

# Programación de Sistemas y Concurrencia

## Tema 3: La Programación Concurrente como Abstracción

Grado en Ingeniería Informática

Grado en Ingeniería del Software

Grado en Ingeniería de Computadores

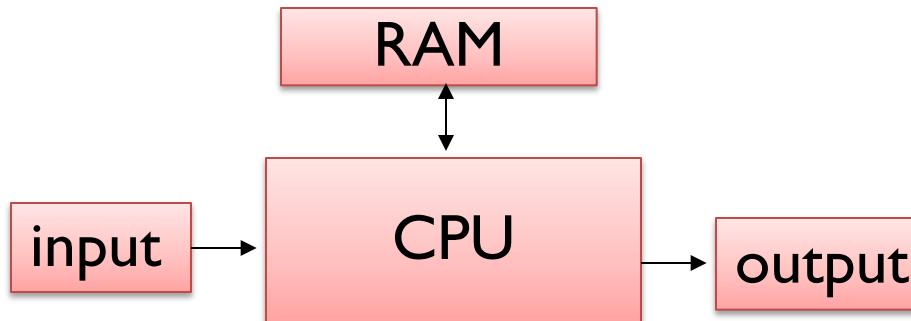
# Contenido

- De la programación secuencial a la programación concurrente
- Beneficios y usos de la programación concurrente
- Problemas de la programación concurrente:
  - Instrucciones atómicas
  - Sección crítica
  - Exclusión mutua
- Plataformas para la ejecución concurrente

# Programación secuencial

- Los lenguajes de programación secuenciales explotan explícitamente las características de la máquina que los ejecuta
- En cada ciclo del reloj
  - Se trae una instrucción de la memoria
  - Se decodifica y se envían señales a los componentes del sistema para que se ejecute

Máquina Von-Neumann



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

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

# Ejemplo

- Si  $P = p_1; p_2; \dots; p_n$  es un programa secuencial
  - $p_1$  siempre precederá a  $p_2$  (cualquiera que sea la ejecución),
  - $p_2$  siempre precederá a  $p_3$
  - ....
  - $p_{n-1}$  siempre precederá a  $p_n$
- Si el símbolo “ $\rightarrow$ ” significa “precede a” y  $\forall e$  significa “para toda ejecución”, el comportamiento puede formalizarse como  $\forall e. (p_1 \rightarrow p_2) \wedge (p_2 \rightarrow p_3) \wedge \dots \wedge (p_{n-1} \rightarrow p_n)$
- Este comportamiento se mantiene incluso si el código  $P$  tiene instrucciones de selección y bucles.

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

- $\forall e. (b \rightarrow A) \vee (b \rightarrow B)$  (depende de los datos de entrada)

# Ejemplo

**while (b) {A}**

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

(el número de iteraciones depende de los datos de entrada)

**Dado  $P = p_1; p_2; \dots; p_n$**

Si  $0 < i < j \leq n$  entonces  $\forall e. p_i \rightarrow p_j$

El programa es determinista.  
Dada una instrucción, y unos datos de entrada, siempre se sabe cual es la siguiente instrucción a ejecutar

Las instrucciones están totalmente ordenadas

# Programación concurrente

- No todas las instrucciones de un programa *tienen que* ejecutarse de forma secuencial

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

El lenguaje nos obliga a establecer un orden de ejecución

# Programación concurrente

- No todas las instrucciones de un programa tienen que ejecutarse de forma secuencial
  - Este código podría ejecutarse de 6 formas distintas y todas ellas correctas

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

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

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

# Programación concurrente

- No todas las instrucciones de un programa tienen que ejecutarse de forma secuencial
- Este código podría ejecutarse de 6 formas distintas y todas ellas correctas
- Supongamos que definimos un nuevo operador `||` para representar este comportamiento

```
x = 0 || y = 0 || z = 0  = {x = 0 → y = 0 → z = 0,  
                           x = 0 → z = 0 → y = 0,  
                           y = 0 → x = 0 → z = 0,  
                           ...  
                           }
```

# Programación concurrente

- Supongamos que definimos un nuevo operador `||` para representar este comportamiento

¿Y si tuviéramos 3 procesadores?

- Podríamos asignar cada código a un procesador, obteniendo un comportamiento correcto y una ejecución posiblemente más rápida.
- **Extendemos** el significado de `P || Q` para indicar que existe una ejecución válida de `P` y `Q`, en la que ambos códigos se solapan en el tiempo

