

# Programación Concurrente ATIC

## Redictado Programación Concurrente

### Clase 1



Facultad de Informática  
UNLP

# Metodología del curso

## ♦ **Horarios:**

- Teorías: miércoles de 12 a 15 (aula 14)
- Prácticas: lunes de 17 a 20 hs. (aula 2) y jueves de 17 a 20 hs. (aula 3)

## ♦ **Cursada:**

- La cursada se aprueba con un parcial práctico con dos temas independientes que cuenta con dos recuperatorios. La primera fecha se encuentra distribuida.

## ♦ **Promoción de final:**

- Rendir 2 parcialitos teóricos (en la primera fecha de cada tema del parcial práctico).
- Un examen teórico (al terminar la cursada). Se aprueba con nota mayor o igual 6, y tienen hasta la mesa de julio de 2026 para inscribirse en una mesa de final.
  - Si la nota del examen fuese mayor o igual a 4 y menor que 6, no tienen la promoción, pero pueden solicitar la realización de un trabajo (y su posterior defensa).

## ♦ **Final:** examen teórico - práctico.

# Metodología del curso

- ♦ **Comunicación:** Plataforma IDEAS ([ideas.info.unlp.edu.ar](http://ideas.info.unlp.edu.ar)).
  - Solicitar inscripción.
- ♦ **Bibliografía / material:**
  - ***Libro base:*** Foundations of Multithreaded, Parallel, and Distributed Programming. Gregory Andrews. Addison Wesley.  
([www.cs.arizona.edu/people/greg/mpdbook](http://www.cs.arizona.edu/people/greg/mpdbook)).
  - Material de lectura adicional: bibliografía, web.
    - Principles of Concurrent and Distributed Programming, 2/E. Ben-Ari. Addison-Wesley
    - An Introduction to Parallel Computing. Design and Analysis of Algorithms, 2/E. Grama, Gupta, Karypis, Kumar. Pearson Addison Wesley.
    - The little book of semaphores. Downey.  
<http://www.cs.ucr.edu/~kishore/papers/semaphores.pdf>.
  - Planteo de temas/ejercicios (recomendado hacerlos).

# Temas del curso

- ◆ **Conceptos básicos.** Concurrencia y arquitecturas de procesamiento. Multithreading, Procesamiento Distribuido, Procesamiento Paralelo.
- ◆ **Concurrencia por memoria compartida.** Procesos y sincronización. Locks y Barreras. Semáforos. Monitores. Resolución de problemas concurrentes con sincronización por MC.
- ◆ **Concurrencia por pasaje de mensajes (MP).** Mensajes asincrónicos. Mensajes sincrónicos. Remote Procedure Call (RPC). Rendezvous. Paradigmas de interacción entre procesos.
- ◆ **Lenguajes que soportan concurrencia.** Características. Similitudes y diferencias.
- ◆ **Introducción a la programación paralela.** Conceptos, herramientas de desarrollo, aplicaciones.



---

# Introducción

---

# Concurrencia

## ¿Que es?

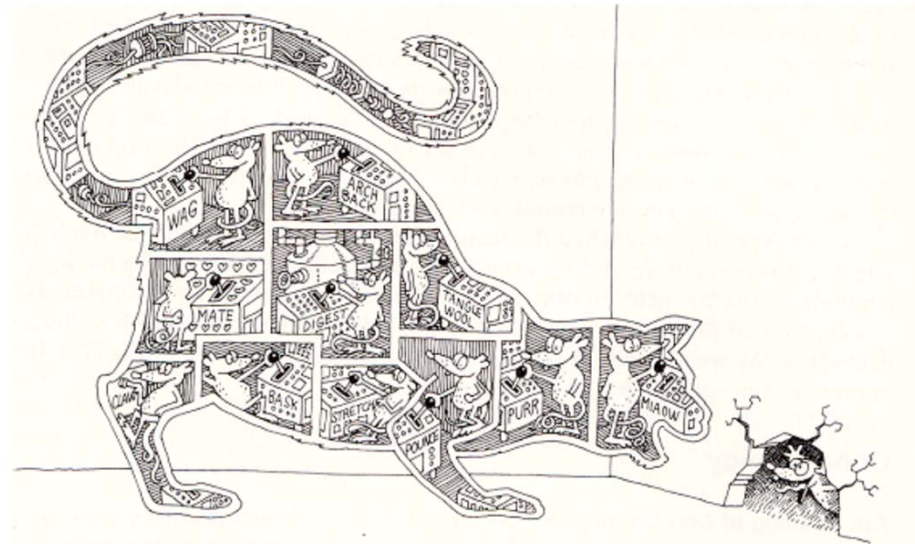
- ◆ Concurrencia es la capacidad de ejecutar múltiples actividades en paralelo o simultáneamente.
- ◆ Permite a distintos objetos actuar al mismo tiempo.
- ◆ Factor relevante para el diseño de hardware, sistemas operativos, multiprocesadores, computación distribuida, programación y diseño.

