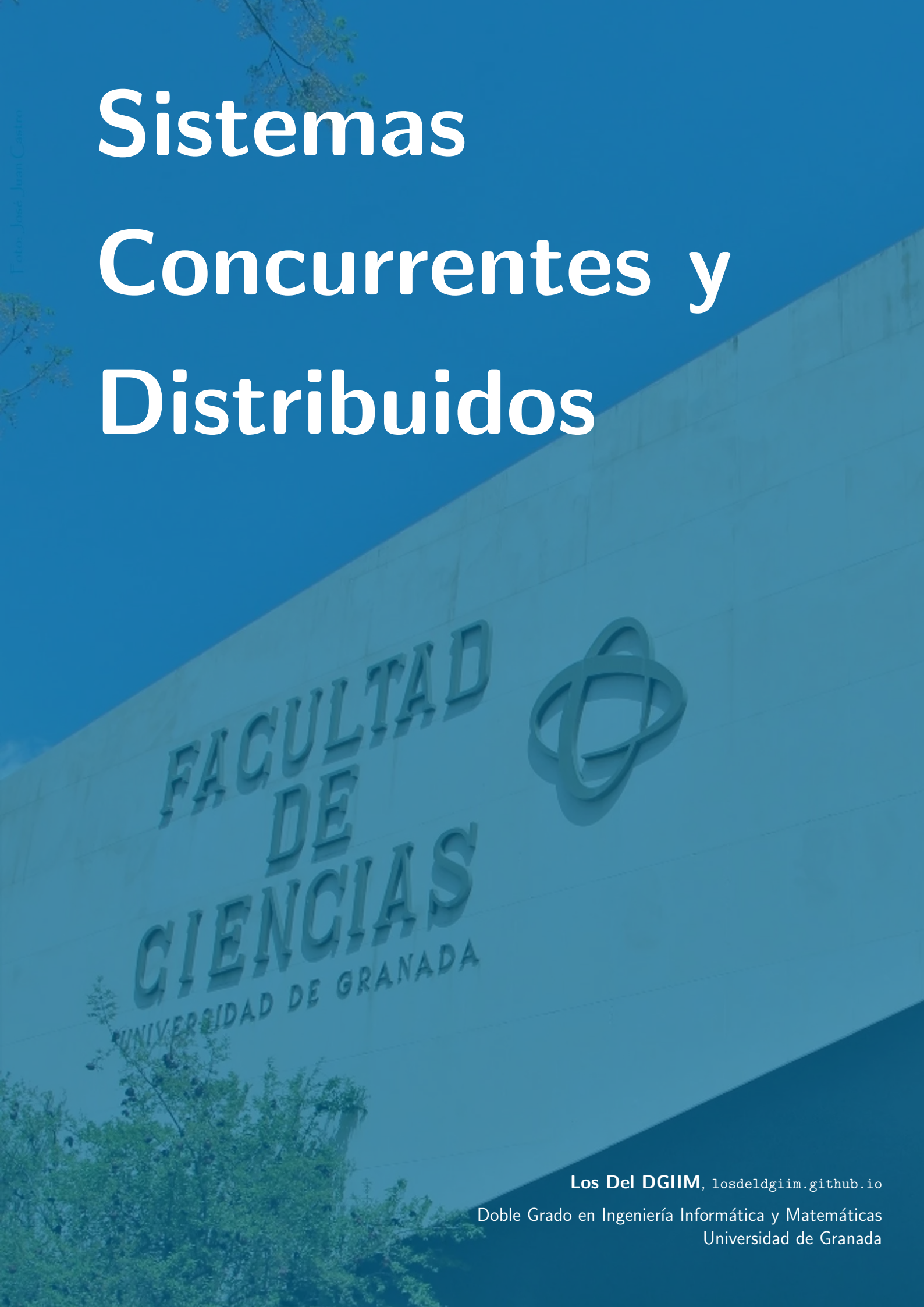
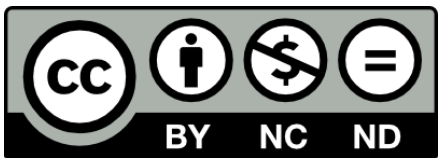
Sistemas Concurrentes y Distribuidos



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`

José Juan Urrutia Milán
Arturo Olivares Martos

Granada, 2024-2025

Índice general

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 con 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 carante 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 programa 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 computaciones 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 proesos 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 succeden 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.

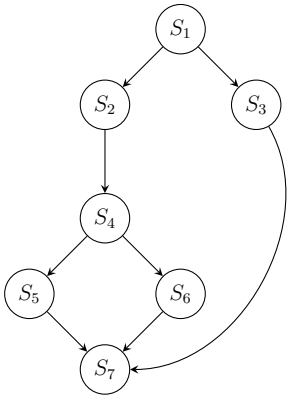


Figura 1.1: Ejemplo de DAG

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:

```
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:

```
begin
  S1;
  cobegin
    begin
      S2;
      S4;
      cobegin
        S5;
        S6;
      coend
    end
    S3;
  coend
  S7;
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 el 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:

```
begin
  S1;
  fork S3;
  S2;
  S4;
  fork S6;
  S5;
  join S3;
  join S6;
  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:

```
begin
  x := 0;
  cobegin
    x := x+1;
    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 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 compeusta 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, ...`, porque leemos una lectura de la variable antes de que el productor escriba en ella, leyendo un valor indeterminado y puediendo provocar el fallo del programa.