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

# Programación Concurrente

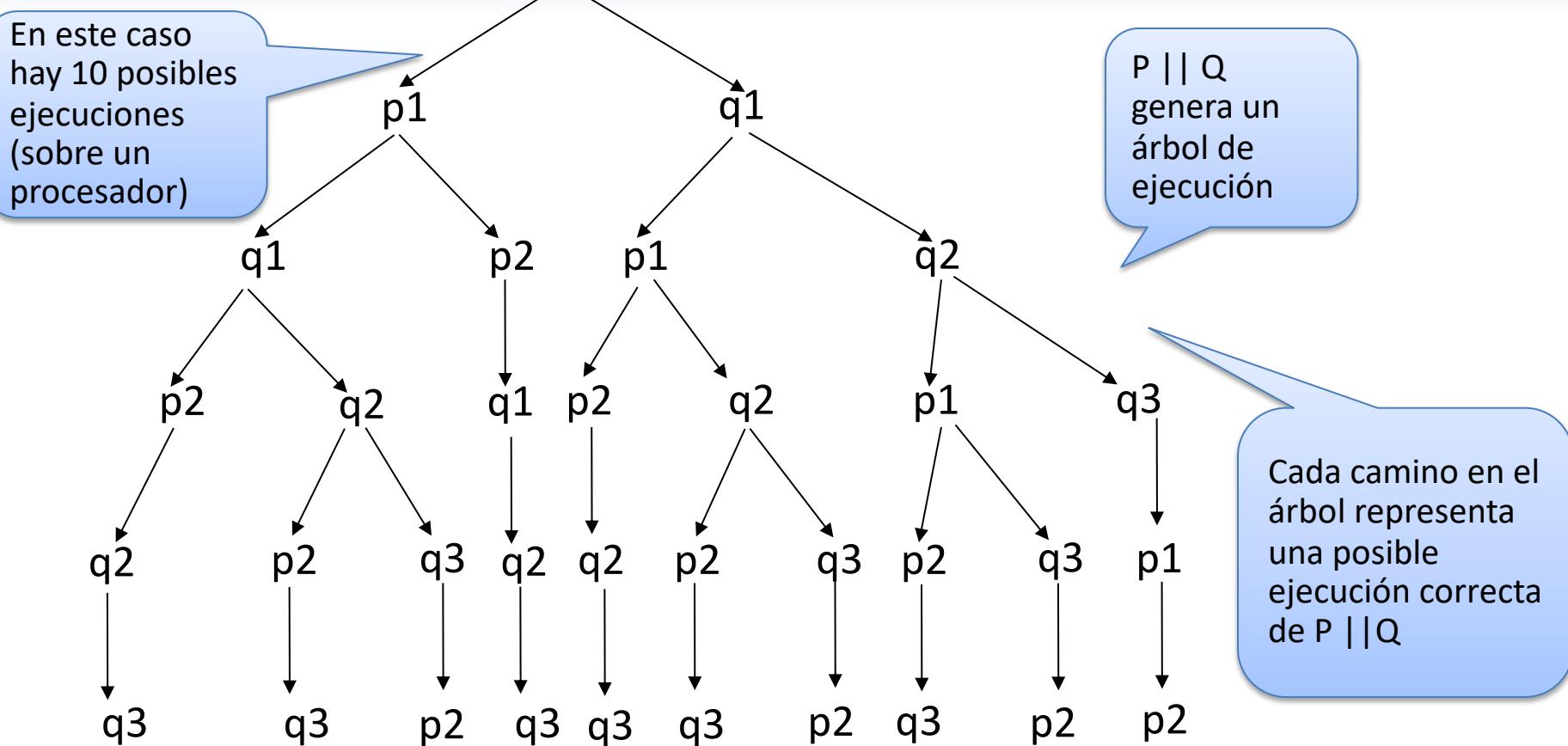
- ¿Cómo razonamos sobre la ejecución de  $P \parallel Q$ ?
  - Podríamos exigir la existencia de dos procesadores para que exista un solapamiento de instrucciones real
    - No es buena idea porque entonces el código de nuestro programa dependería de la arquitectura subyacente
  - En su lugar, suponemos que existen un par de **procesadores lógicos** cada uno ejecutando uno de los códigos.  
Un procesador lógico puede coincidir con uno real, pero ¿qué hacemos si no hay suficientes procesadores?

Supongamos que  $P = p_1 \rightarrow p_2 \rightarrow \dots \rightarrow p_n$  y que  
 $Q = q_1 \rightarrow q_2 \rightarrow \dots \rightarrow q_m$

¿Cómo puede ser la ejecución  $P \parallel Q$  si sólo hay un procesador?

# Interleaving

Supongamos  $n = 2$  y  $m = 3$ :  $P = p_1 \rightarrow p_2$  y  $Q = q_1 \rightarrow q_2 \rightarrow q_3$