## ¿Donde está?

- ◆ Navegador Web accediendo una página mientras atiende al usuario.
- ◆ Varios navegadores accediendo a la misma página.
- ◆ Acceso a disco mientras otras aplicaciones siguen funcionando.
- ◆ Impresión de un documento mientras se consulta.
- ◆ El teléfono avisa recepción de llamada mientras se habla.
- ◆ Varios usuarios conectados al mismo sistema (reserva de pasajes).
- ◆ Cualquier objeto más o menos “inteligente” exhibe concurrencia.
- ◆ Juegos, automóviles, etc.

# Concurrencia

- ◆ Los sistemas biológicos suelen ser masivamente concurrentes: comprenden un gran número de células, evolucionando simultáneamente y realizando (independientemente) sus procesos.



- ◆ En el mundo biológico los sistemas secuenciales rara vez se encuentran.
- ◆ En algunos casos se tiende a pensar en sistemas secuenciales en lugar de concurrentes para simplificar el proceso de diseño. Pero esto va en contra de la necesidad de sistemas de cómputo cada vez más poderosos y flexibles.

# Concurrencia “natural”

- ◆ **Problema:** Desplegar cada 3 segundos un cartel ROJO.
- ◆ **Solución secuencial:**

*Programa Cartel*

Mientras (true)

Demorar (3 seg)

Desplegar cartel

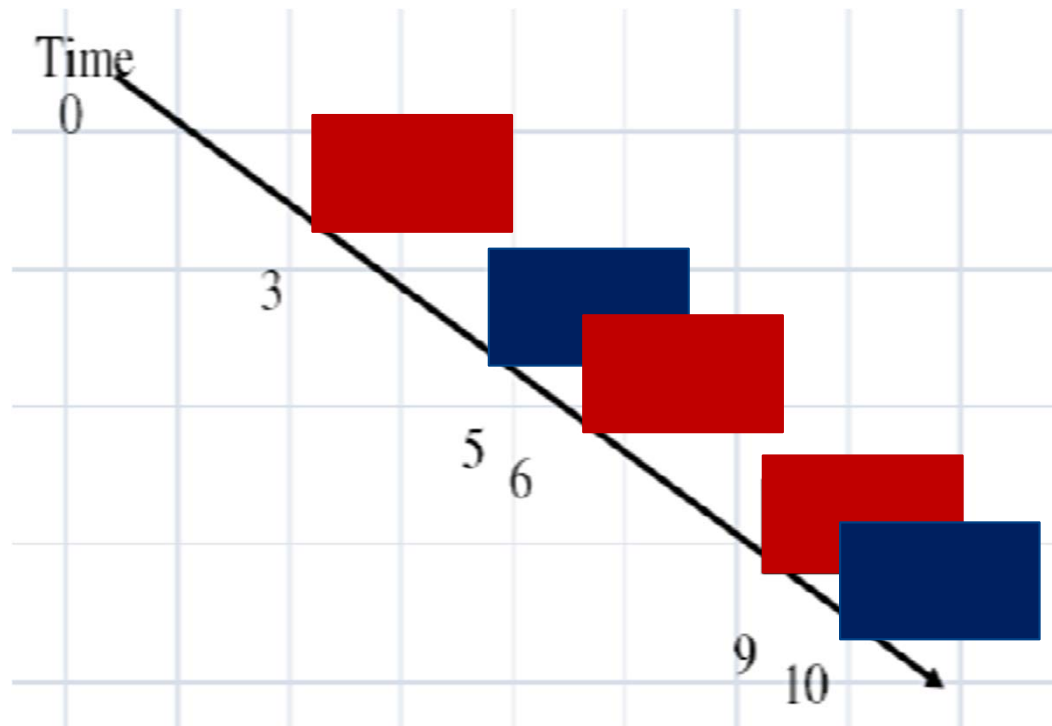
Fin mientras

*Fin programa*



# Concurrencia “natural”

- ◆ **Problema:** Desplegar cada 3 segundos un cartel ROJO y cada 5 segundos un cartel AZUL.