2. `E, L, E, E, L, ...`, 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, ...`, 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 leido 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.

1.4. Propiedades de los sistemas concurrentes

Una propiedad de un sistema concurrente es un atributo que se cumple en toda la ejecución del sistema, mientras que el conjunto de todas sus ejecuciones (de todas las posibles trazas generadas en la ejecución) nos dan el comportamiento del sistema.

Cualquier propiedad de un sistema concurrente puede ser formulada como combinación de dos tipos de propiedades fundamentales:

- Propiedades de seguridad (*safety*): Una propiedad de este tipo afirma que hay un estado del programa que es inalcanzable.

- Propiedades de vivacidad (*liveness*): Propiedades que afirman que en algún momento se alcanzará un estado deseado.

1.4.1. Propiedades de seguridad (*safety*)

Estas propiedades expresan determinadas condiciones que han de cumplirse durante toda la ejecución del programa. Cualquier propiedad que pueda ser formulada por la existencia de un estado inalcanzable, es una propiedad de seguridad. En dicho caso, deberíamos poder definir qué estado es inalcanzable y demostrar que el programa concurrente nunca puede llegar a dicho estado.

Las propiedades de seguridad pueden ya comprobarse en tiempo de compilación, ya que se cumplen independientemente de la ejecución concreta que sigue el sistema en tiempo de ejecución. Es por esto que se trata de una propiedad estática.

Ejemplos de problemas donde vemos propiedades de seguridad son:

- El problema de la exclusión mutua: La condición de que dos procesos del programa no puedan ejecutar simultáneamente las instrucciones de una sección crítica es de seguridad.
- Problema del productor/consumidor: Todos los estados que lleven a una traza distinta de `E, L, E, L, ...` son estados prohibidos.
- La situación de interbloqueo: Es una de las situaciones más críticas que se dan tras quebrantar una propiedad de seguridad, ya que hay procesos ocupando recursos que no están usando y que no liberarán.

1.4.2. Propiedades de vivacidad (*liveness*)

Las propiedades de vivacidad expresan que el sistema llegará en un futuro a cumplir determinadas condiciones (en un tiempo no indeterminado). En determinados ejemplos, dichas propiedades pueden entenderse como que las condiciones dinámicas de ejecución no lleven a que determinados procesos sean sistemáticamente adelantados por otros, no pudiendo avanzar en la ejecución de instrucciones útiles, de forma que el proceso sufra inanición (*starvation*).

Para demostrar que una propiedad es de vivacidad, debemos definir un "buen estado" del programa, y demostrar que es alcanzable para todos los procesos en un determinado tiempo.

Ejemplos de problemas donde vemos propiedades de vivacidad son:

- El problema de la exclusión mutua: Las secciones críticas se ejecutan por un proceso a vez. El sistema debe garantizar que, en la espera por entrar a una región crítica, no ocurra que un proceso sea siempre adelantado por otros procesos, llevando a que dicho proceso nunca ejecute la región crítica (que es nuestro estado deseado).
- El problema del productor/consumidor: Un proceso que quiera escribir o leer de la variable compartida ha de poder hacerlo en un tiempo finito.

Debemos notar que el axioma de proceso finito expuesto en secciones anteriores no tiene nada que ver con la ausencia de inanición:

- El axioma de proceso finito afirma que los procesos no pueden quedarse parados arbitrariamente, sino que estos deben intentar ejecutar instrucciones conforme les sea posible.
- Un proceso puede estar ejecutando instrucciones en un bucle indefinido pero no avanzar en la ejecución de las instrucciones de su código (es decir, puede estar realizando un trabajo inútil). En este caso, se cumpliría el axioma del proceso finito pero no se cumpliría la propiedad de vivacidad, ya que el proceso sufriría inanición.

Como ya venimos avisando, el no cumplimiento de la propiedad de vivacidad puede llevar a uno o más procesos a un estado de inanición (es indefinidamente pospuesto por otros, de forma que no pueda realizar aquello para lo que está programado). Aunque dicha situación es menos grave que una situación de interbloqueo

(ya que hace que el programa no avance nada), tenemos procesos inoperantes (que no realizan su trabajo), por lo que consideraríamos que el programa concurrente no es correcto.

De esta forma, un programa concurrente solo podrá ser completamente correcto cuando se demuestre que los procesos que lo integran no sufren inanición en ninguna de sus posibles ejecuciones.

