

Sistemas Concurrentes y Distribuidos

FACULTAD
DE
CIENCIAS
UNIVERSIDAD DE GRANADA



Los Del DGIIM, losdeldgiim.github.io

Doble Grado en Ingeniería Informática y Matemáticas
Universidad de Granada



Esta obra está bajo una Licencia Creative Commons Atribución-NoComercial-SinDerivadas 4.0 Internacional (CC BY-NC-ND 4.0).

Eres libre de compartir y redistribuir el contenido de esta obra en cualquier medio o formato, siempre y cuando des el crédito adecuado a los autores originales y no persigas fines comerciales.

Sistemas Concurrentes y Distribuidos

Los Del DGIIM, losdeldgiim.github.io

José Juan Urrutia Milán
Arturo Olivares Martos

Granada, 2024-2025

Índice general

1. Introducción a la Programación Concurrente	5
1.1. Conceptos básicos	5
1.1.1. Comparación de programas concurrentes con secuenciales . . .	5
1.1.2. Definición de concurrencia	6
1.1.3. Axiomas de la programación concurrente	6
1.2. Modelos para creación de procesos en un programa	8
1.2.1. Grafos de Sincronización	8
1.2.2. Definición estructurada de procesos	9
1.2.3. Definición no estructurada de procesos	9
1.3. Exclusión mutua y sincronización	10
2. Relaciones de problemas	13
2.1. Introducción	13

1. Introducción a la Programación Concurrente

1.1. Conceptos básicos

Hasta ahora, nos hemos dedicado al estudio y desarrollo de programas secuenciales, que podemos entender de forma intuitiva como una ejecución lineal de instrucciones.

En programación concurrente, tendremos ahora múltiples unidades de ejecución independientes, a las que llamaremos procesos (sea un core o un procesador). La programación concurrente trata de coordinar los procesos para que cooperen entre sí con el fin de realizar un problema global de forma mucho más rápida de como lo haría un programa secuencial.

Podemos pensar que un proceso es una unidad de software abstracta conformada por un conjunto de instrucciones a ejecutar y por el contexto del procesador (como los valores de los registros, el contador de programa, el puntero de pila, la memoria Heap, memoria para variables, el acceso a determinados recursos, ...), al que llamamos estado del proceso.

Cuando en esta asignatura aparezca “flujo de control”, debemos pensar en una secuencia de ejecución de instrucciones. Es decir, como si fuera un proceso pero carente de un estado.

Nuestro trabajo en esta asignatura será gestionar la concurrencia, es decir, la ejecución independiente de dichos procesos con el fin de que no sea una sucesión de eventos incontrolados.

1.1.1. Comparación de programas concurrentes con secuenciales

Normalmente, en un programa concurrente tendremos más procesos que núcleos donde ejecutar dichos procesos, de donde aparece el concepto de concurrencia: en programación concurrente debe parecer que todos los procesos avanzan de forma simultánea, pese a haber más procesos que núcleos.

Si provocamos cambios de contexto dejando avanzar al resto de flujos de con-

trol, el programa no sufrirá las latencias provocadas por los procesos de E/S (por ejemplo), haciendo que el programa global sea más eficiente gracias a la concurrencia.

En sistemas que simulen el mundo real, podemos asociar un proceso con cada ente que intervenga en nuestro sistema (como una simulación del tráfico en una ciudad, o del movimiento de planetas), con lo que los sistemas de simulación pueden modelarse mejor con procesos concurrentes independientes, más que con programas secuenciales.

1.1.2. Definición de concurrencia

Podríamos definir la concurrencia como el paralelismo potencial que existe en los programas que puede aprovecharse independientemente de las limitaciones del hardware en el que se ejecuta el programa.

Como ya hemos mencionado, podremos tener un mayor número de procesos que de cores, y con este modelo cada uno de los procesos se ejecuta aparentemente al mismo tiempo que los demás.

El concepto de concurrencia es un concepto de programación a alto nivel que trata de representar el paralelismo potencial que existe en un programa. Con los compiladores adecuados, podemos programar en función de dichas características sin limitarnos por la arquitectura hardware del ordenador.

El objetivo fundamental de la concurrencia es simplificar toda la parte de la sincronización y comunicación entre los diferentes procesos de un programa, el cual suele ser un problema complejo sin solución fácil. Nos da un nivel algorítmico suficientemente independiente de los detalles del hardware para resolver dichos problemas, facilitando la portabilidad del código entre arquitecturas y lenguajes de programación.