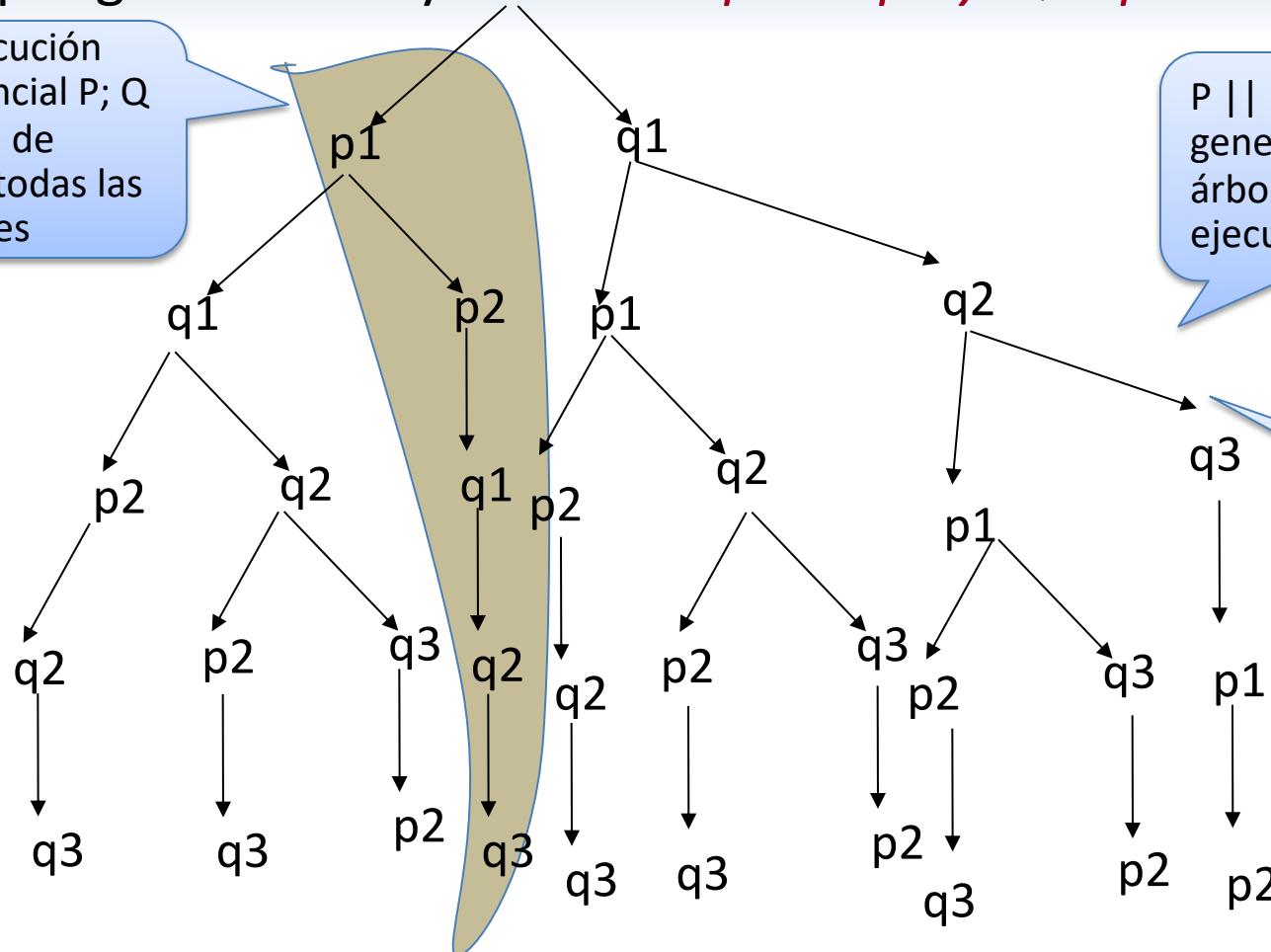


# Interleaving

Supongamos  $n = 2$  y  $m = 3$ :  $P = p_1 \rightarrow p_2$  y  $Q = q_1 \rightarrow q_2 \rightarrow q_3$

La ejecución secuencial P; Q es una de entre todas las posibles

$P \parallel Q$  genera un árbol de ejecución



Cada camino en el árbol representa una posible ejecución correcta de  $P \parallel Q$

# Orden Parcial

- Una ejecución correcta de  $P \parallel Q$   
 $(P = p_1 \rightarrow p_2 \rightarrow \dots \rightarrow p_n, Q = q_1 \rightarrow q_2 \rightarrow \dots \rightarrow q_m)$   
se obtiene intercalando las instrucciones de P y Q,  
**pero no de cualquier forma**
- El **orden relativo** de las instrucciones en P y Q debe mantenerse,

Dados dos índices i, j

El orden en P y Q se conserva

- si  $i < j$  entonces  $\forall e. (p_i \rightarrow p_j) \wedge (q_i \rightarrow q_j)$
- si  $i < j$  entonces  $\forall e. (p_i \rightarrow q_j) \vee (q_j \rightarrow p_i)$

Pero no hay orden establecido entre las instrucciones de P y las de Q

# Orden Parcial

- Una ejecución correcta de  $P \parallel Q$   
 $(P = p_1 \rightarrow p_2 \rightarrow \dots \rightarrow p_n, Q = q_1 \rightarrow q_2 \rightarrow \dots \rightarrow q_m)$   
se obtiene intercalando las instrucciones de P y Q,  
**pero no de cualquier forma**
- El orden relativo de las instrucciones en P y Q debe mantenerse,

