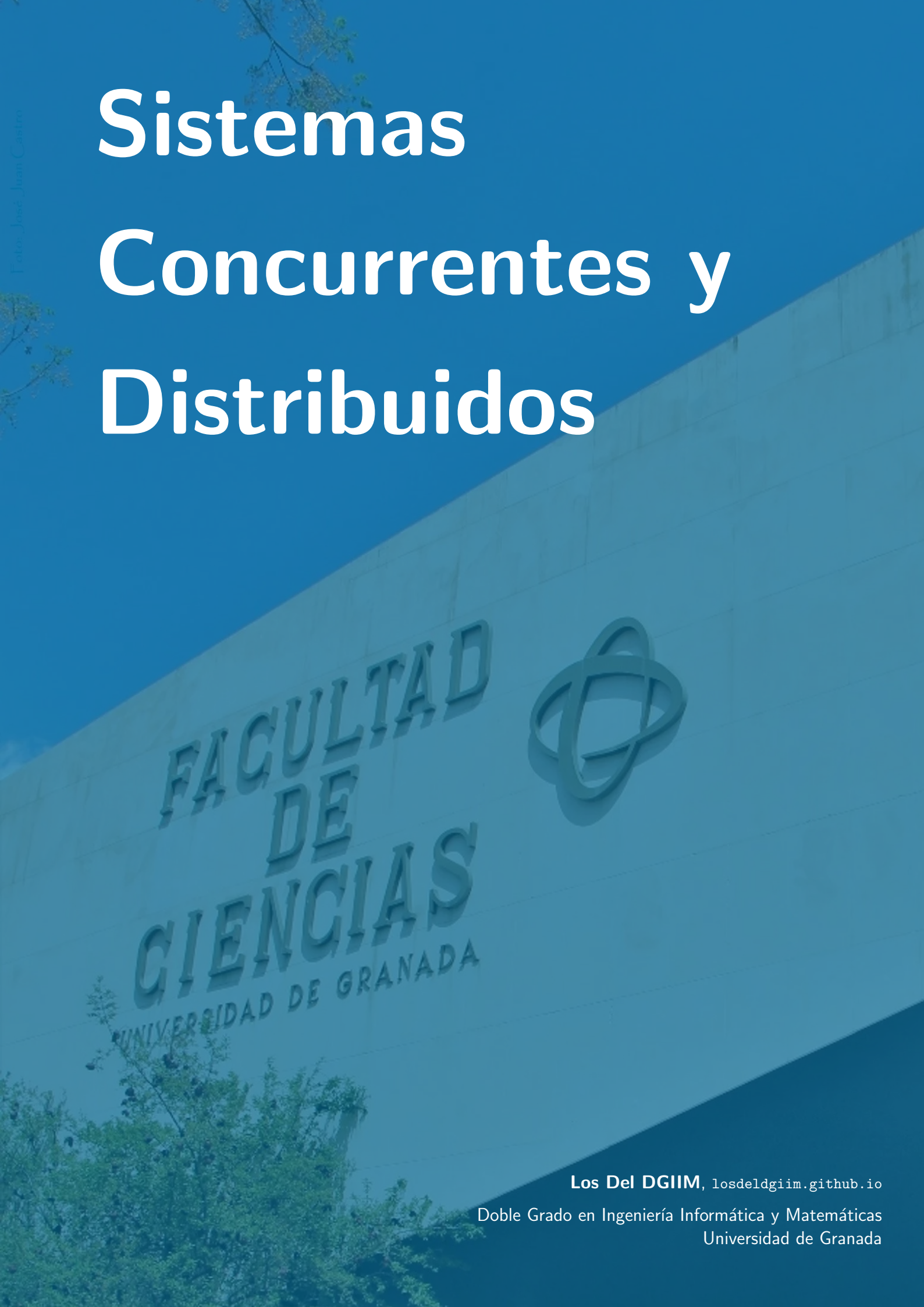
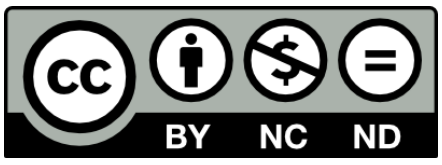
Sistemas Concurrentes y Distribuidos



Los Del DGIIM, losdeldgiim.github.io

Doble Grado en Ingeniería Informática y Matemáticas
Universidad de Granada



Sistemas Concurrentes y Distribuidos

Los Del DGIIM, `losdeldgiim.github.io`

Arturo Olivares Martos

Granada, 2024-2025

Índice general

1. Relaciones de problemas

1.1. Introducción

Ejercicio 1.1.1. 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
  begin
    for i:= 1 to 2 do begin
      x:= x + 1;
    end
  end
end
```

