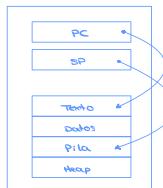


Conceptos básicos y motivación

Un programa secuencial es un conjunto de datos e instrucciones que se ejecutan siguiendo una secuencia de instrucciones. Luego, podemos entender un programa concurrente como dos o más procesos que cooperan por la realización de una determinada tarea.

Desde el punto de vista del sistema operativo un proceso se considera por:

El concepto de proceso ha de ser entendido como una entidad software abstracta, dinámica, activa, que ejecuta instrucciones y almacena diferentes estados.



d) Zona de memoria:

- Término: secuencia de instrucciones ejecutándose.
 - Datos: espacio fijo ocupado por variables globales.
 - Pila: espacio variable ocupado por variables locales.
 - Map: espacio ocupado por variables dinámicas (punteros)
- #### e) Variables internas asociadas de forma implícita:
- PC: dirección de memoria que contiene la siguiente instrucción a ejecutar.
 - SP: dirección de memoria que contiene la última posición ocupada por la pila.

En un programa concurrente existirán muchas secuencias de ejecución de instrucciones con, al menos, una línea de control de ejecución independiente para cada uno de los procesos que lo componen.

Si la plataforma de ejecución todo tiene un procesador, los múltiples flujos de control se entrelazan y forman uno solo.

Un programa concurrente así, en general, más eficiente que uno secuencial, ya que permite que los procesos que hacen E/S no monopolicen el procesador y modelen mejor un sistema real, ya que los sistemas reales suelen estar compuestos de varias actividades que se ejecutan en paralelo y cada una de ellas puede ser modelada de forma natural mediante un proceso independiente que se ejecute concurrentemente con el resto.

Podemos definir la programación concurrente como el conjunto de notaciones y técnicas de programación utilizadas para expresar el paralelismo potencial de los programas y para resolver los problemas de sincronización y comunicación que se presentan entre procesos.

Modelo abstracto de la programación concurrente

Los buenos lenguajes concurrentes deben proporcionar primitivas de programación para resolver problemas de sincronización y comunicación que son útiles en diferentes arquitecturas maquina. El modelo abstracto que vamos a introducir consigue avanzar mucho en esta dirección:

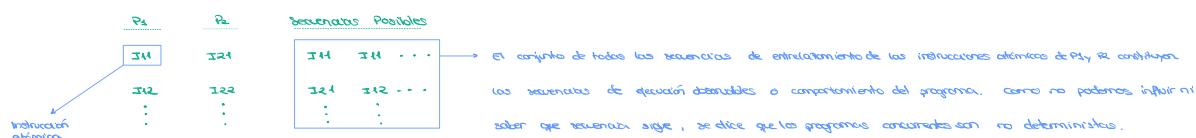
- Poder resolver la ejecución de los problemas a un nivel abstracto, sin entrar en detalles de demasiado bajo nivel.
- facilitar la notación, ya que se pueden utilizar lenguajes de programación y primitivas de alto nivel, sin necesidad de utilizar código de bajo nivel.
- Los programas desarrollados son independientes de una máquina concreta \Rightarrow son transportables.

Hipótesis del modelo abstracto de programación concurrente

1) Atómicidad y entrelazamiento de las instrucciones de los procesos:

- Normas: instrucción atómica a aquellas instrucciones que se ejecutan sin ser interrumpidas por un cambio de contexto o cualquier otra interrupción del sistema.

Supongamos el siguiente programa:



2) Coherencia de acceso concurrente a los datos: La ejecución concurrente de dos instrucciones atómicas que acceden a una misma dirección de memoria deben producir el mismo resultado y, al terminar el acceso, dejar la memoria en un estado coherente.

3) Impredictabilidad de la secuencia de instrucciones: El número de secuencias de entrelazamiento de las instrucciones atómicas de un programa concurrente es inabordable, haciendo casi imposible la depuración. Pueden aparecer errores transitorios (errores que aparecen en unas secuencias y en otras no). Por lo que es necesario usar métodos basados en la Lógica Matemática para verificar su correctitud.

“Independencia de la velocidad relativa de ejecución de los procesos”
 4) Velocidad de ejecución de los procesos: La corrección de los programas concurrentes no pueden depender de la velocidad de ejecución relativa de unos procesos con respecto a otros. Si no se cumpliera, se produciría:

	P1	P2
• Falta de trasportabilidad		
• Condiciones de carrera	a:=datos; a+=1; datos:=a;	b:=datos; b+=1; datos:=b;

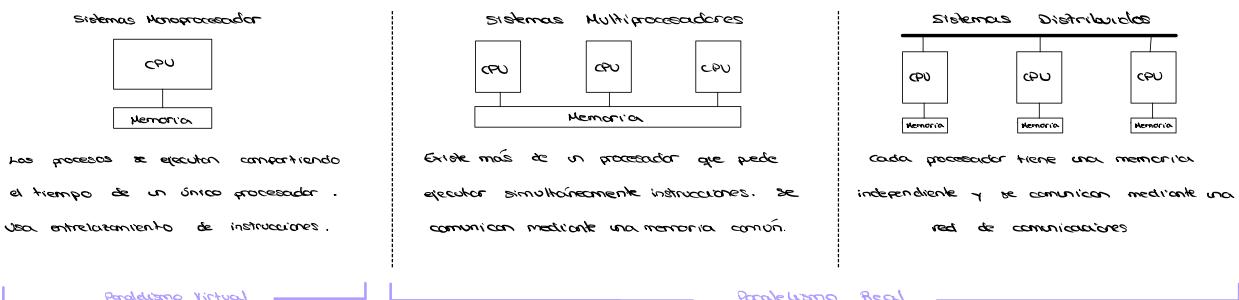
Existe una excepción, los programas para sistemas de tiempo real, donde si se hace suposición acerca del tiempo de ejecución de determinados procesos.

5) Hipótesis de progreso finito de los procesos: Todos los procesos de un programa concurrente conseguirán ejecutarse alguna vez, ya sea:

- Globalmente → Si existe al menos un proceso programado, eventualmente se permitirá la ejecución de algún proceso del programa.
- Localmente → Si un proceso comienza la ejecución de una sección de código, eventualmente completará su ejecución en tiempo finito.

Consideraciones sobre el hardware

Los sistemas implementan la concurrencia según 3 modelos:



Exclusión mutua y sincronización

La exclusión mutua asegura que una serie de sentencias del lenguaje de un proceso, compartidas con otros procesos del programa, se ejecutarán de forma atómica. No se permitiría, por tanto, que varios procesos del programa ejecuten un mismo bloque de sentencias de manera concurrente. A este bloque se le denominaría sección crítica.

Con las primitivas de programación concurrente adecuadas, si 2 o más procesos intentan ejecutar una sección crítica, sólo lo conseguirá uno al tiempo, los demás han de esperar a que dicho proceso acabe y entonces lo vuelven a intentar.

Resto: # operaciones fuera de la sección crítica

Protocolo de adquisición: En caso de que haya competencia de varios procesos, decide cuál de ellos entra en la sección crítica. El resto espera hasta que termina el proceso que entró.

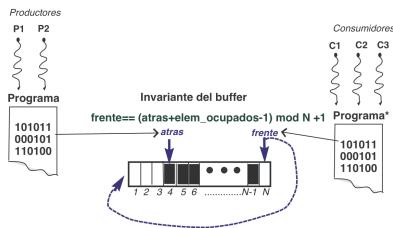
sentencias que pertenecen a la sección crítica

Protocolo de restitución: sirve para permitir a un nuevo proceso entrar en la sección crítica.

Antes hemos comentado la comunicación entre procesos dependiendo del tipo de plataforma. Esta comunicación da lugar a la necesidad de que exista una sincronización entre ellos durante la ejecución. Existen dos casos:

- Sincronización con condiciones: Consiste en detener la ejecución de un proceso hasta que se cumpla una determinada condición.
- Exclusión mutua: caso particular comentado anteriormente (un proceso no puede acceder hasta que la sección crítica quede libre y se le autorice entrar en ella).