# Concurrencia “natural”

## *Programa Carteles*

Proximo\_Rojo = 3

Proximo\_Azul = 5

Actual = 0

Mientras (true)

Si (Proximo\_Rojo < Proximo\_Azul)

Demorar (Proximo\_Rojo – Actual)

Desplegar cartel ROJO

Actual = Proximo\_Rojo

Proximo\_Rojo = Proximo\_Rojo +3

sino

Demorar (Proximo\_Azul – Actual)

Desplegar cartel AZUL

Actual = Proximo\_Azul

Proximo\_Azul = Proximo\_Azul +5

Fin mientras

*Fin programa*

# Concurrencia “natural”

- ◆ Obliga a establecer un orden en el despliegue de cada cartel.
- ◆ Código más complejo de desarrollar y mantener.
- ◆ ¿Que pasa si se tienen más de dos carteles?
- ◆ **Más natural:** cada cartel es un elemento independiente que actúa concurrentemente con otros → *es decir, ejecutar dos o más algoritmos simples concurrentemente.*

***Programa Cartel (color, tiempo)***

Mientras (true)

Demorar (***tiempo*** segundos)

Desplegar cartel (color)

Fin mientras

***Fin programa***

- ◆ No hay un orden preestablecido en la ejecución ⇒ ***no determinismo*** (ejecuciones con la misma “entrada” puede generar diferentes “salidas”)

# ¿Por qué es necesaria la concurrencia?

- No hay más ciclos de reloj → Multicore → ¿cómo aprovecharlo?
- Aplicaciones con estructura más natural.
- Mejora en la respuesta – mejora el rendimiento
- Sistemas distribuidos

# Objetivos de los sistemas concurrentes

*Ajustar el modelo de arquitectura de hardware y software al problema del mundo real a resolver.*

*Incrementar la performance, mejorando los tiempos de respuesta de los sistemas de cómputo, a través de un enfoque diferente de la arquitectura física y lógica de las soluciones.*

## Algunas ventajas ⇒

- La velocidad de ejecución que se puede alcanzar.
- Mejor utilización de la CPU de cada procesador.
- Explotación de la concurrencia inherente a la mayoría de los problemas reales.

# Procesamiento secuencial, concurrente y paralelo

Supongamos que tenemos una unidad de procesamiento (core) para realizar  $N$  tareas.

Puede trabajar de acuerdo a dos enfoques diferentes:

- ♦ **Solución secuencial:** hace las tareas de a una a la vez, hasta no terminar una no se puede comenzar el siguiente. La solución secuencial **nos fuerza** a establecer un **estricto orden temporal**.
- ♦ **Solución concurrente (y no paralela):** dedica una parte del tiempo a cada tarea, aprovechando tiempos ociosos en la realización de cada uno de ellas  
⇒ **Concurrencia sin paralelismo de hardware**.

**Dificultades** ⇒

- Distribución de carga de trabajo.
- Necesidad de compartir recursos.
- Necesidad de esperarse en puntos clave.
- Necesidad de comunicarse.
- Necesidad de recuperar el “estado” de cada proceso al retomarlo.



# Procesamiento secuencial, concurrente y paralelo

Supongamos que tenemos  $N$  unidades de procesamiento (cores) para hacer las mismas  $N$  tareas.

- ♦ **Solución paralela (además es concurrente):** todos los cores trabajan al mismo tiempo haciendo cada uno una única tarea  $\Rightarrow$  **Concurrencia con paralelismo de hardware**. Sigue siendo concurrente, pero además es paralelo.

**Consecuencias  $\Rightarrow$**

- Menor tiempo para completar el trabajo.
- Menor esfuerzo individual.
- Paralelismo del hardware.

