

# Programación Concurrente

## Clase 2



Facultad de Informática  
UNLP

# Links a los archivos con audio (formato MP4)

Los archivos con las clases con audio están en formato MP4. En los link de abajo están los videos comprimidos en archivos RAR.

- ◆ Clases de Instrucciones

<https://drive.google.com/uc?id=1bdsNk8uY2MKpA3usLnp8tqZt8nG6pZRU&export=download>

- ◆ Acciones Atómicas y Sincronización

<https://drive.google.com/uc?id=1DzEl1aKJ-fXW9k3t7tDgy59C9HtdS2vf&export=download>

- ◆ Propiedades y Fairness

<https://drive.google.com/uc?id=1lxnI0SIV-movMHbamVD2tl6VYmRS4Vij&export=download>



---

# Clases de Instrucciones

---

# Clases de instrucciones

## Programación secuencial y concurrente

Un programa concurrente esta formado por un conjunto de programas secuenciales.

- La programación secuencial estructurada puede expresarse con 3 clases de instrucciones básicas: **asignación**, **alternativa** (decisión) e **iteración** (repetición con condición).
- Se requiere una clase de instrucción para representar la concurrencia.

### DECLARACIONES DE VARIABLES

- Variable simple: **tipo variable = valor** . Ej: **int x = 8; int z, y;**
- Arreglos: **int a[10]; int c[3:10]**  
**int b[10] = ([10] 2)**  
**int aa[5,5]; int cc[3:10,2:9]**  
**int bb[5,5] = ([5] ([5] 2))**

# Clases de instrucciones

## Programación secuencial y concurrente

### ASIGNACION

- Asignación simple:  $\mathbf{x = e}$
- Sentencia de asignación compuesta:  $\mathbf{x = x + 1; y = y - 1; z = x + y}$   
 $\mathbf{a[3] = 6; aa[2,5] = a[4]}$
- Llamado a funciones:  $\mathbf{x = f(y) + g(6) - 7}$
- swap:  $\mathbf{v1} ::= \mathbf{v2}$
- **skip**: termina inmediatamente y no tiene efecto sobre ninguna variable de programa.

# Clases de instrucciones

## Programación secuencial y concurrente

### ALTERNATIVA

- Sentencias de alternativa simple:  
    **if B  $\rightarrow$  S**  
    B expresión booleana. S instrucción simple o compuesta (`{ }`).  
    **B “guarda” a S** pues S no se ejecuta si B no es verdadera.
- Sentencias de alternativa múltiple:  
    **if B1  $\rightarrow$  S1**  
    **□ B2  $\rightarrow$  S2**  
    .....  
    **□ Bn  $\rightarrow$  Sn**  
    **fi**  
    Las guardas se evalúan en algún orden arbitrario.  
    Elección no determinística.  
    Si ninguna guarda es verdadera el *if* no tiene efecto.
- Otra opción:  
    **if (cond) S;**  
    **if (cond) S1 else S2;**

# Clases de instrucciones

## Programación secuencial y concurrente

### Ejemplos de *Sentencia Alternativa Múltiple*

Ejemplo 1:

```
if p > 2 → p = p * 2
  □ p < 2 → p = p * 3
  □ p == 2 → p = 5
fi
```

¿Puede terminar sin tener efecto?

Ejemplo 2:

```
if p > 2 → p = p * 2
  □ p < 2 → p = p * 3
fi
```

¿Que sucede si  $p = 2$ ?

Ejemplo 3:

```
if p > 2 → p = p * 2
  □ p < 6 → p = p + 4
  □ p == 4 → p = p / 2
fi
```

¿Que sucede con los siguiente valores de  $p = 1, 2, 3, 4, 5, 6, 7$ ?

# Clases de instrucciones

## Programación secuencial y concurrente

### ITERACIÓN

- Sentencias de alternativa ITERATIVA múltiple:

**do**  $B1 \rightarrow S1$

$\square B2 \rightarrow S2$

....