Para entender mejor este concepto veamos el ejemplo de un buffer circular finito (consumidor - productor). Al buffer circular acceden procesos productores para insertar datos y procesos consumidores para retirarlos. El objetivo de la estructura de datos buffer es la de “desacoplar” la ejecución de los procesos de tipo productor y consumidor. Por ejemplo, si inicialmente se ejecutan una tanda de productores, pero todavía no hay consumidores, entonces se quedan temporalmente en el buffer, a la espera de ser retirados.



En este ejemplo, la exclusión mutua se utiliza para que un productor y un consumidor no accedan al buffer al mismo tiempo, ya que esto produciría una condición de carrera.

La sincronización con condiciones se utilizará para que un mensaje introducido no sea sobrescrito hasta ser leído, o bien evitar la inserción de datos en un buffer lleno o la extracción en uno vacío.

Desde el punto de vista del modelo abstracto, el papel de la sincronización consiste en restringir el conjunto de todas las posibles secuencias de ejecución a sólo aquellas que pueden considerarse correctas desde el punto de vista de las propiedades que ha de cumplir el programa.

creación de procesos

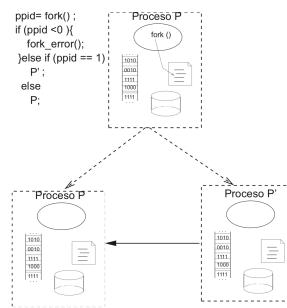
La declaración de procesos puede ser:

- **Estatística:** se declara un número fijo de procesos que se activan cuando se inicia la ejecución del programa.
- **Dinámica:** se especifica un número variable de procesos, que se activan en cualquier momento. Con esta modalidad se pueden desactivar los procesos que no interesan durante la ejecución del programa.

Creación mediante ramificación

Es una forma no estructurada de creación de procesos, empleada por los s.o. UNIX y Linux.

- **Instrucción de ramificación (fork()):** La instrucción fork() ramifica el flujo de control del proceso o programa que la llama. Como resultado, la función llamada puede comenzar a ejecutarse como un proceso concurrente independiente, que entrelazará sus instrucciones con el resto del programa.
- **Instrucción de unión (join()):** Permite que un proceso P, mediante la llamada a join(), espere a que otro proceso P' termine. P es el proceso que invoca la operación de unión y P' el proceso objetivo. Al finalizar la llamada, se puede estar seguro que el proceso objetivo ha finalizado con seguridad. La ventaja que tiene este mecanismo, frente a que P realice una espera ad infinitum hasta que P' termine, es que el proceso que hace la llamada no consume ciclos del procesador durante dicha espera.



Los ventajos de la ramificación frente a otros mecanismos de creación de procesos consiste en una creación práctica y potente (dinámica y no estructurada). Como inconveniente está la falta de estructuración, lo que dificulta la verificación y depuración.

La sentencia COBEGIN - COEND

Es una sentencia estructurada de creación de procesos. La sentencia COBEGIN inicia la ejecución concurrente de las instrucciones, nombres de procesos, llamadas a procedimientos, etc...

COBEGIN : S1,S2,...,S_n COEND;

La instrucción siguiente a COEND sólo se ejecutará cuando todas las sentencias componentes, S₁, ..., S_n, hayan terminado.

Propiedades de los sistemas concurrentes

Existen dos propiedades fundamentales:

- **Seguridad:** Afirma que ninguna ejecución incluida en el comportamiento del sistema puede llegar a entrar en un estado prohibido (no deseado).
- **Expresión determinadas condiciones** que se han de cumplir durante la ejecución.
 - **Problema de la exclusión-mutua :** 2 procesos no pueden entrar a la vez a la sección crítica.
 - **Problema del productor - consumidor :** El consumidor no puede leer datos de un buffer vacío, ni un productor introducirlos en uno lleno.
 - **Interbloqueo:** Un conjunto de procesos mantienen sus procesadores al mismo tiempo que intentan escribir datos en memoria, la memoria se llena y no hay ningún procesador disponible para escribir en disco.

- vivacidad: Afirma que el sistema finalmente entrará en un estado deseado.
- problema exclusión mutua: Si un proceso desea entrar en la sección crítica, no puede estar esperando por siempre, alguna vez conseguirá entrar.
- problema del productor-consumidor: Un proceso que quiere introducir o eliminar datos del buffer, lo conseguirá en un tiempo no finito.

Si incumplimiento de esta regla provoca la inactividad del sistema concurrente. Se dice que un proceso sufre inactivación si es indefinidamente bloqueado por los otros y nunca consigue realizar su tarea.

Verificación

Herramientas verificación de software a la demostración de que un código es correcto, es decir, que satisface todas las propiedades esperadas con anterioridad. Existen varios métodos, ya que realizar diferentes ejecuciones del programa solo permite considerar un número finito y limitado de trazos, lo que no demuestra la ausencia de casos indeseados:

- Enfoque operacional: Análisis exhaustivo de todos. Se comprueba la corrección de todos los posibles trazos. Su utilidad es muy limitada cuando se trata de programas concurrentes, ya que el número de entrelazamientos crece exponencialmente con el número de instrucciones de los procesos.
- Enfoque axiomático: se define un sistema lógico formal que permite establecer propiedades de programas en base a axiomas y reglas de inferencia.

Enfoque axiomático

Se usan fórmulas lógicas, llamadas átomas o predicados, para caracterizar un conjunto de estados. Los sentencias átomicas actúan como transformadores de tales átomas.

$$\{P\} S \{Q\}$$

La interpretación de un teorema (o triple) es que si la ejecución de la sentencia S comienza en algún estado verdadero para el asserto P (precondición), entonces Q (postcondición) será verdadero en el estado resultante.

Algunas fórmulas proposicionales válidas son (tautologías):

- Leyes distributivas:
 - $P \vee (Q \wedge R) = (P \vee Q) \wedge (P \vee R)$
 - $P \wedge (Q \vee R) = (P \wedge Q) \vee (P \wedge R)$
- Leyes de De Morgan:
 - $\neg(P \wedge Q) = \neg P \vee \neg Q$
 - $\neg(P \vee Q) = \neg P \wedge \neg Q$
- Eliminación - And: $(P \wedge Q) \rightarrow P$
- Eliminación - Or: $P \rightarrow (P \vee Q)$

Axiomas y reglas de inferencia

Sirven para poder llevar a cabo las demostraciones de programas, ya que cada línea de la demostración es un axioma o se deriva de la anterior mediante una regla de inferencia.

- Axioma: Fórmulas que sabemos que son ciertas en cualquier estado del programa.
- Regla de inferencia: Para derivar fórmulas ciertas a partir de las anteriores o de otras fórmulas que se han demostrado que son ciertas previamente.

- 1) Axioma de la sentencia nula ($\{P\} \text{NULL}\{P\}$): Si el asserto era cierto antes de ejecutar esa sentencia, seguirá siendo lo mismo interpretación cuando termine.
- 2) Sustitución trivial ($\{P[x]\}$): Es el resultado de sustituir la expresión e en cualquier aparición de la variable x en P.
- 3) Axioma de asignación ($\{P[x:=s]\} = \{P\}$): Cambia solo el valor de la variable objetivo. El resto conservan los mismos valores.

$$\{r\}_{x=5} = 5 \{x=5\} = \{P\} \Leftrightarrow \text{en aviso que } P_{x=5} = \{x=5\}_5 = \{s=s\} = r$$

- 4) Regla de consecuencia (1): $\frac{P \wedge Q \wedge R}{P \wedge R}$
- 5) Regla de consecuencia (2): $\frac{P \rightarrow Q, P \rightarrow R}{P \rightarrow Q \wedge R}$
- 6) Regla de composición: $\frac{P \wedge Q, P \wedge R}{P \wedge Q \wedge R}$
- 7) Regla del IF: $\frac{P \wedge A \wedge B \wedge C \wedge D \rightarrow E}{P \text{ if } B \text{ then } C \text{ else } D \wedge E}$
- 8) Regla de la iteración: $\frac{\text{while } B \text{ do } C \text{ enddo } M \wedge P}{P}$

* El asserto $\{P\}$ considera a todos los estados del programa (precondición más débil)

* El asserto $\{P\}$ no se cumple por parte de ningún punto del programa (precondición más fuerte)

Dentro tenemos $\langle \dots \rangle$ a una acción elemental.