Como beneficios de este modelo abstracto (el de la concurrencia), podemos destacar:

- Da herramientas, instrucciones y sentencias útiles para problemas de sincronización entre procesos.
- Las primitivas de programación en un lenguaje de alto nivel (como son los lenguajes concurrentes) son más fáciles de utilizar que con lenguajes de bajo nivel. Por ejemplo, los semáforos son más complejos que los monitores.
- Evita la dependencia con instrucciones de bajo nivel, haciendo que el programa pueda ejecutarse en otra computadora.

1.1.3. Axiomas de la programación concurrente

La programación concurrente es un modelo abstracto definido en base a 5 axiomas que nos dicen si un lenguaje es o no concurrente. En caso de no cumplirse, el código no va a poder ser transportable ni verificable.

Estos axiomas son:

1. Atomicidad y entrelazamiento de instrucciones atómicas.

Al menos ciertas instrucciones han de ser atómicas (esto es, instrucciones que no pueden ser interrumpidas, como por ejemplo las lecturas y escrituras en memoria).

2. Consistencia de los datos tras un acceso concurrente.

Si tenemos muchos procesos actuando a la vez sobre un conjunto de datos compartidos, debemos estar seguros de que los accesos a los mismos no los estropeen.

3. Irrepetibilidad de las secuencias de instrucciones.

Cuando se ejecuta un programa concurrente, se sucede un entrelazamiento de las instrucciones de los procesos que se ejecutan a la vez, con lo que la secuencia de instrucciones que obtenemos como resultado de volver a ejecutar el mismo programa con otros datos es muy probable que no sea la misma.

Esto dificulta el debugging de un programa concurrente, ya que podemos tener un error en el programa que repercuta en el mal funcionamiento del mismo sólo cuando se suceda una secuencia de instrucciones específica en la ejecución del mismo.

4. Independencia de la velocidad de los procesos.

No puede hacerse ninguna suposición en la velocidad de ejecución de un proceso, ya que este puede verse suspendido o ralentizado.

La corrección en programas concurrentes no debe depender de la velocidad relativa de los procesos.

5. Hipótesis del progreso finito.

- Un proceso debe tratar de avanzar todo lo que pueda. Esto es, si un proceso se está ejecutando, debe tratar de ejecutar tantas instrucciones como sea posible.
- Una vez que un proceso comienza a ejecutar una sección de código, debe terminar dicha sección.
- Todo proceso debe seguir progresando durante la ejecución de un programa.

Cuando se interpreta la ejecución de un programa concurrente como un conjunto de trazas de las cuales elegimos una al ejecutar el programa, estamos ignorando ciertos detalles, como:

- El estado de la memoria asignado a cada proceso.
- El valor de los registros de cada proceso.
- El coste computacional de los cambios de contexto.
- La política de planificación que se emplea de los procesos.
- El desarrollo de los programas es independiente del hardware.

1.2. Modelos para creación de procesos en un programa

En relación al número de procesos que se ejecutan en un programa, podemos clasificarlos en:

- Sistemas estáticos: El número de procesos en el programa es el mismo durante su ejecución. Dicho número se define al programarlo y en el momento de la compilación.
- Sistemas dinámicos: El número de procesos es variable, de forma que durante la ejecución del programa pueden crearse y destruirse procesos.

1.2.1. Grafos de Sincronización

Un Grafo de Sincronización es un Grafo Dirigido Acíclico (DAG) donde cada nodo representa una secuencia de sentencias del programa (o actividad). Nos sirven para definir situaciones de preferencia en la ejecución de un programa. Tenemos que tener instrucciones en el lenguaje concurrente que nos permitan representar el comienzo de las instrucciones con un DAG.

En un DAG, se suceden dependencias secuenciales, esto es, un proceso no empieza hasta que termina otro: dadas dos actividades S_1 y S_2 , una arista desde la primera hacia la segunda ($S_1 \rightarrow S_2$) significa que S_2 no puede comenzar su ejecución hasta que S_1 haya finalizado.

Ejemplo. El DAG de la Figura 1.1 nos indica que la primera actividad que tendrá lugar en nuestro programa será la actividad S_1 . Tras el fin de esta, se sucederán de forma concurrente las actividades S_2 y S_3 . Tras terminar S_2 , comenzará S_4 y, tras esta, se ejecutarán de forma concurrente S_5 y S_6 . Finalmente, tras el final de S_5 , S_6 y S_3 , el programa terminará con la actividad S_7 .