Ejemplo (Cena de filósofos)**.** Disponemos de cinco filósofos, F_0, F_1, F_2, F_3 y F_4, que dedican su vida a pensar y en algún momento desean comer. Acceden a una mesa redonda en la que hay un plato del que todos pueden comer, siempre y cuando dispongan de dos palillos. Sólo hay 5 palillos, estos distribuidos de forma que entre dos filósofos hay un palillo.

Los filósofos son cabezotas, por lo que una vez congen un palillo, no están dispuestos a soltarlo. Ademś, no pueden arrebatar un palillo a otro filósofo por ir en contra de sus ideales morales.

Ante la situación descrita, podemos llegar a ver los dos ejemplos siguientes:

- Si todos los filósofos cogen a la vez el palillo de su derecha, cada filósofo dispondrá de un palillo y no habrá más palillos libres.

  Estamos ante una situación de interbloqueo: ningún filósofo puede comer y no podrá hacerlo jamás. Como resultado, todos los filósofos se morirán de hambre.

- Si, por ejemplo, los filósofos F_0 y F_2 (que rodean al filósofo F_1 en la mesa) conspiran para dejar morir de hambre al filósofo F_1 de forma que cuando F_0 deje el palillo que hay entre F_0 y F_1, F_2 coja el palillo que hay entre él y F_1 (y viceversa), conseguirán que F_1 nunca consiga sus dos palillos, llevando al filósofo a un estado de inanición y, posteriormente, la muerte.

Esta asignatura trata de crear protocolos que podamos demostrar que cumplen con las propiedades de seguridad y vivacidad, con el fin de no llevar nunca a situaciones de interbloqueos, inanición de algún (os) proceso (s), o alguna de las malas situaciones comentadas anteriormente.

1.5. Lógica de programas de Hoare y verificación de programas concurrentes

1.5.1. Corrección de los programas concurrentes

En los programas secuenciales, para comprobar la corrección de los mismos, debemos probar que el programa **termina** dando **salidas esperadas** ante determinadas entradas.

En un programa secuencial, hay un único conjunto de datos de entrada que provoca un único conjunto de datos de salida. Esto no sucede en programas concurrentes, ya que el indeterminismo en la ejecución provoca distintas trazas posibles del programa, y es bastante probable que todas las trazas posibles no provoquen los

mismos resultados.

Para extender la definición de programa correcto a los programs concurrentes, notemos primero que muchos de ellos están pensado para no terminar nunca, de forma que su fin esté relacionado con alguna situación de error. Los sistemas operativos o los cajeros automáticos son programas concurrentes que están pensados para que nunca terminen, por lo que no podemos decir que una condición necesaria para que un programa concurrente sea correcto es que termine.

Para llevar a cabo la verificación de software, es decir, la demostración de que un programa es correcto, podemos emplear diferentes métodos:

Depuración de código. Explorar algunas ejecuciones de un código y comprobar que dichas ejecuciones son aceptables porque se cumplen las propiedades previamente fijadas.

Este método sirve para programas secuenciales, pero no para programas paralelos, ya que nos es imposible depurar un código ante todas las posibles combinaciones de distintas trazas de ejecución.

Razonamiento operacional. Realizar un análisis de casos exhaustivo para explorar todas las posibilidades de secuencias de ejecución de un código con el fin de garantizar que todas son correctas.

Es un método inviable para programas concurrentes. Por ejemplo, en un programa que use dos procesos, cada uno con 3 instrucciones atómicas, el número de posibles trazas de ejecución es de 20.

Razonamiento asertivo. Realizar un análisis abstracto basado en Lógica Matemática que permita representar de forma abstracta los estados[1] concretos que un programa alcanza.

De esta forma, el únifco enfoque posibles es el razonamiento asertivo.

1.5.2. Lógica de Hoare

Axiomática del lenguaje

Construiremos ahora un sistema lógico formal (SLF) que facilite la elaboración de asertos o proposiciones lógicas ciertas con una base lógico-matemática precisa.

Nuestro SLF estará formado por:

- Símbolos: Como sentencias del lenguaje de programación, variables proposicionales, operadores, ...
- Fórmulas: Secuencias de símbolos bien formadas[2].
- Axiomas: Proposiciones que mediante un consenso se consideran verdaderas.

[1] Un estado del programa viene definido por los valores que tienen las variables del programa en determinado instante durante su ejecución.

[2] Entendemos por esto a sucesiones de símbolos con un significado fácilmente entendible.

- Reglas de inferencia o de derivación: Reglas que nos permiten derivar fórmulas ciertas a partir de axiomas o de fórmulas que ya conocemos que son ciertas.

Podemos pensar en las reglas de inferencia como teoremas matemáticos: tienen unas hipótesis y unas tesis de forma que, en cualquier situación que las hipótesis sean ciertas, las tesis lo serán.

Notación. Notaremos a las reglas de inferencia por:

$$(\text{nombre de la regla})\frac{H_1, H_2, \ldots, H_n}{C}$$

De forma que disponemos de n hipótesis (H_1, H_2, ..., H_n) en conjunción que nos llevan a la tesis C.