Regla de interferencia de la composición concurrente de sentencias

Si $\{P_1\} \sqcap \{P_2\}$ no interfieren, entonces se puede componer el triple:

```

    {P1 A ... A Pn}1
    COBEGIN
    S1 || ... || Sn
    COEND
    {Q1 A ... A Qn}2
  
```

Si los asertos de los procesos no se invalidan entre sí, entonces su composición concurrente transforma la conjunción de las precondiciones en la conjunción de sus postcondiciones.

- Regla de no interferencia: la precondición de la sentencia `COBEGIN` ha de ser equivalente a la conjunción de las precondiciones de las sentencias componentes. Análogo con `COEND`.

```

    {x==0}
    -- cobegin
    {y==0 V y==2}   {y==0 V y==1}
    <x=y+1>      <y=x+2>
    {x==2 V x==3}  {x==2 V x==3}
    -- coend
    {x==2}
  
```

Aplicando la regla de no interferencia tendremos:

- $(x==0 \vee x==2) \wedge (x==0 \vee x==1) \equiv (x==0)$
- $(x==2 \vee x==3) \wedge (x==2 \vee x==3) \equiv (x==2)$

Llamaremos invariante global a las variables globales del programa que se demuestra cierto en el estado inicial de cada proceso y se mantiene cierto ante cualquier asignación dentro de los procesos.

GIIM. Relación de problemas-Grupo A. Tema 1. 16/09/2021

1. Considerar el siguiente fragmento de programa para 2 procesos P1 y P2: Los dos procesos pueden ejecutarse a cualquier velocidad. ¿Cuáles son los posibles valores resultantes para la variable x? Suponer que x debe ser cargada en un registro para incrementarse y que cada proceso usa un registro diferente para realizar el incremento.

```
{ variables compartidas }
var x : integer := 0 ;
Process P1;
    var i: integer;
begin
    for i:= 1 to 2 do begin
        x:= x + 1;
    end
end
Process P2;
    var j: integer;
begin
    for i:= 1 to 2 do begin
        x:= x + 1;
    end
end
```

2. ¿Cómo se podría hacer la copia del fichero f en otro g, de forma concurrente, utilizando la instrucción concurrente *cobegin-coend*? Para ello, suponer que:

- Los archivos son una secuencia de ítems de un tipo arbitrario T, y se encuentran ya abiertos para lectura (f) y escritura (g). Para leer un ítem de f se usa la llamada a función leer(f) y para saber si se han leído todos los ítems de f, se puede usar la llamada fin(f) que devuelve verdadero si ha habido al menos un intento de leer cuando ya no quedan datos. Para escribir un dato x en g se puede usar la llamada a procedimiento escribir(g,x).
- El orden de los ítems escritos en g debe coincidir con el de f
- Dos accesos a dos archivos distintos pueden solaparse en el tiempo

3. Construir, utilizando las instrucciones concurrentes *cobegin-coend* y *fork-join*, programas concurrentes que se correspondan con los grafos de precedencia que se muestran a continuación:

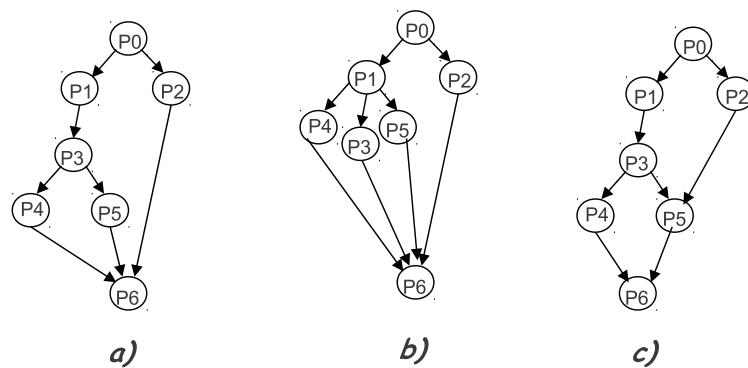


Figura 1: Grafos de sincronización con forks y joins.

4. Dados los siguientes fragmentos de programas concurrentes, obtener sus grafos de precedencia asociados:

```

begin
  P0 ;
  cobegin
    P1 ;
    P2 ;
    cobegin
      P3 ; P4 ; P5 ; P6 ;
    coend
    P7 ;
  coend
P8;
end

begin
  P0 ;
  cobegin
    begin
      cobegin
        P1; P2;
      coend
      P5;
    end;
    begin
      cobegin
        P3; P4;
      coend;
      P6;
    end;
  coend;
  P7;
end;

```

5. Suponer un sistema de tiempo real que dispone de un captador de impulsos conectado a un contador de energía eléctrica. La función del sistema consiste en contar el número de impulsos producidos en 1 hora (cada Kwh consumido se cuenta como un impulso) e imprimir este número en un dispositivo de salida. Para ello se dispone de un programa concurrente con 2 procesos: un proceso acumulador (lleva la cuenta de los impulsos recibidos) y un proceso escritor (escribe en la impresora). En la variable común a los 2 procesos n se lleva la cuenta de los impulsos. El proceso acumulador puede invocar un procedimiento Espera_impulso para esperar a que llegue un impulso, y el proceso escritor puede llamar a Espera_fin_hora para esperar a que termine una hora. El código de los procesos de este programa podría ser el siguiente:

```

{ variable compartida: }

var n : integer; { contabiliza impulsos }

begin
  while true do begin
    Espera_impulso();
    < n := n+1 > ; { (1) }
  end
end

process Escritor ;
begin
  while true do begin
    Espera_fin_hora();
    write( n ) ; { (2) }
    < n := 0 > ; { (3) }
  end
end

```

En el programa se usan sentencias de acceso a la variable n encerradas entre los símbolos < y >. Esto significa que cada una de esas sentencias se ejecuta en exclusión mutua entre los dos

procesos, es decir, esas sentencias se ejecutan de principio a fin sin entremezclarse entre ellas. Supongamos que en un instante dado el acumulador está esperando un impulso, el escritor está esperando el fin de una hora, y la variable n vale k. Después se produce de forma simultánea un nuevo impulso y el fin del periodo de una hora.

Obtener las posibles secuencias de interfolicación de las instrucciones (1),(2), y (3) a partir de dicho instante, e indicar cuales de ellas son correctas y cuales incorrectas (las incorrectas son aquellas en las cuales el impulso no se contabiliza).

6. Supongamos un programa concurrente en el cual hay, en memoria compartida dos vectores a y b de enteros y con tamaño par, declarados como sigue:

```
var a,b : array[1..2*n] of integer ; { n es una constante predefinida }
```

Queremos escribir un programa para obtener en b una copia ordenada del contenido de a (nos da igual el estado en que queda a después de obtener b). Para ello disponemos de la función Sort que ordena un tramo de a (entre las entradas s y t, ambas incluidas). También disponemos la función Copiar, que copia un tramo de a (desde s hasta t) en b (a partir de o).

```
procedure Sort( s,t : integer ) ;
  var i, j : integer ;
begin
  for i := s to t do
    for j:= s+1 to t do
      if a[i] < a[j] then
        swap( a[i], b[j] ) ;
end
procedure Copiar( o,s,t : integer ) ;
  var d : integer ;
begin
  for d := 0 to t-s do
    b[o+d] := a[s+d] ;
end
```

El programa para ordenar se puede implementar de dos formas:

- Ordenar todo el vector a, de forma secuencial con la función Sort, y después copiar cada entrada de a en b, con la función Copiar
- Ordenar las dos mitades de a de forma concurrente, y después mezclar dichas dos mitades en un segundo vector b (para mezclar usamos un procedimiento Merge).