Figura 1.1: Ejemplo de DAG

En relación a cómo podemos crear los procesos, destacamos dos formas que podemos encontrarnos en los lenguajes paralelos:

1.2.2. Definición estructurada de procesos

En programación estructurada, contaremos con dos palabras reservadas del lenguaje que nos permitirán recrear la siguiente funcionalidad a explicar. En pseudocódigo, nos referiremos a ellas como **cobegin** y **coend**.

Dados dos procesos P_1 y P_2 que queremos que se ejecuten de forma concurrente, bastará especificar en pseudocódigo:

```
1  cobegin
    P1;
    P2;
coend
```

Hasta llegar a la palabra **cobegin** no comenzará ningún proceso. Tras esta, se sucederá un entrelazado de las instrucciones de P_1 y P_2 , y no se saldrá de dicha región hasta que terminen ambos procesos.

Ejemplo. Un programa utilizando la definición estructurada de procesos que cumpla el DAG de la Figura 1.1 es el siguiente:

```
1  begin
    S1;
    cobegin
        begin
5      S2;
        S4;
        cobegin
            S5;
            S6;
10     coend
        end
    S3;
    coend
    S7;
15 end
```

1.2.3. Definición no estructurada de procesos

En lenguajes concurrentes que no cuenten con palabras reservadas que simulen **cobegin** y **coend**, contaremos con dos llamadas al sistema que nos permitirán replicar dicha funcionalidad para crear procesos:

fork

Duplica el proceso que actualmente se está ejecutando y lo lanza a ejecución. Si se le especifica una rutina, cambiará el código del clon por dicha rutina.

join

Espera a que cierto proceso termine de ejecutarse antes de proseguir con la ejecución del resto de instrucciones.

La función **fork** ya se vió en la asignatura de Sistemas Operativos, por lo que el estudiante debería estar familiarizado con ella.

Ejemplo. Un programa con definición no estructurada de procesos para el DAG de la Figura 1.1 es el siguiente:

```
1  begin
    S1;
    fork S3;
    S2;
5  S4;
    fork S6;
    S5;
    join S3;
    join S6;
10 S7;
end
```

1.3. Exclusión mutua y sincronización

No todas las secuencias de entrelazamiento de un programa concurrente van a ser aceptables. Para impedir que sucedan ciertas secuencias (o trazas) tenemos condiciones de sincronización, relacionadas con instrucciones de lenguajes de programación de tal forma que dichas instrucciones no se ejecutan hasta que no es cierta una condición que depende de las variables del proceso.

De esta forma, una **condición de sincronización** es una restricción en el orden en que se pueden entremezclar las instrucciones que generan los procesos de un programa. Podemos utilizarlas para asegurarnos de que todas las trazas del programa son correctas.

La **exclusión mutua** es un caso particular de sincronización en el que se obliga a que un trozo de código de un proceso sea ejecutado de forma totalmente secuencial de manera que no se permita el entrelazamiento con otros procesos. Este trozo de código (en el que no se permite el entrelazamiento de instrucciones con otros procesos) recibe el nombre de **sección crítica**. Se dice que las secciones críticas se ejecutan en exclusión mutua.

La mayoría de instrucciones en un programa son instrucciones compuestas (esto es, formadas por varias instrucciones en lenguaje máquina). Si queremos establecer secciones críticas para la ejecución de cada una de dichas instrucciones, rodearemos la instrucción por < y >.

Ejemplo. Por ejemplo, ante el siguiente código concurrente:

```
1  begin
    x := 0;
    cobegin
        x := x+1;
5  x := x-1;
    coend
end
```

El resultado obtenido en la variable **x** es indeterminado, ya que puede ser 1, -1 o 0:

- El segundo proceso puede leer la variable x antes de que el primero escriba en ella, leyendo 0; y podría escribir en ella después de que lo haga el primer proceso, escribiendo finalmente un -1.
- Podría ejecutarse el primer proceso antes que el segundo, dejando la variable x a 1 y el segundo le cambiaría el valor a 0.
- El primer proceso puede leer la variable x antes de que el segundo escriba en ella, leyendo 0; y podría escribir en ella después de que lo haga el segundo proceso, escribiendo finalmente un 1.

Notemos que esto sucede ya que la instrucción $x := x \text{ OP } a$ es una instrucción compuesta de las instrucciones máquina: `LOAD x , OP a , x` y `STORE x` .

Sin embargo, ante el siguiente código concurrente:

```

1  begin
    x := 0;
    cobegin
        < x := x+1 >;
5    < x := x-1 >;
    coend
end

```

Obtenemos siempre 0 en x , ya que las instrucciones de cada instrucción compuesta no se entrelazan, al ser secciones críticas.

