I
Programación Concurrente
Clases 1 y 2: Conceptos básicos. Comunicación y sincronización. Interferencia.

Ramiro Martínez D'Elía

2021

Índice general

Capítulo 1

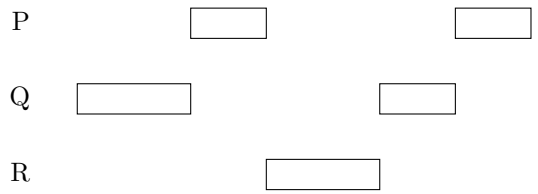
Programa concurrente

1.1. Definición

La ***concurrencia*** es la capacidad de ejecutar múltiples actividades de forma simultánea. Un ***programa concurrente***, en consecuencia, es la especificación de uno o más programas secuenciales que pueden ejecutarse de forma concurrente como procesos o tareas. Vale aclarar que, la concurrencia no implica paralelismo.

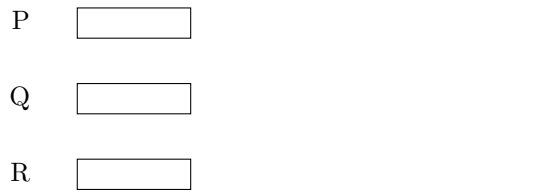
Concurrencia interleaved

- Procesamiento simultáneo lógicamente.
- Ejecución intercalada en un único CPU.
- Pseudo paralelismo.



Concurrencia simultánea

- Procesamiento simultáneo físicamente.
- Ejecución en múltiples CPU.
- Paralelismo full.



1.2. Patrones de resolución

1.2.1. Paralelismo iterativo

En el ***paralelismo iterativo*** un programa consta de un conjunto de procesos, posiblemente idénticos, que cooperarán para resolver un único problema. Donde cada proceso es un algoritmo iterativo que, trabaja sobre un subconjunto de datos del problema.

Multiplicación de matrices (por filas)

$$C_{1,1} = A_{1,n} x B_{n,1} = A_{1,1} x B_{1,1} + A_{1,2} x B_{2,1} + A_{1,3} x B_{3,1} \tag{1.1}$$

$$C_{1,2} = A_{1,n} x B_{n,2} = A_{1,1} x B_{1,2} + A_{1,2} x B_{2,2} + A_{1,3} x B_{3,2} \tag{1.2}$$

$$A \begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,1} & a_{2,2} & a_{2,3} \\ a_{3,1} & a_{3,2} & a_{3,3} \end{bmatrix} xB \begin{bmatrix} b_{1,1} & b_{1,2} & b_{1,3} \\ b_{2,1} & b_{2,2} & b_{2,3} \\ b_{3,1} & b_{3,2} & b_{3,3} \end{bmatrix} = C \begin{bmatrix} c_{1,1} & c_{1,2} & c_{1,3} \\ c_{2,1} & c_{2,2} & c_{2,3} \\ c_{3,1} & c_{3,2} & c_{3,3} \end{bmatrix} \tag{1.3}$$

```
double  a[n,n],
        b[n,n],
        c[n,n];     # Matriz resultante.

# Lanzamos 1 proceso por cada fila de A.
co  [a_row = 1 to n]

    # Iteramos sobre las columnas de B
    # En funcion del proceso id:a_row
    for [b_col = 1 to n]

        # Inicializamos la celda acumuladora de C.
        c[a_row, b_col] = 0;

        # Iteramos entre las columnas (A) y filas (B) de interes.
        for [k = 1 to n]
            c[a_row, b_col] += a[a_row, k] * b[k, b_row];
    end;
oc
```

> 💎 ¿Qué pasaría si, hay menos de n procesadores? (Ver apunte de Gemas)

1.2.2. Paralelismo recursivo

En este patrón, el programa puede descomponerse en procesos recursivos que trabajan sobre partes del conjunto total de datos. Es un claro ejemplo de la metodología ***divide y conqueror***. Por ejemplo, problema del viajante.

1.2.3. Productores y consumidores

En este patrón, se observan procesos que se comunican. Unos produciendo información y, los otros, consumiendo dicha información. Es común, organizar los procesos en forma de tubería (pipe); a través de la cual fluye la información. Por ejemplo, aplicar filtros sobre imágenes.

> 💎 ¿Podemos ordenar un vector utilizando este patrón? (Ver apunte de Gemas)

1.2.4. Clientes y servidores

En este patrón, existen procesos que; realizan pedidos y esperan por una respuesta (clientes) y, otros procesos que, esperan pedidos para procesar (servidores.)

Este tipo de soluciones, predominan en aplicaciones distribuidas.

1.2.5. Pares que interactúan

En este esquema los procesos, posiblemente idénticos, resuelven parte del problema y se comunican para avanzar en la tarea y completar el objetivo.

Similar al iterativo pero, requiere de sincronización por condición (barriers) entre pares.

1.3. 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)***: $s_1 \rightarrow s_2 \rightarrow ...s_n$.

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:

```
1  int x = 5;       # (a)
2
3  co x = 0;        # (b)
4  /  x = x + 1;    # (c)
5  oc;
```

Trace	Resultado
$A \rightarrow B \rightarrow C$	$X = 1$
$A \rightarrow C \rightarrow B$	$X = 0$

1.4. 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.

Las acciones atómicas, pueden ser ***condicionales*** o ***incondicionales***. Según tengan, o no, una condición de guarda B que demore su ejecución hasta que la misma sea verdadera.