A continuación vemos el código de ambas versiones:

```
procedure Secuencial() ;
var i : integer ;
begin
Sort( 1, 2*n ); { ordena a }
Copiar( 1, 2*n ); { copia a en b }
end
procedure Concurrente() ;
begin
cobegin
Sort( 1, n );
Sort( n+1, 2*n );
coend
Merge( 1, n+1, 2*n );
end
```

El código de Merge se encarga de ir leyendo las dos mitades de a, en cada paso, seleccionar el menor elemento de los dos siguientes por leer (uno en cada mitad), y escribir dicho menor elemento en la siguiente mitad del vector mezclado b. El código es el siguiente:

```

procedure Merge( inferior, medio, superior: integer ) ;
  { siguiente posicion a escribir en b }
  var escribir : integer := 1 ;
  { siguiente pos. a leer en primera mitad de a }
  var leer1 : integer := inferior ;
  { siguiente pos. a leer en segunda mitad de a }
  var leer2 : integer := medio ;
begin
  { mientras no haya terminado con alguna mitad }
  while leer1 < medio and leer2 <= superior do begin
    if a[leer1] < a[leer2] then begin { minimo en la primera mitad }
      b[escribir] := a[leer1] ;
      leer1 := leer1 + 1 ;
    end else begin { minimo en la segunda mitad }
      b[escribir] := a[leer2] ;
      leer2 := leer2 + 1 ;
    end
    escribir := escribir+1 ;
  end
  { se ha terminado de copiar una de las mitades,
  copiar lo que quede de la otra }
  if leer2 > superior then
    { copiar primera } Copiar( escribir, leer1, medio-1 );
  else Copiar( escribir, leer2, superior ); { copiar segunda }
end

```

Llamaremos $T_s(k)$ al tiempo que tarda el procedimiento Sort cuando actua sobre un segmento del vector con k entradas. Suponemos que el tiempo que (en media) tarda cada iteración del bucle interno que hay en Sort es la unidad (por definición). Es evidente que ese bucle tiene $\frac{k(k-1)}{2}$ iteraciones, luego:

$$T_s(k) = \frac{k(k-1)}{2} = 1/2k^2 - 1/2k$$

El tiempo que tarda la versión secuencial sobre $2n$ elementos (llamaremos S a dicho tiempo) será evidentemente $T_s(2n)$, luego:

$$S = T_s(n) = 1/2(2n)^2 - 1/2(2n) = 2n^2 - n$$

con estas definiciones, calcula el tiempo que tardará la versión paralela, en dos casos:

- (a) Las dos instancias concurrentes de Sort se ejecutan en el mismo procesador (llamamos P1 al tiempo que tarda).
 - (b) Cada instancia de Sort se ejecuta en un procesador distinto (lo llamamos P2)
- Escribe una comparación cualitativa de los tres tiempos ($S, P1$ y $P2$). Para esto, hay que suponer que cuando el procedimiento Merge actua sobre un vector con p entradas, tarda p unidades de tiempo en ello, lo cual es razonable teniendo en cuenta que en esas circunstancias Merge copia p valores desde a hacia b. Si llamamos a este tiempo $T_m(p)$, podemos escribir $T_m(p) = p$

7. Supongamos que tenemos un programa con tres matrices (a,b y c) de valores flotantes declaradas como variables globales. La multiplicación secuencial de a y b (almacenando el resultado en c) se puede hacer mediante un procedimiento MultiplicacionSec declarado como aparece aquí:

```
var a, b, c : array[1..3,1..3] of real ;
procedure MultiplicacionSec()
  var i,j,k : integer ;
  begin
    for i := 1 to 3 do
      for j := 1 to 3 do begin
        c[i,j] := 0 ;
        for k := 1 to 3 do
          c[i,j] := c[i,j] + a[i,k]*b[k,j] ;
      end
    end
  end
```

Escribir un programa con el mismo fin, pero que use 3 procesos concurrentes. Suponer que los elementos de las matrices a y b se pueden leer simultáneamente, así como que elementos distintos de c pueden escribirse simultáneamente.

8. Un trozo de programa ejecuta nueve rutinas o actividades (P1, P2, . . . , P9), repetidas veces, de forma concurrentemente con cobegin coend (ver trozo de código), pero que requieren sincronizarse según determinado grafo (ver la figura):

```
while true do
cobegin
P1 ; P2 ; P3 ;
P4 ; P5 ; P6 ;
P7 ; P8 ; P9 ;
coend
```

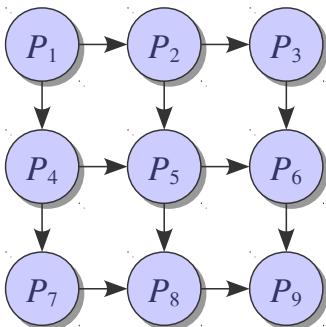


Figura 2: Grafo de sincronización de actividades.

Supón que queremos realizar la sincronización indicada en el grafo, usando para ello llamadas desde cada rutina a dos procedimientos (`EsperarPor` y `Acabar`). Se dan los siguientes hechos:

- El procedimiento `EsperarPor(i)` es llamado por una rutina cualquiera (la número k) para esperar a que termine la rutina número i, usando espera ocupada. Por tanto, se usa por la rutina k al inicio para esperar la terminación de las otras rutinas que corresponda según el grafo.
- El procedimiento `Acabar(i)` es llamado por la rutina número i, al final de la misma, para indicar que dicha rutina ya ha finalizado.

- Ambos procedimientos pueden acceder a variables globales en memoria compartida.
- Las rutinas se sincronizan únicamente y exclusivamente mediante llamadas a estos procedimientos, siendo la implementación de los mismos completamente transparente para las rutinas.

Escribe una implementación de EsperarPor y Acabar (junto con la declaración e inicialización de las variables compartidas necesarias) que cumpla con los requisitos dados.

9. En el problema 8 los procesos P1, P2, . . . , P9 se ponen en marcha usando cobegin/coend. Escribe un programa equivalente, que ponga en marcha todos los procesos, pero que use declaración estática de procesos, usando un vector de procesos P, con índices desde 1 hasta 9, ambos incluidos. El proceso P[n] contiene una secuencia de instrucciones desconocida, que llamamos Sn, y además debe incluir las llamadas necesarias a Acabar y EsperarPor (con la misma implementación que antes) para lograr la sincronización adecuada. Se incluye aquí una plantilla:

```
Process P[ n : 1..9 ]
begin
    .... { esperar (si es necesario) a los procesos que corresponda }
    Sn ; { sentencias específicas de este proceso (desconocidas) }
    .... { senalar que hemos terminado }
end
```

10. Para los siguientes fragmentos de código, obtener la *poscondición* adecuada para convertirlo en un *triple* demostrable con la Lógica de Programas:

- $\{i < 10\} i = 2 * i + 1 \{ \ }$
- $\{i > 0\} i = i - 1; \{ \ }$
- $\{i > j\} i = i + 1; j = j + 1 \{ \ }$
- $\{\text{falso}\} a = a + 7; \{ \ }$
- $\{\text{verdad}\} i = 3; j = 2 * i \{ \ }$
- $\{\text{verdad}\} c = a + b; c = c/2 \{ \ }$

11. ¿Cuáles de los siguientes triples no son demostrables con la Lógica de Programas?