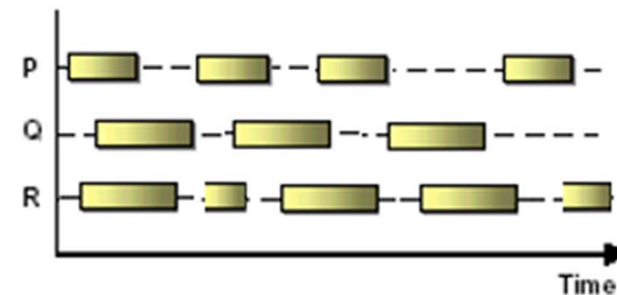
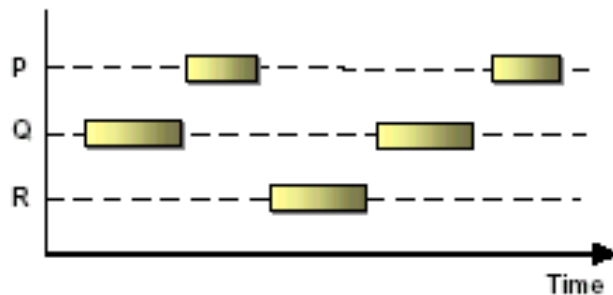
**Dificultades  $\Rightarrow$**

- Distribución de la carga de trabajo (en relación con la arquitectura y el trabajo)
- Necesidad de compartir recursos evitando conflictos.
- Necesidad de esperarse en puntos clave.
- Necesidad de comunicarse.
- Tratamiento de las fallas.

**CONCURRENCIA  $\Rightarrow$**  Concepto de software no restringido a una arquitectura particular de hardware ni a un número determinado de procesadores.

# Programa Concurrente

- Por ahora llamaremos “*Proceso*” a elemento computacional que ejecuta un único flujo de control en forma secuencial: ejecuta una instrucción y cuando esta finaliza ejecuta la siguiente.
- **Programa Concurrente:** múltiples procesos que trabajan “simultáneamente” para resolver un mismo problema y/o utilizar recursos compartidos.





# Programa Concurrente

- Un programa concurrente puede tener  $N$  *procesos* habilitados para ejecutarse concurrentemente y un *sistema concurrente* puede disponer de  $M$  *elementos de procesamiento* ( $M \geq 1$ ) cada uno de los cuales puede ejecutar uno o más procesos.
- Cuando hay más de un elemento de procesamiento ( $M > 1$ ) además de *Sistema Concurrente* es un *Sistema Paralelo*.
- Cuando hay sólo UN elemento de procesamiento ( $M = 1$ ) es un *Sistema Concurrente* pero NO un *Sistema Paralelo*.
  - El tiempo de CPU es compartido entre varios procesos, por ejemplo, por *time slicing*.
  - El sistema operativo controla y planifica procesos: si el slice expiró o el proceso se bloquea el sistema operativo hace *context (process) switch*.

# Posibles comportamientos de los procesos

Múltiples procesos trabajando “simultáneamente” pueden actuar de forma independiente, cooperar y/o competir entre ellos.

## Procesos independientes

- Relativamente raros.
- Poco interesante.

## Competencia

- Típico en Sistemas Operativos y Redes.
- Compiten por el uso de recursos compartidos.

## Cooperación

- Los procesos colaboran para resolver una tarea común.
- Sincronización y comunicación.

# Diferencia entre procesos e hilos

- **Procesos:** Cada proceso tiene su propio espacio de direcciones y recursos.
- **Procesos livianos, threads o hilos:**
  - Proceso “liviano” que tiene su propio contador de programa y su pila de ejecución, pero no controla el “contexto pesado” (por ejemplo, las tablas de página).
  - Todos los hilos de un proceso comparten el mismo espacio de direcciones y recursos (los del proceso).
  - El programador o el lenguaje deben proporcionar mecanismos para evitar interferencias.
  - La concurrencia puede estar provista por el lenguaje (Java) o por el Sistema Operativo (C/POSIX).



---

# Conceptos básicos de concurrencia

---

# Conceptos básicos de concurrencia

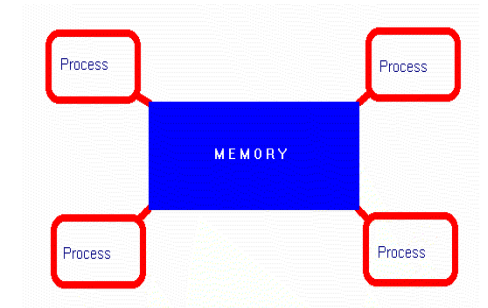
## Comunicación entre procesos

La comunicación entre procesos concurrentes indica el modo en que se organizan y transmiten datos entre tareas concurrentes. Esta organización requiere especificar **protocolos** para controlar el progreso y la corrección. Los procesos se **COMUNICAN**:

- Por *Memoria Compartida*.
- Por *Pasaje de Mensajes*.