Para proseguir con el detallamiento del SLF, es necesario antes la definición de interpretación:

Definición 1.1 (Interpretación)**.** Sea A el conjunto de todos los asertos o fórmulas lógicas, una interpretación será una aplicación de dominio A y codominio el conjunto $\{V, F\}$[3].

De esta forma, dada una interpretación y un aserto v, podemos ver la veracidad o falsedad de v gracias a la interpretación.

Para que las demostraciones de nuestro SLF sean confiables, este sistema debe ser seguro y completo. Fijada una interpretación:

- Decimos que un sistema es seguro si todos los asertos son hechos ciertos.
- Decimos que un sistema es completo si todos los hechos ciertos son asertos.

A partir de ahora, supondremos que no hay diferencia entre asertos y hechos ciertos. Es decir, que nuestro sistema es seguro y completo.

Lógica proposicional

Las fórmulas del SLF que estamos construyendo se llaman proposiciones, y están formadas por:

- Constantes proposicionales $\{V, F\}$.
- Variables proposicionales $\{p, q, r, \ldots\}$.
- Operadores lógicos $\{\neg, \wedge, \vee, \rightarrow, \leftarrow, \longleftrightarrow\}$.
- Expresiones formadas por constantes, variables y operadores.

Al igual que sucedía en la asignatura de Lógica y Métodos Discretos, podemos extender la definición de las interpretaciones y aplicarlas sobre las proposiciones del lenguaje, mediante unas reglas ya conocidas.

De esta forma, diremos que:

[3]Cuyos elementos interpretamos como verdadero y falso.

- Una fórmula es satisfacible si existe alguna ínterpretación que la satisfaga.
- Una fórmula será válida si se satisface en cualquier estado del programa (es decir, si cualquier interpretación la satisface).

Llamaremos a las fórmulas válidas tautologías.

Dentro de la lógica proposicional de este SLF son tautologías algunas fórmulas ya conocidas, como la distributiva de $\wedge$ y de $\vee$, la conmutatividad de las mismas, ...

Definición 1.2. Dadas dos fórmulas P y Q, diremos que son equivalentes siempre y cuando que P se satisfaga para una cierta interpretación si y solo si Q se satisface para la misma interpretación.

Por ejemplo, $p \rightarrow q$ y $\neg q \rightarrow p$ son fórmulas equivalentes.

Lógica de programas

Este SLF trata de hacer afirmaciones sobre la ejecución de un programa. Incluimos por tanto a los triples, que tienen la forma

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

donde P y Q son asertos (llamados precondición y postcondición, respectivamente) y S es una sentencia simple o estructurada de un lenguaje de programación. En P y Q podrán aparecer tanto variables lógico-matemáticas como variables del propio programa. Para distinguirlas, notaremos a las primeras con letas mayúsculas y a las segundas con minúsculas.

Un triple $\{P\}S\{Q\}$ se interpreta como cierto si, ante cualquier estado del programa que satisfaga P y después de la ejecución de cualquier entrelazamiento de instrucciones atómicas de S[4], llegamos a un estado del programa que satisfaga Q.

Notación. A partir de la notación de los triples, y siendo P una fórmula del lenguaje, notaremos por

$$\{P\}$$

Al conjunto de los estados del programa que verifican P.

De esta forma, $\{V\}$ es el conjunto de todos los estados de un programa, ya que todos los estados de dicho programa verifican V.

Análogamente, $\{F\}$ se corresponde con el conjunto vacío.

[4]Por tanto, S ha de finalizar en algún momento.

Notación. Dadas P y Q asertos equivalentes, entonces obtenemos el mismo conjunto de estados del programa que verifican dichos asertos:

$$\{P\} = \{Q\}$$

Sin embargo, para evitar la confusión con el operador de asignación, notaremos las igualdades entre los conjuntos de estados de un programa con el operador $\equiv$.

Definición 1.3 (Sustitución textual)**.** Dado un aserto P, que contiene al menos una aparición libre de la variable x y una expresión e, definimos la sustitución textual de x por e, notado por P_e^x como la sustitución textual de todas las ocurrencias libres de x en P por e.

Enumeramos ahora los axiomas de nuestra Lógica de programas:

Axioma de la sentencia nula $\{P\}$ *null* $\{P\}$**.**
Es decir, si el aserto P es cierto antes de la ejecución de la sentencia nula (esta es, la que no cambia nada en el programa), P seguirá siendo cierto tras la ejecución de la misma.

Axioma de asignación $\{P_e^x\}\ x = e\ \{P\}$**.**
Es decir, la asignación de un determinado valor e a una variable x solo cambia en el programa el valor de dicha variable x.

Ejemplo. Un ejemplo de uso del axioma de asignación es el siguiente:
Tratamos de probar que el triple $\{V\}\ x = 5\ \{x = 5\}$ es cierto. Es decir, que desde cualquier estado del programa, si asignamos 5 a x, acabaremos en cualquier estado del programa en el que x valga 5.