- $\{i > 0\} i = i - 1; \{i \geq 0\}$
- $\{x \geq 7\} x = x + 3; \{x \geq 9\}$
- $\{i < 9\} i = 2 * i + 1; \{i \leq 20\}$
- $\{a > 0\} a = a - 7; \{a > -6\}$

12. Si el triple $\{P\} C \{Q\}$ es demostrable, indicar por qué los siguientes triples también lo son (o no se pueden demostrar y por qué):

- $\{P\} C \{Q \vee P\}$
- $\{P \wedge D\} C \{Q\}$
- $\{P \vee D\} C \{Q\}$
- $\{P\} C \{Q \vee D\}$
- $\{P\} C \{Q \wedge P\}$

13. Si el triple $\{P\} C \{Q\}$ es demostrable, ¿cuál de los siguientes triples no se puede demostrar?

- (a) $\{P \wedge D\} C \{Q\}$
- (b) $\{P \vee D\} C \{Q\}$
- (c) $\{P\} C \{Q \vee D\}$
- (d) $\{P\} C \{Q \vee P\}$

14. Dado el programa `int x = 5, y = 2; cobegin <x = x + y>; <y = x * y> coend;`, obtener:

- (a) Valores finales de x e y
- (b) Valores finales de x e y si quitamos los símbolos $<>$ de instrucción atómica.

15. Comprobar si la demostración del triple $\{x \geq 2\} <x = x - 2>; \{x \geq 0\}$ interfiere con los teoremas siguientes:

- (a) $\{x \geq 0\} <x = x + 3> \{x \geq 3\}$
- (b) $\{x \geq 0\} <x = x + 3> \{x \geq 0\}$
- (c) $\{x \geq 7\} <x = x + 3> \{x \geq 10\}$
- (d) $\{y \geq 0\} <y = y + 3> \{y \geq 3\}$
- (e) $\{x \text{ es impar}\} <y = x + 1> \{y \text{ es par}\}$

16. Dado el siguiente triple:

```
{x==0}
cobegin
<x=x+a> || <x=x+b> || <x=x+c>
coend
{x==a+b+c}
```

Demostrarlo utilizando la lógica de asertos para cada una de las tres instrucciones atómicas y después que se llega a la poscondición final $x==a+b+c$ utilizando para ello la regla de la *composición concurrente* de instrucciones atómicas.

17. El siguiente triple:

$$\begin{aligned} &\{x == 0 \wedge y == 0 \wedge z == 0\} \\ &<x = z + a> \parallel <y = x + b> \\ &\{x == a\} \wedge \{y == b \vee y == a + b\} \wedge \{z == 0\} \end{aligned}$$

- (a) Es indemostrable salvo que se cumpla siempre que $a==0$
- (b) El triple anterior es demostrable para cualquier valor de las variables a o b
- (c) Es indemostrable salvo que se cumpla siempre que $b==0$
- (d) Es indemostrable salvo que se cumpla siempre que $a == 0 \wedge b == 0$

18. Suponer que $\{ \text{suma} > 1 \} \text{ suma} = \text{suma} + 4\{\text{suma} > 5\}$ es demostrable, entonces: ¿cuál de los siguientes triples es también demostrable? (indicar por qué)

- (a) $\{ \text{suma} > 2 \} \text{ suma} = \text{suma} + 4\{\text{suma} > 5\}$
- (b) $\{ \text{suma} \geq 1 \} \text{ suma} = \text{suma} + 4\{\text{suma} > 5\}$
- (c) $\{ \text{suma} > 0 \} \text{ suma} = \text{suma} + 4\{\text{suma} > 5\}$
- (d) $\{ \text{suma} > 1 \} \text{ suma} = \text{suma} + 4\{\text{suma} > 6\}$

19. Suponer que $\{x < y\} C_1\{u < v\}$ es demostrable, entonces: ¿cuál/cuáles de los siguientes triples es/son también demostrable? (indicar por qué)

- (a) $\{x \leq y\} C_1\{u < v\}$
- (b) $\{x \leq y - 2\} C_1\{u < v\}$
- (c) $\{x \leq y\} C_1\{u \leq v\}$
- (d) $\{x < y\} C_1\{u < v - 2\}$

20. Seleccionar el valor correcto de las 2 variables (x e y) después de ejecutarse el siguiente programa concurrente:

```
int x=5, y =2;
cobegin <x= x+y>; <y= x*y>; <x= x-y>; coend;
```

- (a) x==7 e y==14
- (b) x==5 e x==10
- (c) x== -7 e y==14
- (d) x== -3 e y==10

21. El siguiente código concurrente no puede ser demostrado directamente con la lógica de aserciones (pre y poscondiciones). Elegir la respuesta que explica correctamente la razón de que ocurra esto.

```
{x==0} cobegin <x=x+a>; <x=x+a> coend; {x==2*a}
(a es un valor entero positivo)
```

- (a) Porque la poscondición que se propone $\{x==2*a\}$ es falsa
- (b) Porque falta incluir la posibilidad de que el valor final de x sea también $\{x==a\}$
- (c) Porque al aplicar directamente la regla de inferencia de la *composición concurrente* utilice unas condiciones (pre y post- condiciones) demasiado débiles
- (d) Porque tengo que incluir en los asertos el valor del contador de programa de cada procesador

22. Estudiar cuáles son los valores finales de las variables x e y en el siguiente programa. Insertar los asertos adecuados entre llaves, antes y después de cada sentencia, para poder obtener una traza de demostración del programa, que incluya en su último aserto los valores finales de las variables.

- (a) $\{ \text{int } x = C1 \}$
- (b) $\{ \text{int } y = C2 \}$
- (c) $\{ x = x + y; \}$
- (d) $\{ y = x * y; \}$
- (e) $\{ x = x - y; \}$

23. Demostrar que el siguiente triple es cierto:

$$\{x == 0\}$$

cobegin

$$< x = x + 1 > || < x = x + 2 > || < x = x + 4 >$$

coend

$$\{x == 7\}$$

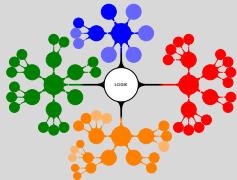
24. Dada la siguiente construcción de composición concurrente P:

cobegin

$$< x = x - 1 >; < x = x + 1 >; || < y = y + 1 >; < y = y - 1 >;$$

coend;

demonstrar que se cumple la invariancia de $\{x == y\}$, es decir, que $\{x == y\} \text{ P } \{x == y\}$; es un triple cierto.

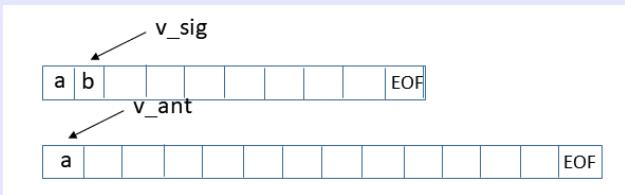


1 Los posibles valores de la variable son: $x = 2, 3$ y 4

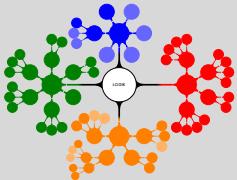
- Cada uno de los dos procesos P1, P2 hace 2 lecturas: L11, L12, L21, L22 y 2 escrituras
- Cada proceso incrementa (+1) x, 2 veces partiendo de 0: el valor final de x $\neq 2$
- Se hacen 4 incrementos de x : el valor final de x $\neq 4$

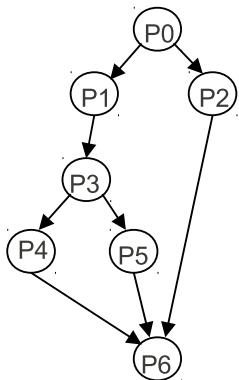
x	P1	P2	x	P1	P2	x	P1	P2
0	L11	-	0	L11	-	0	L11	-
0	-	L21	0	-	L21	1	E11	-
1	E11	-	1	E11	-	1	-	L21
1	-	E21	1	-	E21	2	-	E21
1	L12	-	1	L12	-	2	L12	-
1	-	L22	2	E12	-	3	E12	-
2	E12	-	2	-	L22	3	-	L22
2	-	E22	3	-	E22	4	-	E22