Dados dos índices i, j

- si  $i < j$  entonces  $\forall e. (p_i \rightarrow p_j) \wedge (q_i \rightarrow q_j)$
- si  $i < j$  entonces  $\forall e. (p_i \rightarrow q_j) \vee (q_j \rightarrow p_i)$

En cada momento puede haber varias instrucciones posibles a ser ejecutadas.  
La ejecución paralela es, por definición,  
INDETERMINISTA

Las instrucciones de  $P \parallel Q$   
están parcialmente  
ordenadas

# Programación Concurrente

- ¿Cómo razonamos sobre la ejecución de  $P \parallel Q$ ?
  - Podríamos imponer que necesariamente tengamos dos procesadores para que exista un solapamiento de instrucciones real
    - No es buena idea porque entonces el código de nuestro programa dependería de la arquitectura subyacente
  - Suponemos que existen un par de procesadores lógicos cada uno ejecutando uno de los códigos.
    - Un procesador lógico puede coincidir con uno real, pero
- ¿qué hacemos si no hay suficientes procesadores?
  - Intercalamos las instrucciones de P y Q en el procesador
    - (semántica del interleaving)
  - Es como si los procesadores lógicos que ejecutan P y Q evolucionaran a velocidades distintas
  - Por lo tanto,
    - No podemos hacer suposiciones sobre la velocidad relativa entre procesadores lógicos
    - No podemos hacer suposiciones sobre la velocidad real de ejecución de los códigos sobre los procesadores

# Programación Concurrente

Resumiendo:

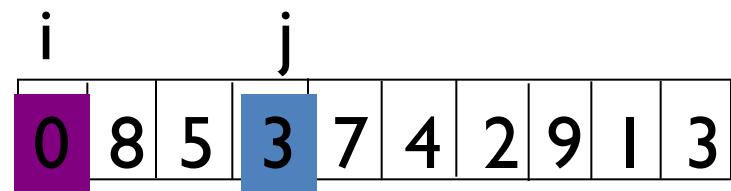
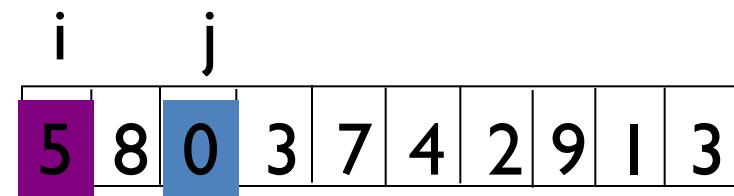
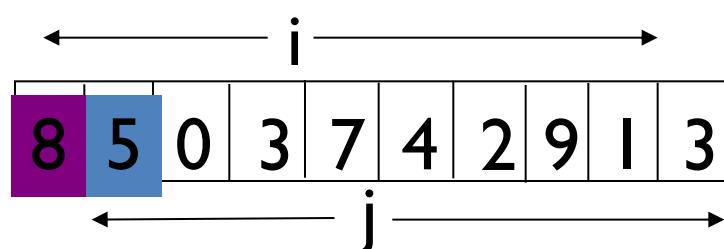
- $P \parallel Q$  representa la ejecución concurrente de P y Q
- P y Q se llaman **procesos/hebras** porque se ejecutan concurrentemente con otros procesos
- Sin embargo, los código de P y Q se ejecutan **secuencialmente**
- Si  $P = p_1 \rightarrow p_2 \rightarrow \dots \rightarrow p_n$  y  $Q = q_1 \rightarrow q_2 \rightarrow \dots \rightarrow q_m$ , el número total de posibles ejecuciones de  $P \parallel Q$  sin tener en cuenta posibles solapamientos de instrucciones es

$$(m+n)!/m! * n!$$

- Cada posible ejecución se denomina **traza** (es un camino en el árbol)
- Para que un programa  $P \parallel Q$  sea **correcto** deben serlo todas sus trazas de ejecución

# Motivación

- Mejorar el rendimiento de los procesadores
- Explotar las arquitecturas multiprocesadores
- Simplificar el modelado de sistemas que son concurrentes de forma natural
- Obtener ganancias en tiempo.
  - Ejemplo ordenación por intercambio



# Ejemplo: Ordenación por intercambio

Implementación del algoritmo de ordenación por intercambio

```
public class Ordenar{  
    public static int intercambio(int[] vector, int inicio, int fin){  
        int numOperBasicas = 0;  
        for (int i = inicio; i<fin-1; i++){  
            for (int j = i+1; j < fin; j++){  
                if (vector[i] > vector[j]){  
                    numOperBasicas++;  
                    int aux = vector[i];  
                    vector[i] = vector[j];  
                    vector[j] = aux;  
                }  
            }  
        }  
        return numOperBasicas;  
    }  
}
```