Paradigma del Productor Consumidor

El paradigma del productor/consumidor es una situación de dos procesos que cooperan, uno escribiendo datos en una variable, al que llamaremos productor; y otro que leerá dicha variable y realizará cálculos con ella, al que llamaremos consumidor.

Este paradigma nos sirve de ejemplo para justificar las condiciones de sincronización, así como para ponerlas en práctica.

Son necesarias condiciones de sincronización ya que no todas las trazas de ejecución de un programa con estructura productor/consumidor son correctas.

Ejemplo. Si notamos por L a las lecturas del consumidor y por E a las escrituras del productor, las tres siguientes trazas de ejecución no son correctas:

1. L, E, L, E, \dots , porque leemos una lectura de la variable antes de que el productor escriba en ella, leyendo un valor indeterminado y pudiendo provocar el fallo del programa.
2. E, L, E, E, L, \dots , porque el consumidor se ha perdido una escritura del productor en la variable, que puede hacer que cambie la salida del programa a una errónea.
3. E, L, L, E, L, \dots , porque el consumidor ha usado un mismo dato dos veces, que también puede resultar en un mal funcionamiento del programa.

Para que el paradigma del productor/consumidor funcione correctamente, han de cumplirse las dos condiciones de sincronización siguientes:

1. El consumidor no puede leer la variable hasta que el productor no haya escrito en ella. Cuando el consumidor lee, debe esperar a que el productor proporcione un nuevo dato antes de volver a leer.
2. El productor no puede escribir un nuevo valor hasta que el consumidor haya leído el último dato escrito (salvo en el primer valor a escribir).

Para cumplir con las condiciones de sincronización, deberemos añadir instrucciones en el código para que:

- El consumidor se detenga la primera vez hasta que el productor escriba en la variable.
- Se impida un segundo ciclo del consumidor hasta que se produzca el siguiente dato.
- Se impida un segundo ciclo del productor hasta que el dato anterior no haya sido leído por el consumidor.

2. Relaciones de problemas

2.1. Introducción

Ejercicio 2.1.1. 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.

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

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

Ejercicio 2.1.2. ¿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 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)**.
2. El orden de los ítems escritos en **g** debe coincidir con el de **f**.
3. Dos accesos a dos archivos distintos pueden solaparse en el tiempo.

Ejercicio 2.1.3. 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 2.1a:

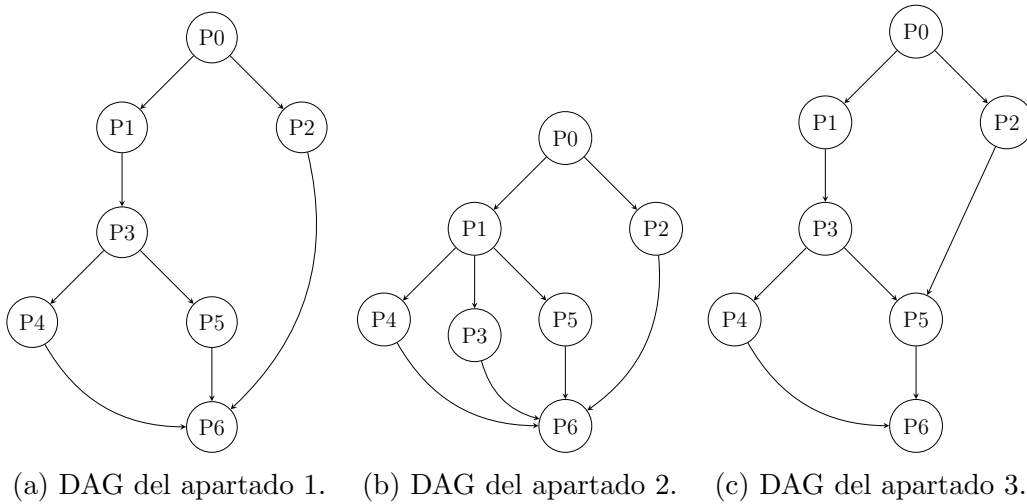


Figura 2.1: Grafos de precedencia del ejercicio 2.1.3.