- Los datos del primer archivo han de ser escritos en el segundo conservando el orden secuencial de aparición
- La escritura de un elemento procedente del primer archivo puede solaparse en el tiempo con la lectura del siguiente
- Hay que evitar una *condición de carrera* en el acceso a la variable compartida que contenga el último dato leído



```
process Correcto ;  
var v_ant, v_sig : T ;  
begin  
    v_sig := leer(f) ;  
    while not fin(f) do begin  
        v_ant := v_sig ;  
        cobegin escribir(g,v_ant); v_sig := leer(f) ; coend  
    end  
end
```

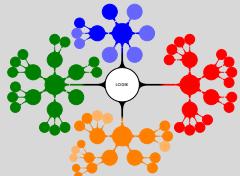




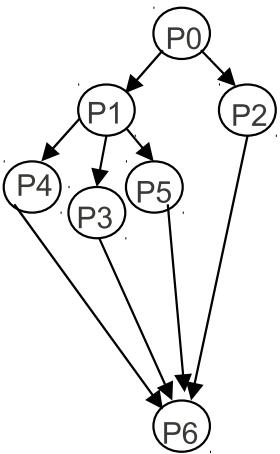
```
begin
  P0 ; fork P2 ;
  P1 ; P3 ; fork P5 ; P4 ;
  join P2 ; join P5 ;
  P6 ;
end
```

```
begin
  P0 ;
  cobegin
    begin
      P1 ; P3 ;
      cobegin P4 ; P5 ; coend
    end
    P2 ;
  coend
  P6 ;
end
```

Problemas-
introducción

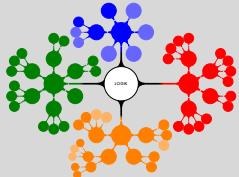


P3

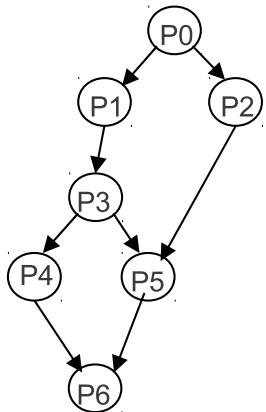


```
begin
  P0 ; fork P2 ;
  P1 ; fork P3 ; fork P5 ;
  P4
  join P2 ; join P3 ; join P5 ;
  P6 ;
end
```

```
begin
  P0 ;
  cobegin
    begin
      P1 ;
      cobegin P3 ; P4 ; P5 ; coend
    end
    P2 ;
  coend
  P6 ;
end
```



P3

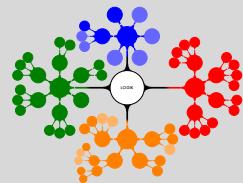


```
begin
  P0 ; fork P2 ;
  P1 ;
  P3 ; fork P4 ;
  join P2 ;
  P5;
  join P4 ;
  P6 ;
end
```

```
begin
  P0 ;
  cobegin
    begin P1 ; P3 ; end
    P2 ;
  coend
  cobegin P4 ; P5 ; coend
  P6 ;
end
```

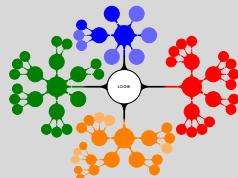
```
begin
  P0 ;
  cobegin
    begin
      P1 ; P3 ; P4 ;
    end ;
    P2 ;
  coend
  P5 ; P6 ;
end
```

Problemas-
introducción



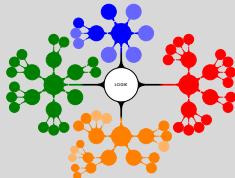
Resolución de la
primera relación de
problemas. SCD-GIIM

Manuel I. Capel
manuelcapel@ugr.es



- Suponemos una variable ficticia **OUT** que se crea como resultado de la instrucción `write(n)` (2) que contiene el valor impreso (éste pasa así a formar parte del estado)
- En el estado inicial se cumple $n == k$
- Sólo serán correctos los entrelazamientos de instrucciones atómicas del programa que sean compatibles con el estado final: $OUT + n == k + 1$
- Los posibles entrelazamientos son: (a) 1,2,3, (b) 2,1,3 y (c) 2,3,1.

(a)			(b)			(c)		
inst.	n	OUT	inst.	n	OUT	inst.	n	OUT
-	k	-	-	k	-	-	k	-
<code>n:=n+1</code>	<code>k+1</code>	-	<code>write(n)</code>	k	k	<code>write(n)</code>	k	k
<code>write(n)</code>	<code>k+1</code>	<code>k+1</code>	<code>n:= n+1</code>	<code>k+1</code>	k	<code>n:= 0</code>	0	k
<code>n:=0</code>	0	<code>k+1</code>	<code>n:=0</code>	0	k	<code>n:= n+1</code>	1	k

Problemas-
introducción

- 1 Calculo del tiempo para 2 instancias concurrentes del procedimiento **Sort** n que se ejecutan en 1 solo procesador:

$$P1 = 2 \cdot T_s(n) + T_m(2n) = (n^2 - n) + 2n = n^2 + n$$

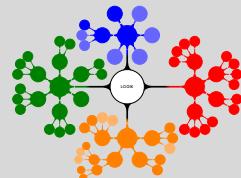
(tiempo de ejecución de la versión totalmente secuencial para ordenar 2n elementos: $S = 2 \cdot n^2 - n$

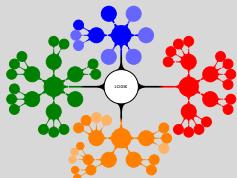
- 2 Cada instancia de **Sort** se ejecuta en un procesador distinto:

$$P2 = T_s(n) + T_m(2n) = (1/2 \cdot n^2 - 1/2 \cdot n) + 2n = 1/2 \cdot n^2 + 3/2 \cdot n$$

- Se podría parallelizar calculando de forma independiente las : filas, columnas, ..., de la matriz resultado
- Utilizamos 3 procesos concurrentes **CalcularFila (i:1..3):**

```
var a, b, c : array [1..3,1..3] of real ;
process CalcularFila[ i : 1..3 ] ;
    var j, k : integer ;
begin
    for j := 1 to 3 do begin
        c[i,j] := 0 ;
        for k := 1 to 3 do
            c[i,j] := c[i,j] + a[i,k]*b[k,j] ;
    end
end
```

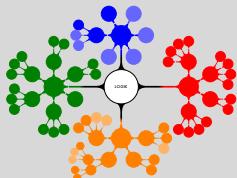




```
{ compartido entre todas las tareas }
var finalizado : array [1..9] of boolean := (false, ...,
false) ;
```

```
procedure EsperarPor( i : integer )
begin
  while not finalizado[i] do begin; end
end
```

```
procedure Acabar( i : integer )
var j : integer ;
begin  if i < 9 then
          finalizado[i] := true ;
      else for j := 1 to 9 do
          finalizado[j] := false ;
end
```



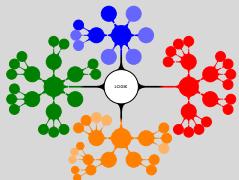
Se resuelve aplicando directamente el axioma de asignación,
basado en la sustitución textual de $\{P\}$ por $\{P\}_e^x$ en la
precondición de los triples :

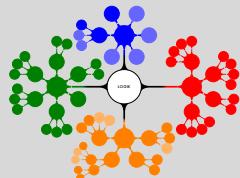
- ① $\{i < 10\} i = 2 * i + 1 \{ i < 21 \}$ puesto que:
 $\{i < 21\}_{2*i+1}^i \equiv \{2 * i + 1 < 21\} \equiv \{i < 10\}$
- ② $\{i > 0\} i = i - 1; \{ i > -1 \}$
- ③ $\{i > j\} i = i + 1; \{ i > j + 1 \} j = j + 1 \{ i > j \}$
- ④ $\{F\} a = a + 7; \{ V \}$
- ⑤ $\{V\} i = 3; \{ i == 3 \} j = 2 * i \{ j == 6 \}$
- ⑥ $\{V\} c = a + b; \{ c == a + b \} c = c/2 \{ c == \frac{a+b}{2} \}$