$\square Bn \rightarrow Sn$

**od**

Las sentencias guardadas son evaluadas y ejecutadas hasta que todas las guardas sean falsas.

La elección es no determinística si más de una guarda es verdadera.

- For-all: forma general de repetición e iteración

**fa** cuantificadores  $\rightarrow$  Secuencia de Instrucciones **af**

Cuantificador  $\equiv$  **variable**  $:=$  exp\_inicial **to** exp\_final **st** **B**

El cuerpo del *fa* se ejecuta 1 vez por cada combinación de valores de las variables de iteración. Si hay cláusula *such-that* (*st*), la variable de iteración toma sólo los valores para los que *B* es true.

Ejemplo: **fa**  $i := 1$  **to**  $n, j := i+1$  **to**  $n$  **st**  $a[i] > a[j] \rightarrow a[i] := a[j]$  **af**

- Otra opción:

**while** (cond) **S**;

**for** [ $i = 1$  **to**  $n, j = 1$  **to**  $n$  **st** ( $j \bmod 2 = 0$ )] **S**;



# Clases de instrucciones

## Programación secuencial y concurrente

### Ejemplos de *Sentencia Alternativa Iterativa Múltiple*

Ejemplo 1:

```
do p > 0 → p = p - 2
  □ p < 0 → p = p + 3
  □ p == 0 → p = random(x)
od
```

¿Cuándo termina?

Ejemplo 2:

```
do p > 2 → p = p * 2
  □ p < 2 → p = p * 3
od
```

¿Cuándo termina?

Ejemplo 3:

```
do p > 0 → p = p - 2
  □ p > 3 → p = p + 3
  □ p > 6 → p = p / 2
od
```

¿Cuándo termina?

¿Que sucede con  $p = 0, 3, 6, 9$ ?

Ejemplo 4:

```
do p == 1 → p = p * 2
  □ p == 2 → p = p + 3
  □ p == 4 → p = p / 2
od
```

¿Cuándo termina?

# Clases de instrucciones

## Programación secuencial y concurrente

### Ejemplos de *For-All*

fa  $i := 1$  to  $n \rightarrow a[i] = 0$  af

Inicialización de un vector

fa  $i := 1$  to  $n, j := i+1$  to  $n \rightarrow m[i,j] := m[j,i]$  af

Trasposición de una matriz

fa  $i := 1$  to  $n, j := i+1$  to  $n$  st  $a[i] > a[j] \rightarrow a[i] := a[j]$  af

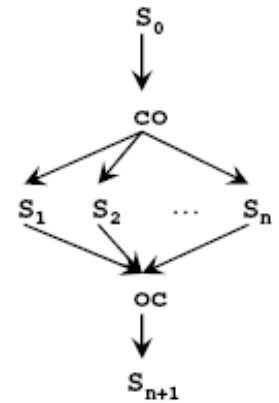
Ordenación de menor a mayor de un vector

# Clases de instrucciones

## Programación secuencial y concurrente

### CONCURRENCIA

- Sentencia **co**:  
**co S1 // .... // Sn oc** → Ejecuta las  $S_i$  tareas concurrentemente.  
La ejecución del **co** termina cuando todas las tareas terminaron.  
Cuantificadores.  
**co [i=1 to n] { a[i]=0; b[i]=0 } oc** → Crea  $n$  tareas concurrentes.
- **Process**: otra forma de representar concurrencia  
**process A {sentencias}** → proceso único independiente.  
Cuantificadores.  
**process B [i=1 to n] {sentencias}** →  $n$  procesos independientes.
- **Diferencia**: **process** ejecuta en **background**, mientras el código que contiene un **co** espera a que el proceso creado por la sentencia **co** termine antes de ejecutar la siguiente sentencia.