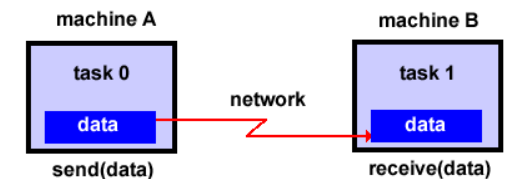
- **Memoria compartida**

Los procesos intercambian información sobre la memoria compartida o actúan coordinadamente sobre datos residentes en ella. Se debe asegurar de no operar simultáneamente sobre la memoria compartida.



- **Pasaje de mensajes**

Es necesario establecer un **canal** (lógico o físico) para transmitir información entre procesos. Se debe tener un protocolo adecuado. Para que la comunicación sea efectiva los procesos deben “saber” cuándo tienen mensajes para leer y cuando deben transmitir mensajes.



# Conceptos básicos de concurrencia

## Sincronización entre procesos

La **sincronización** es la posesión de información acerca de otro proceso para coordinar actividades. Los procesos se sincronizan:

- Por *exclusión mutua*.
- Por *condición*.

- **Sincronización por exclusión mutua**

- Asegurar que sólo un proceso tenga acceso a un recurso compartido en un instante de tiempo.
- Si el programa tiene **secciones críticas** que pueden compartir más de un proceso, la exclusión mutua evita que dos o más procesos puedan encontrarse en la misma sección crítica al mismo tiempo.

- **Sincronización por condición**

- Permite bloquear la ejecución de un proceso hasta que se cumpla una condición dada.

# Conceptos básicos de concurrencia

## Interferencia

**Interferencia:** un proceso toma una acción que invalida las suposiciones hechas por otro proceso.

**Ejemplo 1:** nunca se debería dividir por 0.

```
int x, y, z;
```

```
process A1
```

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

```
process A2
```

```
{ .....  
  if (y <> 0) z = x/y;  
  .....  
}
```

**Ejemplo 2:** siempre *Público* debería terminar con valor igual a  $E1 + E2$ .

```
int Público = 0
```

```
process B1
```

```
{ int E1 = 0;  
  for i= 1..100  
    { esperar llegada  
      E1 = E1 + 1;  
      Público = Público + 1;  
    }  
}
```

```
process B2
```

```
{ int E2 = 0;  
  for i= 1..100  
    { esperar llegada  
      E2 = E2 + 1;  
      Público = Público + 1;  
    }  
}
```

# Conceptos básicos de concurrencia

## Manejo de los recursos

Uno de los temas principales de la programación concurrente es la **administración de recursos compartidos**:

- Esto incluye la asignación de recursos compartidos, métodos de acceso a los recursos, bloqueo y liberación de recursos, seguridad y consistencia.
- Una propiedad deseable en sistemas concurrentes es el equilibrio en el acceso a recursos compartidos por todos los procesos (***fairness***).
- Dos situaciones NO deseadas en los programas concurrentes son la ***inanición*** de un proceso (no logra acceder a los recursos compartidos) y el ***overloading*** de un proceso (la carga asignada excede su capacidad de procesamiento).
- Otro problema importante que se debe evitar es el ***deadlock***.



# Conceptos básicos de concurrencia

## Problema de deadlock



Dos (o más) procesos pueden entrar en *deadlock*, si por error de programación ambos se quedan esperando que el otro libere un recurso compartido. La ausencia de deadlock es una propiedad necesaria en los procesos concurrentes.

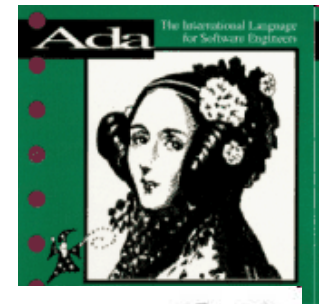
# Conceptos básicos de concurrencia

## Requerimientos para un lenguaje concurrente

Independientemente del mecanismo de comunicación / sincronización entre procesos, los **lenguajes de programación concurrente** deberán proveer primitivas adecuadas para la especificación e implementación de las mismas.

