


Módulo I

 **Clases 1 y 2:** *Conceptos básicos. Comunicación y sincronización. Interferencia.*

- Módulo I
 - Programa concurrente: estados, acciones e historias
 - Acciones atómicas
 - Propiedad ASV
 - Sincronización
 - Especificación
 - Propiedades
 - Scheduling y Fairness
 - Fairness incondicional
 - Fairness débil
 - Fairness fuerte
 - Ejemplo

Programa concurrente: estados, acciones e historias

El **estado**, de un programa concurrente, consiste en los valores de las variables del programa en un instante de tiempo dado. Pudiendo ser, variables explícitas (las definidas por el programador) como variables implícitas (*program counter*, *stack pointer*, etc ...).

La ejecución, del programa concurrente, puede ser vista como el intercalado de secuencias de **acciones atómicas** ejecutadas por cada proceso. Así, cada ejecución particular del programa puede ser vista como una **historia (trace)**: $S_1 \rightarrow S_2 \rightarrow \dots S_n$.

Cada ejecución de un programa concurrente, produce una historia. Inclusive, hasta en los programas más triviales, el número de historias posibles puede ser enorme. Por ejemplo; para el siguiente programa existen 2 (dos) historias posibles:

```
var x = 5; // (a)

co
    x = 0; // (b)
    x = x + 1; // (c)
oc
```

#	Historia	Valor X
1	$A \rightarrow B \rightarrow C$	$x = 1$
2	$A \rightarrow C \rightarrow B$	$x = 0$

Acciones atómicas

Una **acción atómica**, es una acción que realiza un cambio de estado indivisible. Esto significa que, cualquier estado intermedio que pueda existir en la implementación de la acción no debe ser visible a los otros procesos.

Una **acción atómica de grano fino**, es aquella implementada por el hardware (CPU) en el cual el programa concurrente es ejecutado.

En un programa secuencial, las sentencias de asignación parecen ser atómicas. Esto es porque ningún estado intermedio es visible al programa. En los programas concurrentes, no se puede afirmar lo mismo. Una sentencia de asignación, puede estar implementada por una secuencia de instrucciones máquina de grano fino. Por ejemplo, para el siguiente programa.

```
var y = 0;
var z = 0;

co
    x = y + z; // (a) load x,y; (b) sum x,z;
    y = 1; // (c)
    z = 2; // (d)
oc
```

Asumiendo que la sentencia $x = y + z$, a nivel máquina, es implementada con las instrucciones atómicas `load` y `sum`; podemos sostener que su ejecución no será atómica, ya que eventualmente podrá ser interrumpida. Esto se ve en, algunas de, las posibles historias del programa.

#	Historia	Valor X
1	$A \rightarrow B \rightarrow C \rightarrow D$	$x = 0$
2	$C \rightarrow A \rightarrow B \rightarrow D$	$x = 1$
3	$A \rightarrow C \rightarrow D \rightarrow B$	$x = 2$
4	$C \rightarrow D \rightarrow A \rightarrow B$	$x = 3$

En todas las historias, la ejecución de B antecede a la de A ($x = y + z$). Lo cual, es deseable. Excepto en la historia número 3, donde se observa una interferencia.

La **interferencia**, es un evento negativo. El cual se produce, cuando un proceso toma una acción que invalida las suposiciones hechas por otro proceso. Técnicas para evitar la interferencia, pueden ser:

1. Variables disjuntas (disjoint variables)
2. Afirmaciones debilitadas (weakened assertions)
3. Invariantes globales (global invariants)
4. Sincronización (synchronization)

Propiedad ASV

Una forma de detectar que una sentencia no posee interferencias y, por consiguiente, su ejecución parecerá atómica es verificar que la misma posea, a lo sumo, una referencia crítica.

Una **referencia crítica**, en una expresión, es la referencia a una variable que es modificada por otro proceso. Toda expresión que contenga a lo sumo una referencia crítica, cumple con la **propiedad "a lo sumo una vez"** y su ejecución parecerá atómica.

Por ejemplo, la sentencia de asignación $x = expr$ satisface la propiedad si se cumple alguna de las siguientes condiciones:

1. Si $expr$ contiene a lo sumo una referencia crítica, en cuyo caso x no podrá ser leída por otro proceso.
2. Si $expr$ no contiene referencias críticas, en cuyo caso x podrá ser leída por otro proceso.

Program Uno;		Program Dos;		Program Tres;
var x = 0;		var x = 0;		var x = 0;
var y = 0;		var y = 0;		var y = 0;
co x = x + 1; (a)		co x = y + 1; (a)		co x = y + 1; (a)
// y = y + 1; (b)		// y = y + 1; (b)		// y = x + 1; (b)
oc		oc		oc



VERIFICAR HISTORIAS Y CONSULTAR POR QUE HAY INTERFERENCIA EN 3



✓ Program Uno

- La sentencia del proceso A, no posee referencias críticas.
- La sentencia del proceso B, no posee referencias críticas.