# Clases de instrucciones


## Programación secuencial y concurrente

Ejemplo: ¿qué imprime en cada caso? ¿son equivalentes?

```
process imprime10
{
    for [i=1 to 10] write(i);
}
```

```
process imprime1 [i= 1..10]
{
    write(i);
}
```

*No determinismo....*



---

# Acciones Atómicas y Sincronización

---

# Atomicidad de grano fino

- **Estado** de un programa concurrente.
- Cada proceso ejecuta un conjunto de sentencias, cada una implementada por una o más acciones atómicas.
- Una **acción atómica** hace una transformación de estado indivisibles (estados intermedios invisibles para otros procesos).
- Ejecución de un programa concurrente → **intercalado** (*interleaving*) de las acciones atómicas ejecutadas por procesos individuales.
- **Historia** de un programa concurrente (*trace*): ejecución de un programa concurrente con un *interleaving* particular. En general el número de posibles historias de un programa concurrente es enorme; pero no todas son válidas.
- **Interacción** → determina cuales historias son correctas.

# Atomicidad de grano fino

- Algunas historias son válidas y otras no.

```
int buffer;
```

```
process 1
```

```
{ int x
```

```
  while (true)
```

```
    p1.1: read(x);
```

```
    p1.2: buffer = x;
```

```
}
```

```
process 2
```

```
{ int y;
```

```
  while (true)
```

```
    p2.1: y = buffer;
```

```
    p2.2: print(y);
```

```
}
```

**Posibles historias:**

p11, p12, p21, p22, p11, p12, p21, p22, ... ☒

p11, p12, p21, p11, p22, p12, p21, p22, ... ☒

p11, p21, p12, p22, .... ☐

p21, p11, p12, .... ☐

- Se debe asegurar un orden temporal entre las acciones que ejecutan los procesos → las tareas se intercalan ⇒ deben fijarse restricciones.

*La sincronización por condición permite restringir las historias de un programa concurrente para asegurar el orden temporal necesario.*

# Atomicidad de grano fino

Una acción atómica de *grano fino* (fine grained) se debe implementar por hardware.

- ¿La operación de asignación  $A=B$  es atómica?

**NO**  $\Rightarrow$  (i) Load PosMemB, reg  
(ii) Store reg, PosMemA

- ¿Qué sucede con algo del tipo  $X=X+X$ ?

(i) Load PosMemX, Acumulador  
(ii) Add PosMemX, Acumulador  
(iii) Store Acumulador, PosMemX



# Atomicidad de grano fino

**Ejemplo 1:** Cuáles son los posibles resultados con 3 procesadores. La lectura y escritura de las variables x, y, z son atómicas.

x = 0; y = 4; z=2;

co

x = y + z

// y = 3

// z = 4

oc

(1)

(2)

(3)

(1) Puede descomponerse por ejemplo en:

(1.1) Load PosMemY, Acumulador

(1.2) Add PosMemZ, Acumulador

(1.3) Store Acumulador, PosMemX

(2) Se transforma en: Store 3, PosMemY

(3) Se transforma en: Store 4, PosMemZ

- y = 3, z = 4 en todos los casos.
- x puede ser:
  - 6 si ejecuta (1)(2)(3) o (1)(3)(2)
  - 5 si ejecuta (2)(1)(3)
  - 8 si ejecuta (3)(1)(2)
  - 7 si ejecuta (2)(3)(1) o (3)(2)(1)
  - 6 si ejecuta (1.1)(2)(1.2)(1.3)(3)
  - 8 si ejecuta (1.1)(3)(1.2)(1.3)(2)
  - .....

# Atomicidad de grano fino

**Ejemplo 2:** Cuáles son los posibles resultados con 2 procesadores. La lectura y escritura de las variables x, y, z son atómicas.

```
x = 2; y = 2;
```

```
co
```

```
z = x + y      (1)
```

```
// x = 3; y = 4;  (2)
```

```
oc
```

(1) Puede descomponerse por ejemplo en:

(1.1) Load PosMemX, Acumulador

(1.2) Add PosMemY, Acumulador

(1.3) Store Acumulador, PosMemZ

(2) Se transforma en:

(2.1) Store 3, PosMemX

(2.2) Store 4, PosMemY

x = 3, y = 4 en todos los casos.

z puede ser: 4, 5, 6 o 7.