# Ejemplo: Ordenación por intercambio. Opción 1

Caso secuencial: ordenar el array con una llamada al algoritmo anterior

```
public static void main(String[] args){  
    int[] vector = new int[2000];  
    Random r = new Random();  
    for (int i = 0; i<vector.length; i++)  
        vector[i] = r.nextInt(25);  
    System.out.println("Vector desordenado");  
    for (int i = 0; i<vector.length; i++)  
        System.out.print(vector[i]+ " ");  
    System.out.println();  
  
    Ordenar.intercambio(vector, 0, vector.length);  
  
    System.out.println("Vector ordenado");  
    for (int i = 0; i<vector.length; i++)  
        System.out.print(vector[i]+ " ");  
}
```

# Ejemplo: Ordenación por intercambio. Opción 2

Utilizar la técnica de divide y vencerás: dividimos el array en dos mitades. Llamamos dos veces al algoritmo para ordenar cada mitad. Necesitamos un método que mezcle las dos mitades ya ordenadas.

```
public static void main(String[] args){  
    int[] vector = new int[2000];  
    Random r = new Random();  
    for (int i = 0; i<vector.length; i++)  
        vector[i] = r.nextInt(25);  
  
    System.out.println("Vector desordenado");  
    for (int i = 0; i<vector.length; i++)  
        System.out.print(vector[i]+ " ");  
    System.out.println();  
  
    Ordenar.intercambio(vector, 0, vector.length/ 2);  
    Ordenar.intercambio(vector, vector.length/ 2, vector.length);  
  
    Ordenar.mezclar(vector,0, vector.length/ 2, vector.length);  
}
```

```
public static void mezclar(int[] v,int inic, int m, int fin){  
    int i = inic;  
    int j = m;  
    while ((i< m) && (j < fin)){  
        if (v[i]<= v[j]){  
            System.out.print(v[i] + " ");  
            i++;  
        } else {  
            System.out.print(v[j] + " ");  
            j++;  
        }  
    }  
    while (i< m) {  
        System.out.print(vector[i] + " "); i++;  
    }  
    while (j < fin){  
        System.out.print(vector[j] + " "); j++;  
    }  
}
```

# Ejemplo: Ordenación por intercambio. Opción 3

Caso Paralelo: utilizar hebras + divide y vencerás: utilizamos hebras diferentes para ordenar cada mitad del array. Al finalizar mezclamos las dos mitades ya ordenadas

```
class HebraO extends Thread{  
    int[] vector;  
    int inicio,fin;  
    public HebraO (int[] vector,int i,int j){  
        this.vector = vector;  
        inicio = i;  
        fin = j;  
    }  
    public void run() {  
        Ordenar.intercambio(vector, inicio, fin);  
    }  
}  
  
HebraO o1 = new HebraO(vector,0,vector.length/2);  
HebraO o2 = new HebraO(vector, vector.length/2,vector.length);  
  
o1.start();  
o2.start();  
//espero que terminen !! IMPORTANTE !!  
Ordenar.mezclar(vector, 0, vector.length/ 2, vector.length);
```

# Ejemplo: Ordenación por intercambio

## Estudio de las complejidades

```
public class Ordenar{  
    public static int intercambio(int[] vector, int inicio,int fin){  
        int numOperBasicas = 0;  
        for (int i = inicio; i<fin-1; i++)  
            for (int j = i+1; j < fin; j++){  
                if (vector[i] > vector[j]) {  
                    numOperBasicas++;  
                    int aux = vector[i];  
                    vector[i] = vector[j];  
                    vector[j] = aux;    }  
    } }  
}
```

Si  $n$  es el número de elementos del vector a ordenar

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

# Ejemplo: Ordenación por intercambio

Si  $n$  es el número de elementos del vector a ordenar

Caso secuencial:  $T(n) \sim n^2/2$

Caso secuencial con dos llamadas:  $T(n) \sim n^2/4 + n$

Caso paralelo  $T(n) \sim n^2/8 + n$

$n \backslash T(n)$	$n^2/2$	$n^2/4+n$	$n^2/8+n$
20	100	120	60
40	800	440	240
1000	500000	251000	126000

# Comunicación y Sincronización

Los procesos en un programa concurrente habitualmente se comunican **y** sincronizan sus acciones.

- ▶ La comunicación puede realizarse a través de la memoria compartida
- ▶ O a través de algún mecanismo de paso de mensajes

## Global x

```
emisor{  
    ....  
    x = m  
}
```

```
receptor{  
    local y  
    ....  
    y = x  
}
```

```
emisor{  
    ....  
    send m to receptor  
    ....  
}
```