✓ Program Dos

- La sentencia del proceso A, contiene una referencia crítica en $y + 1$. No obstante, x no es leída por otro proceso.
- La sentencia del proceso B, no posee referencias críticas.

✗ Program Tres

- La sentencia del proceso A, contiene una referencia crítica en $y + 1$ y x es leída por otro proceso.
- La sentencia del proceso B, contiene una referencia crítica en $x + 1$ y y es leída por otro proceso.

La propiedad también aplica para expresiones que, no sean de asignación.

Se asume que, en el proceso A del **program dos**, leer y antes o después de su actualización es indistinto.

Sincronización

La interacción entre los procesos, conlleva a una gran cantidad de historias posibles, como se vio en secciones anteriores. No obstante, no todas las historias posibles son deseables. Por ejemplo, una expresión que no satisfaga ASV provocará una historia no deseada.

En general, es deseable, ejecutar secuencias de instrucciones como una única acción atómica que prevengan intercalados no deseados. A estas secuencias de instrucciones las llamaremos **acciones atómicas de grano grueso**.

El rol de la **sincronización**, es la de permitir la construcción de estas acciones atómicas de grano grueso. Y así evitar la interferencia entre procesos.

1. Sincronización por **exclusión mutua (mutex)**
2. Sincronización por **condición (barrier)**

Especificación

LEER EN EL LIBRO

Propiedades

Una propiedad, en un programa concurrente, es un atributo que debe ser cierto en cada historia posible del programa. Existen 2 (dos) grandes categorías, de propiedades.

La primera, es la de **seguridad (safety)**. La cual, debe asegurar que nada malo ocurra durante la ejecución; para garantizar que el estado final del programa sea el correcto. Dos propiedades fundamentales, dentro de esta categoría, son:

1. **Exclusión mutua:** El evento negativo, en este caso, sería que 2 (dos) o más programas; accedan a la misma sección crítica en el mismo instante de tiempo.
2. **Ausencia de deadlock:** El evento negativo, en este caso, sería que un proceso quede demorado, esperando por una condición que nunca será verdadera.

La segunda, es la de **vida (liveness)**. La cual, debe asegurar que eventualmente algo bueno ocurrirá; para garantizar la correcta terminación del programa. Por ejemplo:

1. Que todo proceso, eventualmente, acceda a su sección crítica.
2. Que todo mensaje, eventualmente, sea recibido por su receptor.

Scheduling y Fairness

La mayoría de las propiedades de vida, de un programa concurrente, dependen del **fairness**. El cual, es un concepto, que se ocupa de garantizar que todos los procesos tienen chances de proceder sin importar lo que hagan los demás.

Para que los procesos avancen, acciones atómicas candidatas deben ser elegidas para su ejecución. Dicha tarea de selección, es llevada a cabo por una política de planificación (**scheduling policy**). Según la bibliografía, existen 3 (tres) grados de fairness que toda política de planificación debería proveer.



El programa es ejecutado en una arquitectura monoprocesador.

Fairness incondicional

Una política de planificación cumple con este grado si cada acción atómica, incondicional elegible, eventualmente es ejecutada.

```
var continue = true;
co while (continue); (P)
/   continue = false; (Q)
oc
```

Una política donde se conceda CPU, a los procesos, hasta que estos terminan o son demorados; no sería incondicionalmente fair. Ya que si (P) ingresa primero entonces, (Q) no tendría oportunidad de ser ejecutada.


Por el contrario, Round Robin si resulta una política incondicionalmente fair. Ya que, (Q) eventualmente podrá acceder a CPU.

Fairness débil

Una política de planificación cumple este grado si, es incondicionalmente fair y cada acción atómica condicional eventualmente es ejecutada. Asumiendo que su condición, se volverá verdadera y permanecerá así hasta ser vista por el proceso ejecutando la acción.

Fairness fuerte

Una política de planificación cumple este grado si, es incondicionalmente fair y cada acción atómica condicional eventualmente es ejecutada. Asumiendo que su condición, se volverá verdadera con infinita frecuencia hasta ser vista por el proceso ejecutando la acción.

 No es posible idear un planificador que sea práctico y fuertemente fair.

Ejemplo

¿Este programa termina?

```
var continue = true;
var try = false;

co while (continue) { try = true; try = false; } (P)
/   < await(try) continue = false > (Q)
oc
```

Con una política débilmente fair, el programa podría no terminar. Ya que, try no se mantiene verdadera hasta ser vista por (Q).

Con una política fuertemente fair, el programa podría terminar. Ya que, try se convierte en verdadera con infinita frecuencia.