Nunca podría parar el programa y ver un estado en que  $x+y = 6$ , a pesar de que  $z = x + y$  si puede tomar ese valor

# Atomicidad de grano fino

## Ejemplo 3: “Interleaving extremo” (Ben-Ari & Burns)

Dos procesos que realizan (cada uno)  $N$  iteraciones de la sentencia  $X=X+1$ .

```
int X = 0  
Process P1  
{ int i  
  for [i=1 to N] → X=X+1  
}  
Process P2  
{ int i  
  fa [i=1 to N] → X=X+1  
}
```

¿Cuál puede ser el valor final de  $X$ ?

- $2N$
- entre  $N+1$  y  $2N-1$
- $N$
- $< N$  (incluso 2...)

### ¿Cuándo valdrá $2N$ ?

En cada iteración ....

1. Proceso 1: *Load X*
2. Proceso 1: *Incrementa su copia*
3. Proceso 1: *Store X*
4. Proceso 2: *Load X*
5. Proceso 2: *Incrementa su copia*
6. Proceso 2: *Store X*

### ¿Cuándo valdrá $N$ ?

En cada iteración ....

1. Proceso 1: *Load X*
2. Proceso 2: *Load X*
3. Proceso 1: *Incrementa su copia*
4. Proceso 2: *Incrementa su copia*
5. Proceso 1: *Store X*
6. Proceso 2: *Store X*

# Atomicidad de grano fino

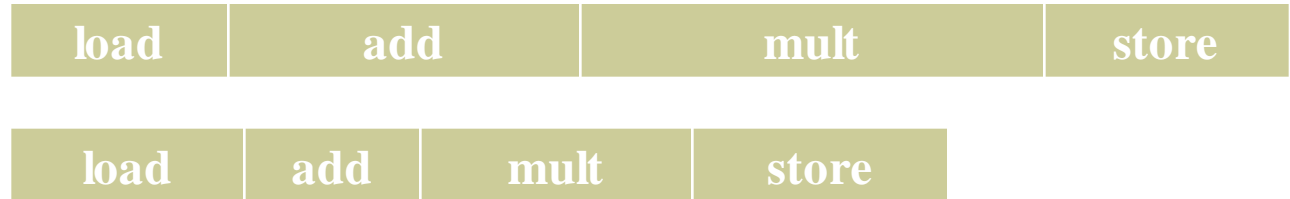
## ¿Cuándo valdrá 2?

1. Proceso 1: *Load X*
2. Proceso 2: *Hace N-1 iteraciones del loop*
3. Proceso 1: *Incrementa su copia*
4. Proceso 1: *Store X*
5. Proceso 2: *Load X*
6. Proceso 1: *Hace el resto de las iteraciones del loop*
7. Proceso 2: *Incrementa su copia*
8. Proceso 2: *Store X*

... no podemos confiar en la intuición para analizar un programa concurrente...

# Atomicidad de grano fino

- ◆ En la mayoría de los sistemas el tiempo absoluto no es importante.
- ◆ Con frecuencia los sistemas son actualizados con componentes más rápidas. La corrección no debe depender del tiempo absoluto.
- ◆ El tiempo se ignora, sólo las secuencias son importantes



- ◆ Puede haber distintos ordenes (*interleavings*) en que se ejecutan las instrucciones de los diferentes procesos; los programas deben ser correctos para todos ellos.

# Atomicidad de grano fino

En lo que sigue, supondremos máquinas con las siguientes características:

- Los valores de los tipos básicos se almacenan en elementos de memoria leídos y escritos como acciones atómicas.
- Los valores se cargan en registros, se opera sobre ellos, y luego se almacenan los resultados en memoria.
- Cada proceso tiene su propio conjunto de registros (context switching).
- Todo resultado intermedio de evaluar una expresión compleja se almacena en registros o en memoria privada del proceso.