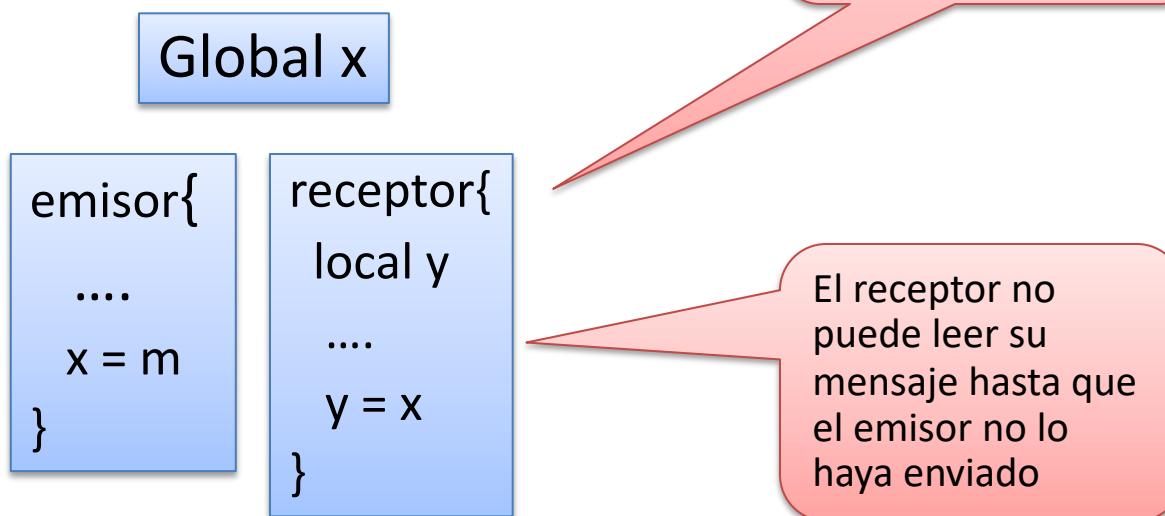
```
receptor{  
    local y  
    ....  
    receive y from emisor  
    ....  
}
```

# Comunicación y Sincronización

Los procesos en un programa concurrente habitualmente se comunican **y sincronizan sus acciones**.

- ▶ La comunicación puede realizarse a través de la memoria compartida

Esto es un ejemplo de una condición de sincronización



El receptor no puede leer su mensaje hasta que el emisor no lo haya enviado

# Problemas de la programación concurrente

## Instrucciones atómicas

- ▶ Sección crítica
- ▶ Exclusión mutua
- ▶ Ejemplo: Máquinas vendedoras de entradas

Pueden comprarse entradas desde distintas taquillas



# Problemas de la programación concurrente



- Cada taquilla es un proceso  $T_i$
- Todas las taquillas se ejecutan concurrentemente
- Todas las taquillas ejecutan el mismo código
- Suponemos que hay un array de booleanos que representa los asientos del teatro

boolean asientos[]

$T_1 \parallel T_2 \parallel \dots \parallel T_n$

## Código de $T_1$

```
...
while (true){
    mostrar butacas libres al usuario
    a = ... //asiento seleccionado
    asientos[a] = true
    emitir la entrada
}
```

....

## Código de $T_n$

```
...
while (true){
    mostrar butacas libres al usuario
    a = ... //asiento seleccionado
    asientos[a] = true
    emitir la entrada
}
```

# Ejemplo: Máquinas vendedoras de entradas

- ▶ Esta solución es incorrecta. Es posible que **dos personas distintas compren el mismo asiento**

Ti: mostrar butacas libres al usuario

Tj: mostrar butacas libres al usuario

Ti: a = ...; //asiento seleccionado en la taquilla i, por ejemplo a = 22

Tj: a = ...; //asiento seleccionado en la taquilla j, por ejemplo a = 22

Ti: asientos[22] = true

Tj: asientos[22] = true

Ti: emitir la entrada

Tj: emitir la entrada



Esto es una traza de ejecución, un posible camino en el árbol, que muestra un error. Sirve para demostrar que un programa es incorrecto

## Código de cada taquilla

```
...
while (true){
    mostrar butacas libres al usuario
    a = ... //asiento seleccionado
    asientos[a] = true
    emitir la entrada
}
```

El asiento 22 se ha vendido a dos clientes distintos

# Ejemplo: Máquinas vendedoras de entradas

## Refinamos el código

### Código de Ti

```
...
while (true){
    exito = false;

    while (!exito){
        mostrar butacas libres al usuario
        a=...; //asiento seleccionado
        if (! asientos[a]) {
            asientos[a] = true ;
            emitir la entrada
            exito = true;
        } else {
            dar un mensaje de error
        }
    }
}
```

Añadimos este código para asegurarnos de que cuando se selecciona una butaca está realmente libre

# Ejemplo: Máquinas vendedoras de entradas

## Código de Ti

```
...
while (true){
    exito = false;

    while (!exito){
        mostrar butacas libres al usuario
        a=...; //asiento seleccionado
        if (! asientos[a]) {
            asientos[a] = true ;
            emitir la entrada
            exito = true;
        } else {
            dar un mensaje de error
        }
    }
}
```