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. Por ejemplo, para el siguiente programa.

```
1  int x = 2,
2      y = 2;
3
4  co  z = x + y;      # (1)
5  /   x = 3; y = 4;   # (2)
6  oc
```

```
1  # Proceso (1)
2  Load    PosMemX, Temp;
3  Add     PosMemY, Temp;
4  Store   Temp, PosMemZ
5
6  # Proceso (2)
7  Store   3, PosMemX;
8  Store   4, PosMemY;
```

Variable	Valor final
x	3
y	4
z	4, 5, 6, ó 7

La cantidad de valores posibles, para z, se debe a que; en algunas historias el proceso (2) invalidó las suposiciones realizadas por (1). Por ejemplo, si:

$$(1)_2 \rightarrow (2)_7 \rightarrow (1)_3 \rightarrow (1)_4 \rightarrow (2)_8 \Rightarrow z = 4$$

La suposición, en este caso, fue que el proceso (1) realizó la suma con $x = 2$. La cual quedó invalidada, cuando el proceso (2) realizó $x = 3$.

Ese evento, negativo, es conocido como ***interferencia***. 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:

- Variables disjuntas (disjoint variables)
- Afirmaciones debilitadas (weakened assertions)
- Invariantes globales (global invariants)
- Sincronización (synchronization)

1.5. Propiedad de 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 de 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	Cumple
Uno	Si
Dos	Si
Tres	No

```
1 Program Uno
2 int x = 0,
3     y = 0;
4 co  x = x + 1;
5 /   y = y + 1;
6 oc
7 End.
```

```
1 Program Dos
2 int x = 0,
3     y = 0;
4 co  x = y + 1;
5 /   y = y + 1;
6 oc
7 End.
```

```
1 Program Tres
2 int x = 0,
3     y = 0;
4 co  x = y + 1;
5 /   y = x + 1;
6 oc
7 End.
```

1.6. Sentencia Await

Si una expresión o sentencia de asignación, no cumplen con la propiedad ASV será necesario que la ejecutemos de forma atómica. Aunque, en general, necesitaremos ejecutar secuencias de instrucciones como una única acción atómica.

En ambos casos, necesitaremos utilizar algún mecanismo de sincronización que nos permita construir ***acciones atómicas de grano grueso***. Las cuales, no son más que una secuencia de acciones atómicas de grano fino que parecen ser indivisibles.

Podemos especificar acciones atómicas, a través de parentesis angulares $\langle$ y $\rangle$. Como también, especificar sincronización a través de la sentencia ***await***:

```
1 <await(B) S;>
```

- B es una condición booleana que, especifica una condición de demora.
- S es un conjunto de setencias que eventualmente terminarán, y que su estado interno no será visible por otros procesos.

Por ejemplo, en el siguiente programa; B demora hasta que x contenga un valor positivo entonces, decrementa x.

```
1 <await(x > 0) a = x-1;>
```

Con la sentencia ***await*** podemos construir acciones atómicas de grano grueso arbitrarias. Por ejemplo, el programa anterior es un ejemplo de la operación P de un semáforo a. No obstante, es complejo implementar esta sentencia en su forma general.

La forma general del ***await*** especifica ambas formas de sincronización (***exlusión mutua*** y ***por condicón***). No obstante podemos utilizar su abreviación para, especificarlas por separado.

Para especificar sincronización por ***exlusión mutua*** podemos emplear $\langle$ ***s*** $\rangle$. El siguiente ejemplo, incrementa a x e y de forma atómica:

```
1 <x = x + 1; y = y + 1;>
```

La acción anterior, es una ***acción atómica incondicional***. Ya que; no posee una guarda.

Para especificar sincronización por ***por condicón***, podemos emplear $\langle$ ***B*** $\rangle$. El siguiente ejemplo, demora la ejecución del proceso hasta que $count > 0$:

```
<await(count > 0)>
```

La acción anterior, es una ***acción atómica condicional***. Ya que; posee una guarda.

Por último, si la condicón de guarda satisface ASV. Podremos implementar el ***await*** mediante ***busy waiting*** o ***spin loops***.

```
BusyWaiting
    do (not B)
        skip;
    od
End.
```

```
SpinLoop
    while (not B);
End.
```

1.6.1. Ejemplo Productores/Consumidores

Sea el problema de procesos que producen y consumen datos, en un búffer con capacidad para n elementos. Donde, existen las siguientes restricciones:

- El productor podrá colocar datos, siempre y cuando la cola no este llena (condición).
- El consumidor podrá leer datos, siempre y cuando la cola no este vacía (condición).
- El acceso, al búffer, debe ser atómico para evitar inconsitencias (mutex).

```
int size = 0;
Queue queue;

Process Productor
    while(true)
        <await(size < N) queue.push(dato)>
    end;
End.

Process Consumidor
    while(true)
        <await(size > 0) queue.pop(dato); size--;>
    end;
End.
```

> 💎 ¿Qué pasaría si, en lugar de una cola fuese un arreglo? (Ver búffers limitados en Apunte II)

1.7. 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 correcto estado final del programa. 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 mensaje, eventualmente, sea recibido por su receptor.
2. Que todo proceso, eventualmente, acceda a su sección crítica.

1.8. 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.

> 💎 Ver el ejercicio en el apunte de Gemas.

1.8.1. Incondicionalmente Fair

Una política de planificación cumple con este grado si cada acción atómica, incondicional elegible, eventualmente es ejecutada. Por ejemplo, para el siguiente programa:

```
1 var continue = true;
2 co  while (continue); (P)
3 /   continue = false; (Q)
4 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.

1.8.2. Débilmente Fair

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.

1.8.3. Fuertemente Fair

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.

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