- **Requerimientos de un lenguaje de programación concurrente:**

- Indicar las tareas o procesos que pueden ejecutarse concurrentemente.
- Mecanismos de sincronización.
- Mecanismos de comunicación entre los procesos.



# Problemas asociados con la Programación Concurrente

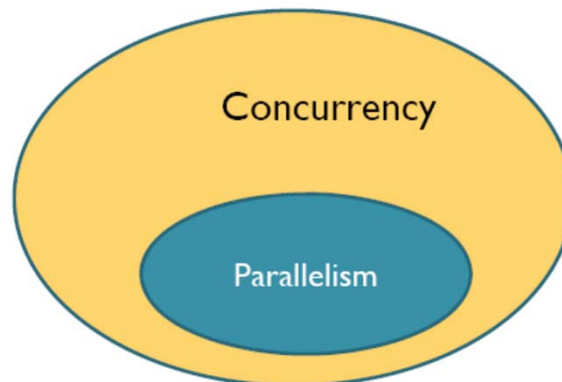
- ◆ Los procesos no son independientes y comparten recursos. La necesidad de utilizar mecanismos de exclusión mutua y/o sincronización agrega complejidad a los programas.
- ◆ Hay un **no determinismo** implícito en el interleaving de procesos concurrentes. Esto significa que dos ejecuciones del mismo programa no necesariamente son idénticas  $\Rightarrow$  *dificultad para la interpretación y debug*.
- ◆ Posible reducción de performance por **overhead** de context switch, comunicación, sincronización, ...
- ◆ Mayor tiempo de desarrollo y puesta a punto. Difícil paralelizar algoritmos secuenciales.
- ◆ Necesidad de adaptar el software concurrente al hardware paralelo para mejora real en el rendimiento.

# Conceptos básicos de concurrencia

## Concurrencia y Paralelismo

**CONCURRENCIA**  $\Rightarrow$  Concepto de software no restringido a una arquitectura particular de hardware ni a un número determinado de procesadores.  
Especificar la concurrencia implica especificar los *procesos concurrentes*, su *comunicación* y su *sincronización*.

**PARALELISMO**  $\Rightarrow$  Se asocia con la ejecución concurrente en múltiples procesadores con el objetivo principal de reducir el tiempo de ejecución.





---

# Clases de Instrucciones

---

# Clases de instrucciones

## Programación secuencial y concurrente

Un programa concurrente está formado por un conjunto de “procesos” 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.

Además de las habituales estructuras de control y asignación, utilizaremos algunas instrucciones y estructuras de control particulares para trabajar con programación concurrente.

# Clases de instrucciones

## Programación secuencial y concurrente

**skip**: termina inmediatamente y no tiene efecto sobre ninguna variable de programa.

Sentencias de alternativa múltiple:

**if**  $B1 \rightarrow S1$

$\square B2 \rightarrow S2$

....

$\square Bn \rightarrow Sn$

**fi**

Elección no determinística.

Si ninguna guarda es verdadera el *if* no tiene efecto.

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.

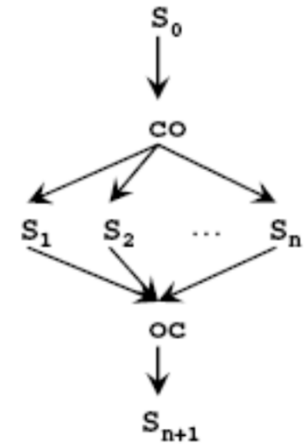
La elección es no determinística si más de una guarda es verdadera.

# 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.
- Una **acción atómica** hace una transformación de estado indivisibles (estados intermedios invisibles para otros procesos).
- Cada proceso ejecuta un conjunto de sentencias, cada una implementada por una o más acciones atómicas.
- 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, .... ☐

*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                   (1)**

**// y = 3                   (2)**

**// z = 4                   (3)**

**oc**

**(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

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*

# 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*  
 $\text{do (not B)} \rightarrow \text{skip od} \quad (\text{while (not B); })$

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 por proceso).

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

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

## Ejemplo:

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



# Fairness y políticas de scheduling

***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.

## Ejemplo:

```
bool continuar = false;  
co < await (continuar) >; sentencias //  
   sentencias; continuar = true; sentencias oc
```

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

## Ejemplo:

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

***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.

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.