Demostración. Sea P la fórmula dada por $x = 5$ y e el valor numérico 5, sabemos que el axioma de asignación es cierto, luego se cumplirá que:

$$\{P_e^x\}\ x = e\ \{P\}$$

de donde:

$$\{V\} \equiv \{5 = 5\}\ x = e\ \{x = 5\}$$

□

Seguidamente, para cada una de las sentencias que afectan al flujo de control en un programa secuencial, contamos con reglas de inferencia para poder formar triples correctos en las demostraciones; además de dos reglas básicas de consecuencia.

Regla de la consecuencia (1).

$$\frac{\{P\}S\{Q\}, \{Q\} \to \{R\}}{\{P\}S\{R\}}$$

Es decir, siempre podemos hacer más débil la postcondición de un triple, de forma que este siga siendo cierto.

Regla de la consecuencia (2).

$$\frac{\{R\} \rightarrow \{P\}, \{P\}S\{Q\}}{\{R\}S\{Q\}}$$

Es decir, siempre podemos hacer más fuerta la precondición de un triple, manteniendo su veracidad.

Regla de la composición.

$$\frac{\{P\}S_1\{Q\}, \{Q\}S_2\{R\}}{\{P\}S_1; S_2\{R\}}$$

Es decir, podemos condensar dos triples en uno, siempre y cuando la postcondición de uno sea la precondición del otro.

Regla del *if*.

$$\frac{\{P \wedge B\}S_1\{Q\}, \{P \wedge \neg B\}S_2\{Q\}}{\{P\}\ \texttt{if B then}\ S_1\ \texttt{else}\ S_2\ \{Q\}}$$

De esta forma, siempre que queramos probar que una tripleta de la forma

$$\{P\}\ \texttt{if X then}\ S_1\ \texttt{else}\ S_2\ \{Q\}$$

es cierta, tendremos que probar que las tripletas

$$\{P \wedge X\}S_1\{Q\} \qquad \{P \wedge \neg X\}S_2\{Q\}$$

son ciertas.

Regla de la iteración. Suponiendo que una sentencia `while` puede iterar un número arbitrario de veces (incluso 0), tenemos que:

$$\frac{\{I \wedge B\}S\{I\}}{\{I\}\ \texttt{while B do}\ S\ \texttt{end do}\ \{I \wedge \neg B\}}$$

Donde a la proposición I la llameremos invariante.

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.

```
{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
```

Observando el código, cada proceso hace 2 lecturas y dos escrituras en `x`, por lo que, como mínimo, el valor de `x` ha de ser 2; y como máximo, 4.

Notando por l_{ij} a la j-ésima lectura del proceso i y por e_{ij} a la j-ésima escritura del proceso i, ambas referidas a la variable `x`, podemos obtener cualquiera de las siguientes trazas de ejecución:

P1	P2	x	P1	P2	x	P1	P2	x	P1	P2	x
l_{11}	-	0	l_{11}	-	0	l_{11}	-	0	l_{11}	-	0
e_{11}	-	1	-	l_{21}	0	e_{11}	-	1	-	l_{21}	0
-	l_{21}	1	e_{11}	-	1	-	l_{21}	1	e_{11}	-	1
-	e_{21}	2	-	e_{21}	1	-	e_{21}	2	-	e_{21}	1
l_{12}	-	2	l_{12}	-	1	l_{12}	-	2	l_{12}	-	1
e_{12}	-	3	e_{12}	-	2	-	l_{22}	2	-	l_{22}	1
-	l_{22}	3	-	l_{22}	2	e_{12}	-	3	e_{12}	-	2
-	e_{22}	4	-	e_{22}	3	-	e_{22}	3	-	e_{22}	2

Luego los posibles valores resultantes para `x` son: 2, 3 y 4.

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 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.

La copia del fichero `f` en el fichero `g` se podría realizar siguiendo el paradigma productor/consumidor que hemos visto en teoría en el Tema 1, mediante el uso de dos procesos:

- Uno que lea un ítem del fichero `f` y lo escriba en una variable compartida.
- Otro que lea dicha variable compartida y escriba el ítem en el fichero `g`.