```


Process P2;
var j: integer;
begin
  begin
    for j:= 1 to 2 do begin
      x:= x + 1;
    end
  end
end
```

Ejercicio 1.1.2. ¿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:

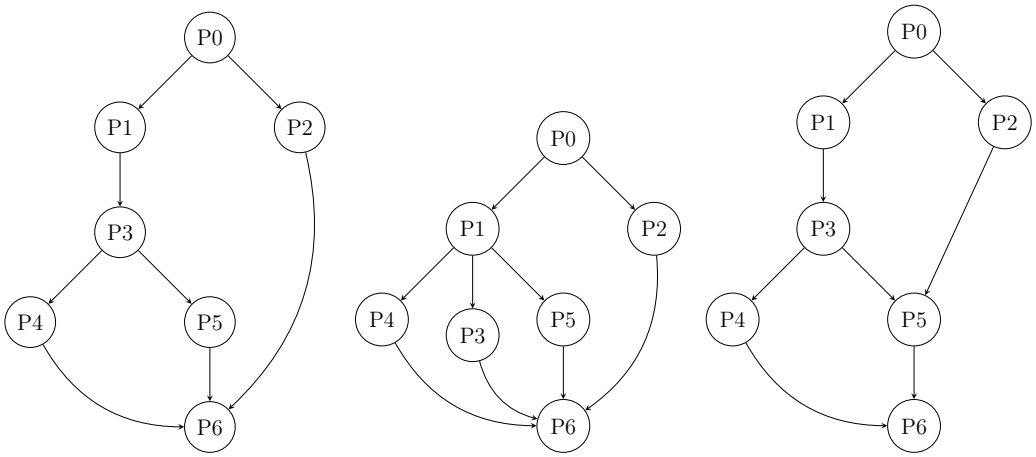
1. Los archivos son una secuencia de items de un tipo arbitrario `T`, y se encuentran ya abiertos para lectura (`f`) y escritura (`g`). Para leer un ítem de `f` se usa la llamada a función `leer(f)` y para saber si se han leído todos los ítems de `f`, se puede usar la llamada `fin(f)` que devuelve verdadero 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)`.

2. El orden de los items escritos en `g` debe coincidir con el de `f`.

3. Dos accesos a dos archivos distintos pueden solaparse en el tiempo.

Ejercicio 1.1.3. Construir, utilizando las instrucciones concurrentes `cobegin-coend` y `fork-join`, programas concurrentes que se correspondan con los grafos de precedencia que se muestran en la figura **??**.

1. Grafo de precedencia de la figura 1.1a:



(a) DAG del apartado 1. (b) DAG del apartado 2. (c) DAG del apartado 3.

Figura 1.1: Grafos de precedencia del ejercicio 1.1.3.

2. Grafo de precedencia de la figura 1.1b:

3. Grafo de precedencia de la figura 1.1c:

Ejercicio 1.1.4. Dados los siguientes fragmentos de programas concurrentes, obtener sus grafos de precedencia asociados:

1. Programa de la figura 1.2a.

2. Programa de la figura 1.2b.

Ejercicio 1.1.5. 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 descrito en el Código Fuente 1.

Observación. 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).

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

(a) Programa 1.

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

(b) Programa 2.

Figura 1.2: Programas concurrentes del ejercicio 1.1.4.

```
{ 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
```

Código fuente 1: Código acumulador-escritor del ejercicio 1.1.5.

```
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
```

Código fuente 2: Procedimientos `Sort` y `Copiar` del ejercicio 1.1.6.

Ejercicio 1.1.6. 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`). Estas funciones se muestran en el Código Fuente 2.

El programa para ordenar se puede implementar de dos formas:

1. 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`.

2. 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`).

En el Código Fuente 3 se muestra el código de ambas versiones.

El código de la función `Merge`, disponible en el Código Fuente 4, 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`.

Llamaremos $T_s(k)$ al tiempo que tarda el procedimiento `Sort` cuando actúa 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:

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

```
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
```

Código fuente 3: Procedimientos `Secuencial` y `Concurrente` del ejercicio 1.1.6.

```
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
```

Código fuente 4: Procedimiento `Merge` del ejercicio 1.1.6.

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

$$S = T_s(n) = \frac{1}{2} \cdot (2n)^2 - \frac{1}{2} \cdot 2n = 2n^2 - n$$

Con estas definiciones, calcular el tiempo que tardará la versión paralela, en dos casos:

1. Las dos instancias concurrentes de `Sort` se ejecutan en el mismo procesador (llamamos P_1 al tiempo que tarda).

2. Cada instancia de `Sort` se ejecuta en un procesador distinto (lo llamamos P_2).

Escribe una comparación cualitativa de los tres tiempos (S, P_1 y P_2). Para esto, hay que suponer que cuando el procedimiento `Merge` actúa 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 $T_m(p) = p$.

Ejercicio 1.1.7. SSupongamos 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
```

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.