P11

$i, x, a \in \mathbb{Z}$

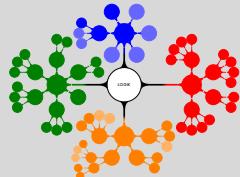
- ① $\{i > 0\} i = i - 1; \{i + 1 > 0\} \Rightarrow \{i \geq 0\}$
- ② $\{x \geq 7\} x = x + 3; \{x \geq 10\} \Rightarrow \{x \geq 9\}$
- ③ $\{i < 9\} i = 2 * i + 1; \{i < 19\} \Rightarrow \{i \leq 20\}$
- ④ $\{a > 0\} a = a - 7; \{a > -7\} \not\Rightarrow \{a > -6\}$





El triple $\{P\} \rightarrow \{Q\}$ es demostrable,

- ① $\{P\} \rightarrow \{Q \vee P\}$ también lo es por debilitamiento de la poscondición
- ② $\{P \wedge D\} \rightarrow \{Q\}$ también lo es por fortalecimiento de la precondición
- ③ $\{P \vee D\} \rightarrow \{Q\}$ No lo es porque se debilita la precondición
- ④ $\{P\} \rightarrow \{Q \vee D\}$ lo mismo que (1)
- ⑤ $\{P\} \rightarrow \{Q \wedge P\}$ No lo es porque se fortalece la poscondición

Problemas-
introducción

- ① $\{P \wedge D\} \subset \{Q\}$
- ② $\{P \vee D\} \subset \{Q\}$ No se puede demostrar porque se debilita la precondición
- ③ $\{P\} \subset \{Q \vee D\}$
- ④ $\{P\} \subset \{Q \vee P\}$

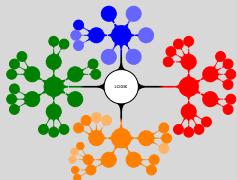
```
int x = 5, y = 2; cobegin <x = x + y>; <y = x * y>
coend;
```

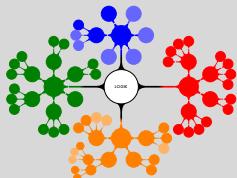
- a) Considerando operaciones atómicas (con los símbolos <,>)

- 1 $\{x == 5 \wedge y == 2\} <x = x + y>; <y = x * y>$
 $\{x == 7 \wedge y == 14\}$
- 2 $\{x == 5 \wedge y == 2\} <y = x * y>; <x = x + y>$
 $\{x == 15 \wedge y == 10\}$

- b) Sin considerar las operaciones atómicas (quitando los símbolos <,>)

- 1 Los valores de (a) y además $\{x == 7 \wedge y == 10\}$

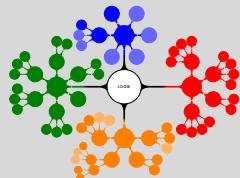




Regla no de interferencia de predicado $\{P\}$ con acción atómica $\{P \wedge \text{pre}(a)\} < a > \{P\}$ El triple

$\{x \geq 2\} < x = x - 2 >; \{x \geq 0\}$ interfiere con:

$\{x \geq 0\} < x = x + 3 > \{x \geq 3\}$	Sí	$\{x \geq 3\} < x = x - 2 > \{x \geq 1\} \not\Rightarrow \{x \geq 3\}$
$\{x \geq 0\} < x = x + 3 > \{x \geq 0\}$	No	$\{x \geq 2\} \wedge \{x \geq 0\} < x = x - 2 >; \{x \geq 0\}$
$\{x \geq 7\} < x = x + 3 > \{x \geq 10\}$	Sí	$\{x \geq 7\} < x = x - 2 > \Rightarrow \{x \geq 5\} \not\Rightarrow \{x \geq 7\}$
$\{y \geq 0\} < y = y + 3 > \{y \geq 3\}$	No	Las variables x e y son disjuntas
$\{x \text{ es impar}\} < y = x + 1 > \{y \text{ es par}\}$	No	$\{x \in 2 + 1\} \wedge \{x \geq 2\} < x = x - 2 >;$ $\{x + 2\} \in \{2 + 1\} \Rightarrow \{x \in 2 + 1\}$



$$\{x == 0\}$$

-- inits cobegin

$$\{x == 0 \vee x == b \vee x == c \vee x == b + c\}$$

$$< x = x + a >$$

||

$$\{x == a \vee x == a + b \vee x == a + c \vee x == a + b + c\}$$

$$\{x == 0 \vee x == a \vee x == c \vee x == a + c\}$$

$$< x = x + b >$$

||

$$\{x == b \vee x == b + a \vee x == b + c \vee x == a + b + c\}$$

$$\{x == 0 \vee x == b \vee x == a \vee x == a + b\}$$

$$< x = x + c >$$

$$\{x == c \vee x == c + b \vee x == c + a \vee x == a + b + c\}$$

-- inits coend

-- aplicando regla de la concurrencia

$$\{x == a + b + c\}$$

⑬ El siguiente triple:

$$\begin{aligned} & \{x=0 \wedge y=0 \wedge z=0\} \\ & \leftarrow x=z+a \quad || \quad y=x+b \\ & \{x=0 \wedge y=b \wedge z=0\} \end{aligned}$$

- a) Es demostrable salvo que se cumpla siempre que $a=0$.
- b) El triple anterior es demostrable para cualquier valores de las variables a o b .
- c) Es indemostable salvo que se cumpla siempre que $b=0$.
- d) Es indemostable salvo que se cumpla siempre que $a=0 \wedge b=0$.

$$\begin{aligned} & \{x=0 \wedge y=0 \wedge z=0\} \\ & \quad \text{-- init3 odexpr} \\ & \{x=0 \wedge y=b \wedge z=0\} \parallel \{x=0 \vee x=a \wedge y=0 \wedge z=0\} \\ & \quad \leftarrow x=z+a \quad \parallel \quad y=x+b \\ & \{x=a \wedge y=b \wedge z=0\} \parallel \{x=0 \wedge y=b \wedge z=0\} \quad \{x=a \wedge y=b \wedge z=0\} \\ & \quad \text{-- init3 odexpr} \\ & \quad \text{-- aplicando regla de la disyunción} \end{aligned}$$

$$\begin{aligned} & [(x=0 \wedge y=0 \wedge z=0) \vee (x=a \wedge y=b \wedge z=0)] \wedge [(x=0 \wedge y=b \wedge z=0) \vee \\ & \quad \forall L \quad x=a \wedge y=b \wedge z=0] = \emptyset \quad (a,b \in \mathbb{R} \setminus \{0\}) \end{aligned}$$

⑭ Suponer que $\{\text{suma} > 1 \wedge \text{suma} = \text{suma} + 4 \wedge \text{suma} > 5\}$ es demostrable, entonces: ¿cuál de los siguientes triples es también demostrable?

- a) $\{\text{suma} > 1 \wedge \text{suma} = \text{suma} + 4 \wedge \text{suma} > 5\}$
- b) $\{\text{suma} > 4 \wedge \text{suma} = \text{suma} + 4 \wedge \text{suma} > 5\}$
- c) $\{\text{suma} > 5 \wedge \text{suma} = \text{suma} + 4 \wedge \text{suma} > 5\}$
- d) $\{\text{suma} > 4 \wedge \text{suma} = \text{suma} + 4 \wedge \text{suma} > 6\}$

Vemos a analizar caso por caso:

- a) Si es demostrable, ya que la precondición es más restrictiva que la original, es decir, tenemos que $\{\text{suma} > 1 \wedge \text{suma} > 4\}$.
- b) No es demostrable ya que en la precondición tenemos que $\{\text{suma} > 1\}$, mientras que en la precondición original $\{\text{suma} > 4\}$.
- c) No es demostrable ya que la precondición original es más restrictiva que la dada en este caso ($\{\text{suma} > 0\} \neq \{\text{suma} > 1\}$, pero $\{\text{suma} > 1\} \subset \{\text{suma} > 0\}$).
- d) No es demostrable ya que la post-condición dada es más restrictiva que la original (debe ser más débil), ya que no podemos garantizar que $\text{suma} > 6$ (según nuestro enunciado, existen casos donde $\{\text{suma} < 6\}$).