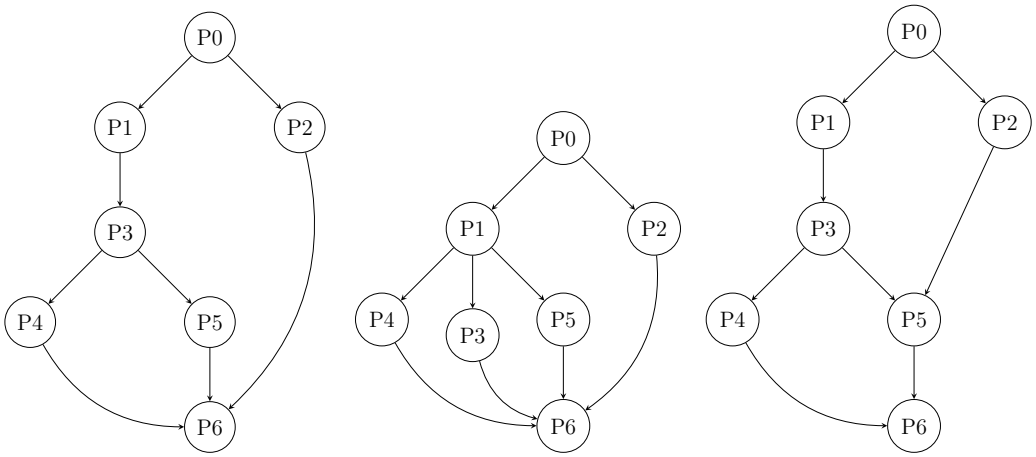
En dicho código, debemos garantizar que:

- El consumidor no lea la variable antes de que el productor escriba en ella.
- En la segunda escritura del productor, debemos esperar a que antes la haya leído el consumidor.
- En la segundaa lectura del consumidor, debemos esperar a que antes haya modificado la variable el productor.

Siguiendo estos pasos, obtendríamos un código como el siguiente:

```
process CopiaFicheros ;
var ant, sig : T ;
begin
  sig = leer(f) ;
  while not fin(f) do begin
    ant = sig ;
    cobegin
      escribir(g, anterior) ;
      sig = leer(f) ;
    coend
  end
end
```

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 2.1.



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

Figura 2.1: Grafos de precedencia del ejercicio 2.1.3.

1. Grafo de precedencia de la figura 2.1a:

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

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

2. Grafo de precedencia de la figura 2.1b:

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

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

3. Grafo de precedencia de la figura 2.1c:

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

Sin embargo, no podemos hacer al 100 % el DAG de la figura 2.1c, ya que tras P3 debemos crear una estructura `cobegin-coend`. Sin embargo, este debe esperar a P2, por lo que la estructura `cobegin-coend` tendrá que esperar a P2, pero es que P4 no necesita que P2 termine.

Por tanto, no se puede programar con creación de hebras de forma estructurada. Sin embargo, podemos ofrecer dos soluciones, cada una que impone algo que el grafo no nos dice:

a) Si obligamos a que P4 también espere a P2, obtendríamos el código:

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

b) Si ahora queremos ejecutar de forma concurrente el flujo que tiene a P1, P3 y P4 con el flujo que tiene a P2, entonces obligamos a que P5 espere a P4 (que no nos lo especifica el DAG, pero lo necesitamos para poder programarlo de forma estructurada):

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

Ejercicio 2.1.4. 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
```

(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 2.4: Programas concurrentes del ejercicio 2.1.4.

1. Programa de la figura 2.4a.

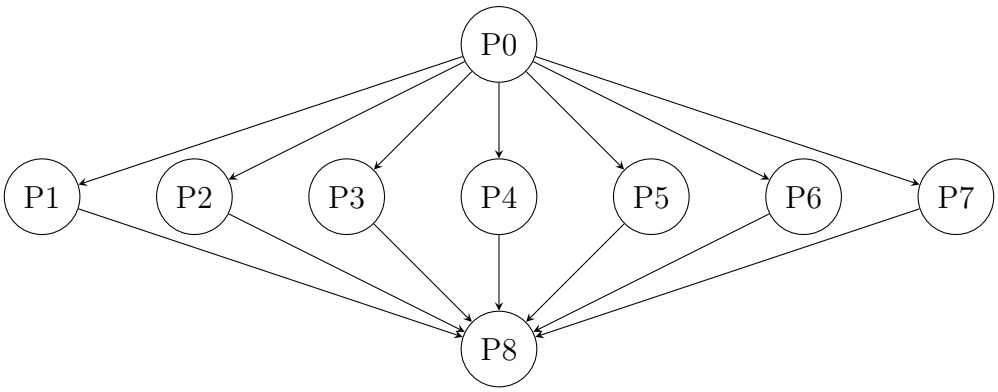


Figura 2.5: DAG para la figura 2.4a.

2. Programa de la figura 2.4b.

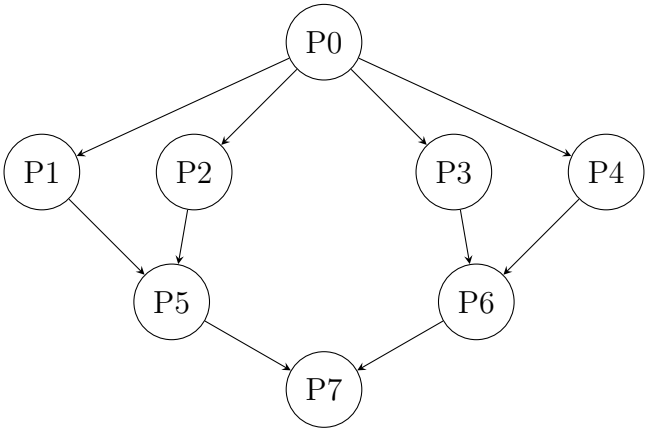


Figura 2.6: DAG para la figura 2.4b.

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).

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:

```
{ 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 2.1.5.

```
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 2.1.6.