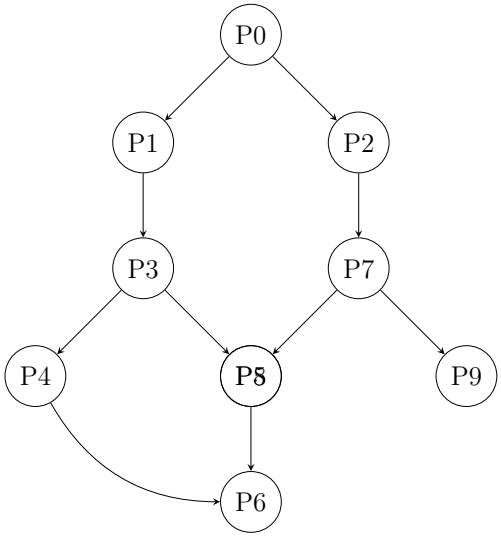
Ejercicio 1.1.8. Un trozo de programa ejecuta nueve rutinas o actividades (`P1`, `P2`, . . . , `P9`), repetidas veces, de forma concurrentemente con `cobegin-coend` (ver trozo de código de la figura 1.3a), pero que requieren sincronizarse según determinado grafo (ver la figura 1.3b).

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.

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

(a) Código del ejercicio 1.1.8.



(b) DAG del ejercicio 1.1.8.

Figura 1.3: Figuras del ejercicio 1.1.8.

- 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 única 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.

Ejercicio 1.1.9. En el ejercicio 1.1.8 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 `S_n`, 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 }
    S_n ; { sentencias especificas de este proceso (desconocidas) }
    ..... { senalar que hemos terminado }
end
```

Ejercicio 1.1.10. Para los siguientes fragmentos de código, obtener la *poscondición* adecuada para convertirlo en un triple demostrable con la Lógica de Programas:

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

Ejercicio 1.1.11. ¿Cuáles de los siguientes triples no son demostrables con la Lógica de Programas?

1. $\{i > 0\} \quad i = i - 1; \quad \{i \geqslant 0\}$
2. $\{x \geqslant 7\} \quad x = x + 3; \quad \{x \geqslant 9\}$
3. $\{i < 9\} \quad i = 2 * i + 1; \quad \{i \leqslant 20\}$
4. $\{a > 0\} \quad a = a - 7; \quad \{a > -6\}$

Ejercicio 1.1.12. 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):

1. $\{P\}C\{Q \vee P\}$
2. $\{P \wedge D\}C\{Q\}$
3. $\{P \vee D\}C\{Q\}$
4. $\{P\}C\{Q \vee D\}$
5. $\{P\}C\{Q \wedge P\}$

Ejercicio 1.1.13. Si el triple $\{P\}C\{Q\}$ es demostrable, ¿cuál de los siguientes triples no se puede demostrar?

1. $\{P \wedge D\}C\{Q\}$
2. $\{P \vee D\}C\{Q\}$
3. $\{P\}C\{Q \vee D\}$
4. $\{P\}C\{Q \vee P\}$

Ejercicio 1.1.14. Dado el siguiente programa, obtener:

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

1. Valores finales de x e y.

2. Valores finales de x e y si quitamos los símbolos < > de instrucción atómica.

Ejercicio 1.1.15. Comprobar si la demostración del siguiente triple interfiere con los teoremas siguientes:

$$\{x \geqslant 2\} \quad < x = x - 2 > \quad \{x \geqslant 0\}$$

1. $\{x \geqslant 0\} \quad < x = x + 3 > \quad \{x \geqslant 3\}$

2. $\{x \geqslant 0\} \quad < x = x + 3 > \quad \{x \geqslant 0\}$

3. $\{x \geqslant 7\} \quad < x = x + 3 > \quad \{x \geqslant 10\}$

4. $\{y \geqslant 0\} \quad < y = y + 3 > \quad \{y \geqslant 3\}$

5. $\{x \text{ es impar}\} \quad < y = x + 1 > \quad \{y \text{ es par}\}$

Ejercicio 1.1.16. Dado el siguiente triple:

$$\{x == 0\}$$
$$\text{cobegin}$$
$$< x = x + a > \;||\; < x = x + b > \;||\; < x = x + c >$$
$$\text{coend}$$
$$\{x == a + b + c\}$$

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.