## TRAZA de ejecución que muestra el error

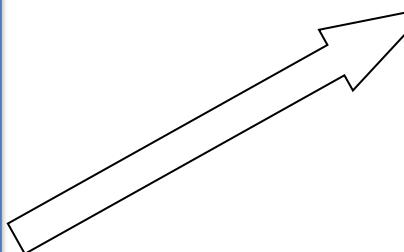
```
Ti: chequea asiento[22] y lo encuentra libre
Tj: chequea asiento[22] y lo encuentra libre
Ti: asientos[22] = true
Tj: asientos[22] = true
Ti: emitir la entrada
Tj: emitir la entrada
```

Estamos en la misma situación de antes, salvo que ahora tenemos localizado el problema

# Ejemplo: Máquinas vendedoras de entradas

- **Instrucciones atómicas:** las que el procesador realiza sin interrupción.
  - Sólo las instrucciones máquina son atómicas

```
while (true){  
    exito = false;  
  
    while (!exito){  
        mostrar butacas libres al usuario  
        a=...; //asiento seleccionado  
        if (! asientos[a]) {  
            asientos[a] = true ;  
            emitir la entrada  
            exito = true;  
        } else {  
            dar un mensaje de error  
        }  
    }  
}
```



¿y si el código marcado fuera atómico?

load asientos[a] to CPU register  
test the value of CPU register  
jump to L1 if false  
set asientos[a] to true  
code for “emitir la entrada”  
set exito to true  
jump to L2  
L1: code for “mensaje error”  
L2: code following if statement

>  
load asientos[a] to CPU register  
test the value of CPU register  
jump to L1 if false  
set asientos[a] to true  
>  
code for “emitir la entrada”  
set exito to true  
jump to L2  
L1: code for “mensaje error”  
L2: code following if statement

# Ejemplo: Máquinas vendedoras de entradas

- Instrucciones atómicas: las que el procesador realiza sin interrupción.
  - Sólo las instrucciones máquina son atómicas

Si el código azul fuera atómico la traza errónea ya no podría ocurrir

Esta ya no sería una TRAZA de ejecución posible en nuestro programa:

Ti: chequea asientos[22] y lo encuentra libre

Tj: chequea asientos[22] y lo encuentra libre

Ti: asientos[22] = true

Tj: asientos[22] = true

Ti: emitir la entrada

Tj: emitir la entrada

>  
load asiento[a] to CPU register  
test the value of CPU register  
jump to L1 if false  
set asiento[a] to true  
>  
code for “emitir la entrada”  
set exito to true  
jump to L2  
L1: code for “mensaje error”  
L2: code following if statement

# Ejemplo: Máquinas vendedoras de entradas

- Instrucciones atómicas: las que el procesador realiza sin interrupción.
  - Sólo las instrucciones máquina son atómicas

El único entrelazado posible ocurre  
**ANTES O DESPUÉS** de  
que Ti haya ejecutado el código azul

## TRAZA de ejecución no errónea

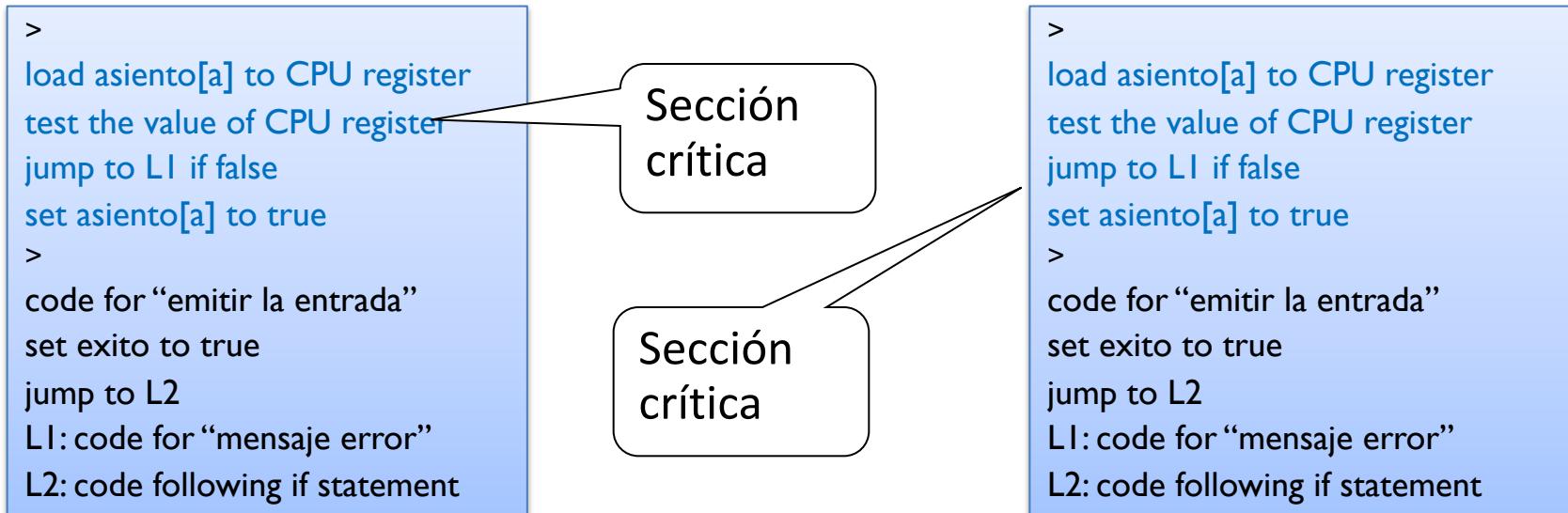
Ti: chequea asiento[22] y lo encuentra libre  
Ti: asientos[22] = true  
Tj: chequea asiento[22] y lo encuentra **ocupado**