```
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 2.1.6.

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:

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

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).

```
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 2.1.6.

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 $T_m(p) = p$.

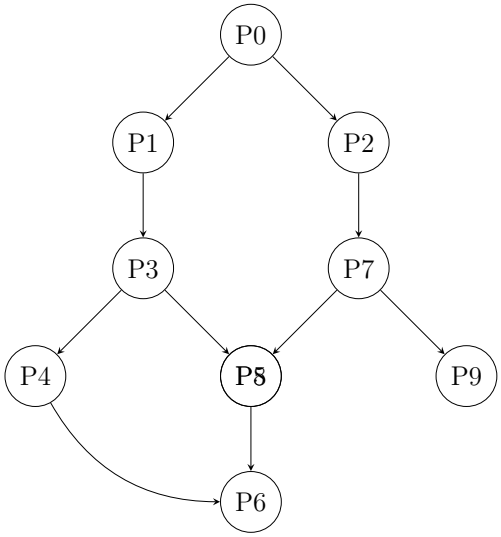
Ejercicio 2.1.7. Supongamos que tenemos un programa con tres matrices (`a`, `b` y `c`) de valores flotantes declaradas como variables globales. La multiplicación secuencial de `a` y `b` (almacenando el resultado en `c`) se puede hacer mediante un procedimiento `MultiplicacionSec` declarado como aparece aquí:

```
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

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

(a) Código del ejercicio 2.1.8.



(b) DAG del ejercicio 2.1.8.

Figura 2.7: Figuras del ejercicio 2.1.8.

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 2.7a), pero que requieren sincronizarse según determinado grafo (ver la figura 2.7b).