# Atomicidad de grano fino

- Si una expresión  $e$  en un proceso no referencia una variable alterada por otro proceso, la evaluación será atómica, aunque requiera ejecutar varias acciones atómicas de grano fino.
- Si una asignación  $x = e$  en un proceso no referencia ninguna variable alterada por otro proceso, la ejecución de la asignación será atómica.

*Normalmente los programas concurrentes no son disjuntos  $\Rightarrow$  es necesario establecer algún requerimiento más débil ...*

**Referencia crítica** en una expresión  $\Rightarrow$  referencia a una variable que es modificada por otro proceso.

Asumamos que toda referencia crítica es a una variable simple leída y escrita atómicamente.

# Atomicidad de grano fino

## Propiedad de “A lo sumo una vez”

Una sentencia de asignación  $x = e$  satisface la propiedad de “A lo sumo una vez” si:

- 1)  $e$  contiene a lo sumo una referencia crítica y  $x$  no es referenciada por otro proceso, o
- 2)  $e$  no contiene referencias críticas, en cuyo caso  $x$  puede ser leída por otro proceso.

Una expresiones  $e$  que no está en una sentencia de asignación satisface la propiedad de “A lo sumo una vez” si no contiene más de una referencia crítica.

*Puede haber a lo sumo una variable compartida, y puede ser referenciada a lo sumo una vez*



# Atomicidad de grano fino

## Propiedad de “A lo sumo una vez”

Si una sentencia de asignación cumple la propiedad ASV, entonces su ejecución *parece* atómica, pues la variable compartida será leída o escrita sólo una vez.

### Ejemplos:

- `int x=0, y=0;`  
`co x=x+1 // y=y+1 oc;`  
No hay ref. críticas en ningún proceso.  
En todas las historias  $x = 1$  e  $y = 1$
- `int x = 0, y = 0;`  
`co x=y+1 // y=y+1 oc;`  
El 1er proceso tiene 1 ref. crítica. El 2do ninguna.  
Siempre  $y = 1$  y  $x = 1$  o  $2$
- `int x = 0, y = 0;`  
`co x=y+1 // y=x+1 oc;`  
Ninguna asignación satisface ASV.  
Posibles resultados:  $x = 1$  e  $y = 2$  /  $x = 2$  e  $y = 1$   
***Nunca debería ocurrir  $x = 1$  e  $y = 1 \rightarrow ERROR$***

# Especificación de la sincronización

- Si una expresión o asignación no satisface ASV con frecuencia es necesario ejecutarla atómicamente.
- En general, es necesario ejecutar secuencias de sentencias como una única acción atómica (*sincronización por exclusión mutua*).

Mecanismo de sincronización para construir una acción atómica *de grano grueso* (*coarse grained*) como secuencia de acciones atómicas de grano fino (*fine grained*) que aparecen como indivisibles.

**⟨e⟩** indica que la expresión *e* debe ser evaluada atómicamente.

**⟨await (B) S;⟩** se utiliza para especificar sincronización.

La expresión booleana *B* especifica una condición de demora.

*S* es una secuencia de sentencias que se garantiza que termina.

Se garantiza que *B* es true cuando comienza la ejecución de *S*.

*Ningún estado interno de S es visible para los otros procesos.*

# Especificación de la sincronización

Sentencia con alto poder expresivo, pero el costo de implementación de la forma general de *await* (exclusión mutua y sincronización por condición) es alto.

