomic
- When two critical sections of two different processes can not be overlapped when executed (interleaving is not allowed) then it is stated that they must be executed with **mutual exclusion**



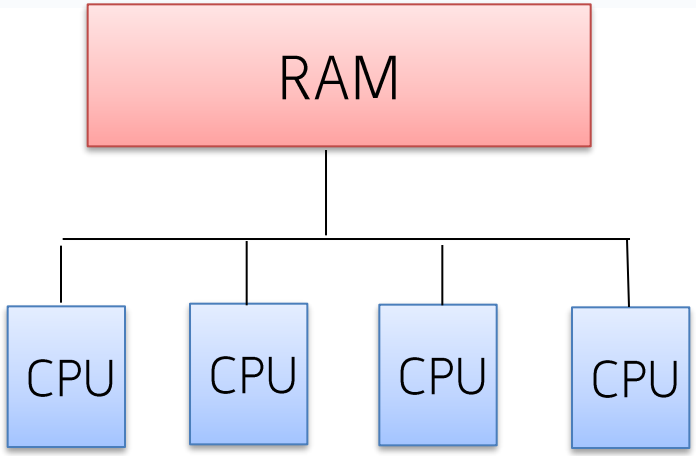
- If CSi and CSj are critical regions of the processes Ti and Tj, then a mutual exclusion states $\forall e.\ (CSi \rightarrow CSj\ ) \vee (CSj \rightarrow CSi\ )$

Platforms

- **Single processor systems**
  - Concurrency only can be implemented by interleaving the sentences of the processes
  - It is useful to serve many users with a single computer
  - When a process executes blocking I/O operations, the idle CPU cycles can be used to execute other processes
  - The memory is shared by all processes, so communication is straightforward by means of shared memory
  - Communications by means of message passing are also allowed

Platforms

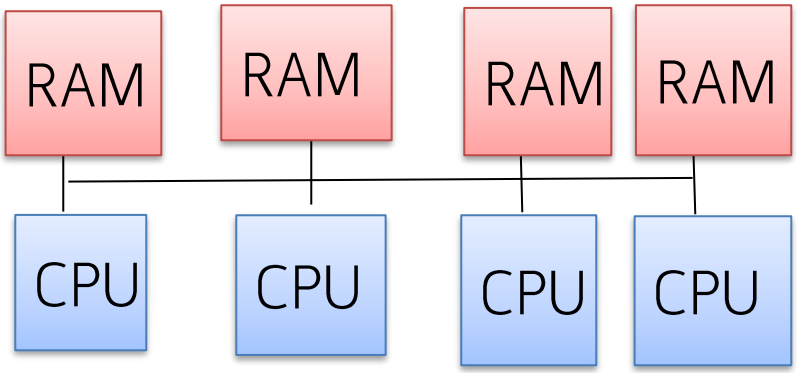
- **Multiprocessors highly coupled**



- A true parallelism can be achieved
- The natural communication among processes is performed by means of shared memory
- Each processor may has its own local memory (usually faster)
- Usually, each CPU is assigned more than one process
- This architecture has a high performance due to the parallel execution and the low cost of communication operations

Platforms

- Multiprocessors loosely coupled (Distributed Systems)



- This architecture allows real concurrency
- Each node of this network may be a single processor or a multiprocessor highly coupled
- The natural communication among processes is performed by means of message passing
- In this architecture communications take much time. Even more, this may lead to neutralize the performance gained due to the parallel execution