Para estos ejercicios, tiene que darse que la precondición sea más estricta y la postcondición más débil.

⑮ Suponer que $\{x < y\} \subset \{x < y\}$ es demostrable, entonces: ¿cuál de los siguientes triples es también demostrable?

- a) $\{x < y \wedge x < y\}$
- b) $\{x < y - 1 \wedge x < y\}$
- c) $\{x < y \wedge x < y\}$
- d) $\{x < y \wedge x < y - 2\}$

A la igual que el ejercicio anterior, iremos analizando una a una.

- a) No es demostrable, ya que la precondición dada es más débil que la original, es decir, en este caso tenemos que el valor $y \in \{x < y\}$, mientras que en la precondición original $y \notin \{x < y\}$.
- b) Es demostrable, ya que la precondición dada es más restrictiva que la original ($\{x < y - 1\} \subset \{x < y\}$, ya que $y - 2 < y$).
- c) Ocurre lo mismo que en el caso a) (No es demostrable).
- d) No es demostrable, ya que la post-condición dada es más estricta que la original (debe ser más débil), ya que no podemos garantizar que $x < y - 2$ (según la post-condición original, existen casos donde $x - 2 \leq x < y$, los cuales no se consideran en la post-condición de este apartado).

⑯ Determinar el valor correcto de las 2 variables (x,y) después de ejecutarse el siguiente programa concurrente:

```
int x=5,y=2;
codasign c1:x=x+y;y:=x+y;x:=x-y;c2:wend;
```

- a) $x=4 \wedge y=10$
- b) $x=5 \wedge y=10$
- c) $x=-4 \wedge y=14$
- d) $x=-3 \wedge y=10$

```

+ x == 5 & y == -6 & z == 2 &
-- inits cabegin
+ x == 5 & x == 7 & x == 3 & y == 2 &
< y = x + y >
+ x == 5 & y == 10 & z == 7 & x == 7 & y == 14 & z == 14 & x == 3 & y == 6 &
+ x == 15 & y == 10 & z == 18 & x == 18 & y == 10 & z == 14 &

```

Por lo que, usando la regla de no interferencia (la conjunción de las postcondiciones de las sentencias componentes es equivalente a la postcondición de la sentencia cond) tenemos que $x == 5 \wedge y == 10$, luego, la opción correcta es la b. También puede ocurrir que si se ejecuta en el orden normal se detenga que $x == 7 \wedge y == 14$ (lo cual es la opción c).

- ② En el siguiente código concurrente no se puede demostrar directamente con lógica de assertions (fres y postcondiciones). Elegir la respuesta que explique correctamente la razón de que esto ocurra.

```

+ x == 0 cabegin < x = x + a >; < x = x + a > cond; + x == 2 * a
( a es un valor entero positivo )

```

- Porque la postcondición que se propone $+x == 2 * a$ es falsa.
- Porque falta incluir la posibilidad de que el valor final de x sea también $+x == a$.
- Aplicar directamente la regla de inferencia de la composición concurrente usando unas condiciones (fres y post-condiciones) demasiado débiles.
- Porque tiene que incluir en los asserts el valor del contador de programa de cada procedimiento.

```

+ x == 0
-- inits cabegin
+ x == 0 & y == a &
< x = x + a > < x = x + a >
+ x == a & y == 2 * a &
+ x == a & y == 2 * a &
-- inits condend

```

Aplicando ahora la regla de no interferencia la postcondición de la sentencia cond ha de ser equivalente a la conjunción de las postcondiciones de las sentencias componentes, es decir:

$$(x == a \vee x == 2a) \wedge (y == a \vee y == 2a) = (x == a \vee y == 2a) \neq (x == 2a)$$

Observamos que falta considerar la posibilidad de que el valor final de x sea también $+x == a$, luego, la respuesta correcta es la b.

C. Según el profesor, es más correcto la opción c).

- ③ Estudiar cuáles son los valores finales de las variables x e y en el siguiente programa. Intenta los assertos adecuados entre llaves, antes y después de cada sentencia, para poder obtener una traza de demostración del programa, que incluya en su último asserto los valores finales de las variables.

- int x = c1
- int y = c2
- $x = x + y$
- $y = x \cdot y$
- $x = x - y$

```

+ x == c1 & y == c2 &
-- inits cabegin
+ x == c1 & y == c2 &
-- inits condend
-- aplicamos la regla de la no interferencia
+ x == (c1 - c2)(c1 + c2) & y == c2(c1 + c2) &

```

- c) tener que aplicar la regla de la no interferencia, hemos calculado la conjunción de las postcondiciones de las sentencias componentes)

(23) Demostrar que el siguiente triplete es cierto:

```

tx==0
cobegin
<x=x+4>; <x=x+2>; <x=x+1>
coend
ty==7

```

Vamos a demostrarlo a través de la traza y viendo que está libre de interferencias:

```

tx==0
-- inits cobegin
{x==0 V x==2 V x==4 V x==6} || {x==0 V x==1 V x==3 V x==5} || {x==0 V x==1 V x==2 V x==3}
<x=x+4>           <x=x+2>           <x=x+1>
{x=2 V x=3 V x=5 V x=7} || {x=4 V x=5 V x=6 V x=7}
-- inits coend
-- Aplicamos la regla de la no interferencia
sx==7

```

A aplicar la regla de la no interferencia, observamos que la postcondición de la sentencia coend es equivalente a la conjunción de las postcondiciones de sus sentencias componentes, es decir:

$$(\text{x} == 1 \vee \text{x} == 3 \vee \text{x} == 5 \vee \text{x} == 7) \wedge (\text{x} == 2 \vee \text{x} == 3 \vee \text{x} == 5 \vee \text{x} == 7) \wedge (\text{x} == 4 \vee \text{x} == 5 \vee \text{x} == 6 \vee \text{x} == 7) \equiv (\text{x} == 7)$$

Lo mismo ocurre al ver que la precondición de la sentencia cobegin es equivalente a la conjunción de las precondiciones de sus sentencias componentes, es decir:

$$(\text{x} == 0 \vee \text{x} == 2 \vee \text{x} == 4 \vee \text{x} == 6) \wedge (\text{x} == 0 \vee \text{x} == 1 \vee \text{x} == 5) \wedge (\text{x} == 0 \vee \text{x} == 1 \vee \text{x} == 2 \vee \text{x} == 3) \equiv (\text{x} == 0)$$

luego, como la traza está libre de interferencias, es demostrable.

(24) Dada la siguiente construcción de composición concurrente P:

```

cobegin
<x=x-d>; <x=x+d>; <y=y-d>; <y=y+d>
coend

```

demostrar que se cumple la invariante de $t_{x,y}$, es decir, que $t_{x,y} = t_x + t_y$ es un triplete cierto.

Vamos a cometer desarrollando la traza. Para ello suponemos como precondición del cobegin $\{x == z\} \wedge \{y == z\}$, donde $z \in \mathbb{Z}$ son valores constantes. Entonces, tenemos:

```

{x ==  $\mathbb{Z}$ } \wedge {y ==  $\mathbb{Z}$ }
-- inits cobegin
tx==z || {y==z}
<x=x-d>           <y=y+d>
{x= $\mathbb{Z}-d$ }           {y= $\mathbb{Z}+d$ }
<x=x+d>           <y=y-d>
{x= $\mathbb{Z}$ }             {y= $\mathbb{Z}$ }
-- inits coend
{x ==  $\mathbb{Z}$ } \wedge {y ==  $\mathbb{Z}$ }

```

Luego, tenemos que tanto la x como la y no varían, por lo que basta tener $z = \mathbb{Z}$ para ver que la tripleta $\{x == y\} \wedge \{x == y\}$ es un triple cierto.