- *Await general:*      $\langle \text{await } (s > 0) \text{ } s = s - 1; \rangle$

- *Await para exclusión mutua:*      $\langle x = x + 1; y = y + 1 \rangle$

- *Ejemplo await para sincronización por condición:*      $\langle \text{await } (\text{count} > 0) \rangle$

Si B satisface ASV, puede implementarse como *busy waiting* o *spin loop*  
 $\text{do } (\text{not } B) \rightarrow \text{skip } \text{od} \quad (\text{while } (\text{not } B); )$

Acciones atómicas incondicionales y condicionales

# Especificación de la sincronización

**Ejemplo:** productor/consumidor con buffer de tamaño N.

*cant: int = 0;*

*Buffer: cola;*

**process Productor**

**{ while (true)**

*Generar Elemento*

*<await (cant < N); push(buffer, elemento); cant++ >*

**}**

**process Consumidor**

**{ while (true)**

*<await (cant > 0); pop(buffer, elemento); cant-- >*

*Consumir Elemento*

**}**



---

# Propiedades y Fairness

---

# Propiedades de seguridad y vida

Una *propiedad* de un programa concurrente es un atributo verdadero en cualquiera de las historias de ejecución del mismo

Toda propiedad puede ser formulada en términos de dos clases: seguridad y vida.

- ***seguridad*** (safety)
  - Nada malo le ocurre a un proceso: asegura estados consistentes.
  - Una *falla de seguridad* indica que algo anda mal.
  - Ejemplos de propiedades de seguridad: exclusión mutua, ausencia de interferencia entre procesos, *partial correctness*.
- ***vida*** (liveness)
  - Eventualmente ocurre algo bueno con una actividad: progresa, no hay deadlocks.
  - Una *falla de vida* indica que las cosas dejan de ejecutar.
  - Ejemplos de vida: *terminación*, asegurar que un pedido de servicio será atendido, que un mensaje llega a destino, que un proceso eventualmente alcanzará su SC, etc  $\Rightarrow$  *dependen de las políticas de scheduling*.

¿Que pasa con la *total correctness*?

# Fairness y políticas de scheduling

***Fairness***: trata de garantizar que los procesos tengan chance de avanzar, sin importar lo que hagan los demás

Una acción atómica en un proceso es ***elegible*** si es la próxima acción atómica en el proceso que será ejecutada. Si hay varios procesos  $\Rightarrow$  hay *varias acciones atómicas elegibles*.

Una ***política de scheduling*** determina cuál será la próxima en ejecutarse.

**Ejemplo:** Si la política es asignar un procesador a un proceso hasta que termina o se demora. ¿Qué podría suceder en este caso?

```
bool continue = true;  
co while (continue); // continue = false; oc
```

# Fairness y políticas de scheduling

***Fairness Incondicional.*** Una política de scheduling es incondicionalmente fair si toda acción atómica incondicional que es elegible eventualmente es ejecutada.

En el ejemplo anterior, RR es incondicionalmente fair en monoprocesador, y la ejecución paralela lo es en un multiprocesador.

***Fairness Débil.*** Una política de scheduling es débilmente fair si :

- (1) Es incondicionalmente fair y
- (2) Toda acción atómica condicional que se vuelve elegible eventualmente es ejecutada, asumiendo que su condición se vuelve *true* y permanece *true* hasta que es vista por el proceso que ejecuta la acción atómica condicional.

No es suficiente para asegurar que cualquier sentencia *await* elegible eventualmente se ejecuta: la guarda podría cambiar el valor (de *false* a *true* y nuevamente a *false*) mientras un proceso está demorado.



# Fairness y políticas de scheduling

***Fairness Fuerte.*** Una política de scheduling es *fuertemente fair* si:

- (1) Es incondicionalmente fair y
- (2) Toda acción atómica condicional que se vuelve elegible eventualmente es ejecutada pues su guarda se convierte en *true* con infinita frecuencia.

**Ejemplo:** ¿Este programa termina?

```
bool continue = true, try = false;  
co while (continue) { try = true; try = false; }  
  // ⟨await (try) continue = false⟩  
oc
```

No es simple tener una política que sea práctica y fuertemente fair. En el ejemplo anterior, con 1 procesador, una política que alterna las acciones de los procesos sería fuertemente fair, pero es impráctica. Round-robin es práctica pero no es fuertemente fair.