2. Grafo de precedencia de la figura 2.1b:
3. Grafo de precedencia de la figura 2.1c:

Ejercicio 2.1.4. Dados los siguientes fragmentos de programas concurrentes, obtener sus grafos de precedencia asociados:

1. Programa de la figura 2.2a.
2. Programa de la figura 2.2b.

Ejercicio 2.1.5. 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).


```

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

```

(a) Programa 1.

```

1  begin
    P0 ;
    cobegin
        begin
5         cobegin
            P1 ; P2 ;
        coend
        P5 ;
        end
10     begin
        cobegin
            P3 ; P4 ;
        coend
        P6 ;
15     end
    coend
    P7 ;
end

```

(b) Programa 2.

Figura 2.2: Programas concurrentes del ejercicio 2.1.4.

```

1  { variable compartida: }
var n : integer; { contabiliza impulsos }
begin
    while true do begin
5     Espera_impulso();
        < n := n+1 > ; { (1) }
    end
end
process Escritor ;
10 begin
    while true do begin
        Espera_fin_hora();
        write( n ) ; { (2) }
        < n := 0 > ; { (3) }
15    end
end

```

Código fuente 1: Código acumulador-escritor del ejercicio 2.1.5.

```

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

```

Código fuente 2: Procedimientos `Sort` y `Copiar` del ejercicio 2.1.6.

Ejercicio 2.1.6. 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:

```

1  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:

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

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

Código fuente 3: Procedimientos `Secuencial` y `Concurrente` del ejercicio 2.1.6.

```
1 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 }  
5  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 }  
10    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 }  
15      b[escribir] := a[leer2] ;  
        leer2 := leer2 + 1 ;  
      end  
      escribir := escribir+1 ;  
    end  
20    { 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 }  
25  end
```

Código fuente 4: Procedimiento `Merge` del ejercicio 2.1.6.

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 2.1.7. 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:

```

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

Ejercicio 2.1.8. 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 2.3a), pero que requieren sincronizarse según determinado grafo (ver la figura 2.3b).

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:

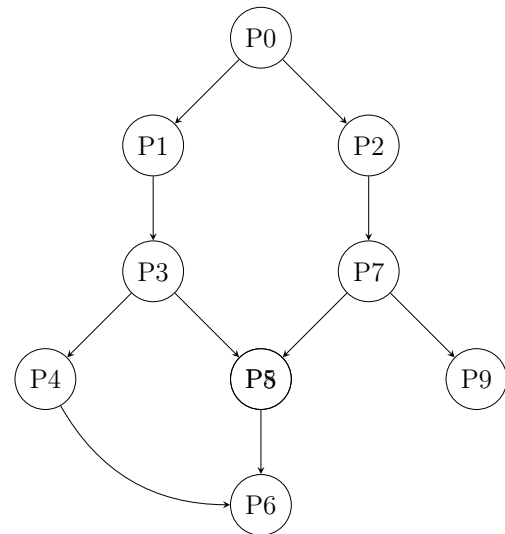
- El procedimiento **EsperarPor**(*i*) es llamado por una rutina cualquiera (la número *k*) para esperar a que termine la rutina número *i*, usando espera ocupada. Por tanto, se usa por la rutina *k* al inicio para esperar la terminación de las otras rutinas que corresponda según el grafo.

```

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

```

(a) Código del ejercicio 2.1.8.



(b) DAG del ejercicio 2.1.8.

Figura 2.3: Figuras del ejercicio 2.1.8.

- 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 2.1.9. En el ejercicio 2.1.8 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:

```

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

```

Ejercicio 2.1.10. 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 2.1.11. ¿Cuáles de los siguientes triples no son demostrables con la Lógica de Programas?

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

Ejercicio 2.1.12. 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 2.1.13. 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 2.1.14. Dado el siguiente programa, obtener:

```

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

```

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

Ejercicio 2.1.15. Comprobar si la demostración del siguiente triple interfiere con los teoremas siguientes:

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

1. $\{x \geq 0\} \quad < x = x + 3 > \quad \{x \geq 3\}$
2. $\{x \geq 0\} \quad < x = x + 3 > \quad \{x \geq 0\}$
3. $\{x \geq 7\} \quad < x = x + 3 > \quad \{x \geq 10\}$
4. $\{y \geq 0\} \quad < y = y + 3 > \quad \{y \geq 3\}$
5. $\{x \text{ es impar}\} \quad < y = x + 1 > \quad \{y \text{ es par}\}$

Ejercicio 2.1.16. Dado el siguiente triple:

$$\begin{array}{c} \{x == 0\} \\ \text{cobegin} \\ < x = x + a > \parallel < x = x + b > \parallel < x = x + c > \\ \text{coend} \\ \{x == a + b + c\} \end{array}$$

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.