...

```
>  
load asiento[a] to CPU register  
test the value of CPU register  
jump to L1 if false  
set asiento[a] to true  
>  
code for "emitir la entrada"  
set exito to true  
jump to L2  
L1: code for "mensaje error"  
L2: code following if statement
```

# Ejemplo: Máquinas vendedoras de entradas

- **Sección crítica:** parte del código de un proceso que *debería* ejecutarse de forma atómica
- Cuando dos secciones críticas de dos procesos no pueden solaparse en el tiempo (su código no puede entrelazarse) se dice que deben ejecutarse en *exclusión mutua*



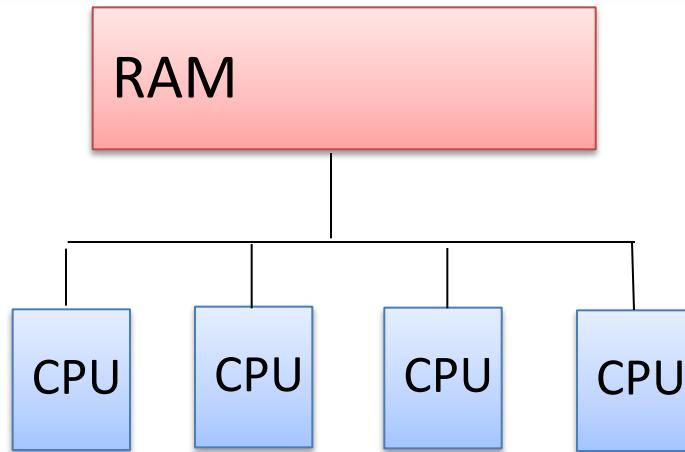
- Si SC<sub>i</sub> y SC<sub>j</sub> son dos secciones críticas de los procesos T<sub>i</sub> y T<sub>j</sub>, la exclusión mutua significa que  $\forall e. (SC_i \rightarrow SC_j) \vee (SC_j \rightarrow SC_i)$

# Plataformas

- Sistemas monoprocesadores
  - La concurrencia siempre se implementa utilizando el entrelazado de las instrucciones de los procesos.
  - Es útil para dar servicio a varios usuarios desde la misma máquina
  - Se aprovechan los ciclos del procesador mientras que está realizando operaciones de entrada/salida
  - Todos los procesos comparten memoria, por lo que la comunicación se realiza de forma natural a través de esta memoria compartida.
  - También es posible modelar sistemas de memoria distribuida en los que los procesos se comunican a través del paso de mensajes.

# Plataformas

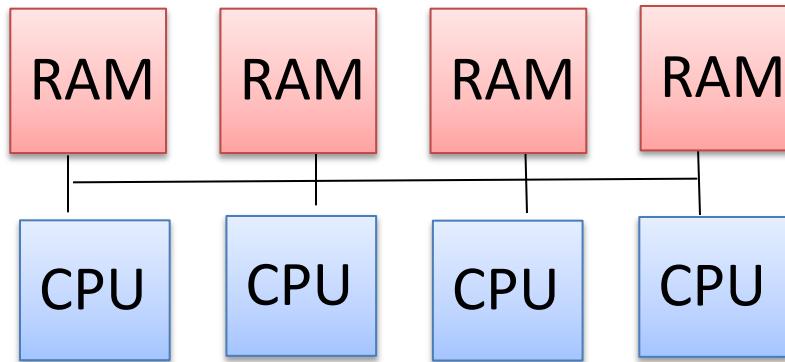
- Multiprocesadores fuertemente acoplados



- Es posible tener paralelismo real
- Los procesos se comunican de forma natural a través del espacio de memoria común que comparten
- Cada procesador puede tener a su vez memoria local
- Normalmente a cada CPU le corresponde más de un proceso
- En este tipo de arquitectura hay buenas ganancias en tiempo por la ejecución paralela y porque el costo de comunicaciones es bajo.

# Plataformas

- Multiprocesadores débilmente acoplados (Sistemas distribuidos)



- En esta arquitectura hay concurrencia real
- Cada nodo de la red podría ser un monoprocesador o un multiprocesador fuertemente acoplado
- La comunicación se realiza de forma natural mediante paso de mensajes a los procesos
- El coste de las comunicaciones en esta arquitectura es relevante, y puede degradar las ganancias en tiempo obtenidas debido a la ejecución paralela