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 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:

```
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 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. $\{i < 10\} \quad i = 2 * i + 1 \quad \{\}$
   Obtenemos la postcondición de este triple razonando matemáticamente:

$$\begin{aligned} i &< 10 \\ 2 * i &< 20 \\ i' = 2 * i + 1 &< 21 \end{aligned}$$

   donde hemos notado por i' al nuevo valor que adopta la variable i.

   Por tanto, la postcondición del triple es: $i < 21$.

   Pasamos ahora a demostrar el triple

$$\{i < 10\} \quad i = 2 * i + 1 \quad \{i < 21\}$$

   usando el axioma de asignación:

$$\begin{aligned} \{i < 21\}^i_{2*i+1} \quad i = 2 * i + 1 \quad \{i < 21\} \\ \{i < 21\}^i_{2*i+1} \equiv \{2 * i + 1 < 21\} \equiv \{i < 10\} \end{aligned}$$

   y obteniendo finalmente

$$\{i < 10\} \quad i = 2 * i + 1 \quad \{i < 21\}$$

2. $\{i > 0\} \quad i = i - 1; \quad \{\}$
   Obtenemos la postcondición:

$$\begin{aligned} i &> 0 \\ i' = i - 1 &> -1 \end{aligned}$$

   Y pasamos ahora a demostrar el triple

$$\{i > 0\} \quad i = i - 1 \quad \{i > -1\}$$

   usando el axioma de asignación:

$$\{i > 0\} \equiv \{i - 1 > -1\} \equiv \{i > -1\}^i_{i-1} \quad i = i - 1 \quad \{i > -1\}$$

3. $\{i > j\} \quad i = i + 1;\ j = j + 1 \quad \{\}$
   De forma matemática y notando por i' y j' a las modificaciones de i y j, respectivamente:

$$\begin{aligned} & i > j \\ i' = i + 1 > & j + 1 \\ & i' > j + 1 = j' \\ & i' > j' \end{aligned}$$

   Demostramos ahora el triple

$$\{i > j\} \quad i = i + 1;\ j = j + 1 \quad \{i > j\}$$

   usando la regla de la composición:

$$\frac{\{P\}S_1\{Q\}, \{Q\}S_2\{R\}}{\{P\}S_1; S_2\{R\}}$$

   Por lo que bastará probar los triples

$$\{i > j\}\ i = i + 1\ \{i > j + 1\} \qquad \{i > j + 1\}\ j = j + 1\ \{i > j\}$$

   para tener demostrado el triple que nos interesa.

   a) Probamos el primer triple mediante el axioma de asignación:

$$\{i > j\} \equiv \{i + 1 > j + 1\} \equiv \{i > j + 1\}^{i}_{i+1}\ i = i + 1\ \{i > j + 1\}$$

   b) Y finalmente probamos el segundo trambién mediante el axioma de asignación:

$$\{i > j + 1\} \equiv \{i > j\}^{j}_{j+1}\ j = j + 1\ \{i > j\}$$

   Como ambos son ciertos, el triple que queríamos demostrar también lo es, gracias a la regla de composición.

4. $\{\text{falso}\} \quad a = a + 7; \quad \{\}$
   En este caso, partimos de un estado del programa inalcanzable, por lo que en la postcondición podemos poner cualquier estado del programa, es decir, $\{\text{verdad}\}$.

5. $\{\text{verdad}\} \quad i = 3;\ j = 2 * i \quad \{\}$
   Como partimos de cualquier estado del programa y sólo se realizan asignaciones, es fácil intuir cuál será la postcondición:

$$\begin{aligned} & i = 3 \\ j = 2 * i = & 2 * 3 = 6 \end{aligned}$$

   Pasamos a demostrar el triple

$$\{\text{verdad}\} \quad i = 3;\ j = 2 * i \quad \{i = 3\ \wedge\ j = 6\}$$

   usando la regla de composición, por lo que nos será suficiente probar los triples:

$$\{\text{verdad}\}\ i = 3\ \{i = 3\} \qquad \{i = 3\}\ j = 2 * i\ \{i = 3\ \wedge\ j = 6\}$$

a) Para el primer triple, usamos el axioma de asignación:

$$\{\text{verdad}\} \equiv \{3 = 3\} \equiv \{i = 3\}^{i}_{3}\ i = 3\ \{i = 3\}$$

b) Para el segundo, volvemos a usar el axioma de asignación:

$$\{i = 3\ \wedge\ j = 6\}^{j}_{2*i}\ j = 2 * i\ \{i = 3\ \wedge\ j = 6\}$$
$$\{i = 3\ \wedge\ j = 6\}^{j}_{2*i} \equiv \{i = 3\ \wedge\ 2 * i = 6\} \equiv$$
$$\equiv \{i = 3\ \wedge 2 * 3 = 6\} \equiv \{i = 3\ \wedge\ 6 = 6\} \equiv \{i = 3\}$$

Ambos triples son ciertos, luego por la regla de la composición tenemos demostrado nuestro triple.

6. $\{\text{verdad}\}\quad c = a + b;\ c = {}^{c}\!/_{2}\quad \{\}$
Notando por c' al nuevo valor de c:

$$c = a + b$$
$$c' = {}^{c}\!/_{2} = \frac{a + b}{2}$$

Tratamos de probar el triple

$$\{\text{verdad}\}\quad c = a + b;\ c = {}^{c}\!/_{2}\quad \left\{c = \frac{a + b}{2}\right\}$$

usando la regla de la composición, basta con probar los triples

$$\{\text{verdad}\}\ c = a + b;\ \{c = a + b\} \qquad \{c = a + b\}\ c = {}^{c}\!/_{2};\ \left\{c = \frac{a + b}{2}\right\}$$

a) Para el primero, usamos el axioma de asignación:

$$\{\text{verdad}\} \equiv \{a + b = a + b\} \equiv \{c = a + b\}^{c}_{a+b}\ c = a + b\ \{c = a + b\}$$

b) Para la segunda, también usamos el axioma de asignación:

$$\{c = a + b\} \equiv \left\{\frac{c}{2} = \frac{a + b}{2}\right\} \equiv \left\{c = \frac{a + b}{2}\right\}^{c}_{c/2}\ c = {}^{c}\!/_{2}\ \left\{c = \frac{a + b}{2}\right\}$$

Usando la relga de composición, tenemos demostrado nuestro triple.

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

1. $\{i > 0\}\quad i = i - 1;\quad \{i \geqslant 0\}$
Tratamos de aplicar el axioma de asignación:

$$\{i \geqslant 1\} \equiv \{i - 1 \geqslant 0\} \equiv \{i \geqslant 0\}^{i}_{i-1}\ i = i - 1\ \{i \geqslant 0\}$$

En dicho caso, el triple es cierto siempre y cuando $i \in \mathbb{Z}$, ya que si i es natural y tenemos que $i > 0$, entonces $i \geqslant 1$.

2. $\{x \geqslant 7\} \quad x = x + 3; \quad \{x \geqslant 9\}$
   Tratamos de aplicar el axioma de asignación:

$$\{x \geqslant 6\} \equiv \{x + 3 \geqslant 9\} \equiv \{x \geqslant 9\}^{x}_{x+3}\ x = x + 3; \quad \{x \geqslant 9\}$$

   Como $\{x \geqslant 7\} \rightarrow \{x \geqslant 6\}$, tenemos que el triple es cierto, por la segunda regla de la consecuencia:

$$\frac{\{R\} \rightarrow \{P\}, \{P\}S\{Q\}}{\{R\}S\{Q\}}$$

3. $\{i < 9\} \quad i = 2 * i + 1; \quad \{i \leqslant 20\}$
   Tratamos de aplicar el axioma de asignación:

$$\{i \leqslant 20\}^{i}_{2*i+1}\ i = 2 * i + 1;\ \{i \leqslant 20\}$$

$$\{i \leqslant 20\}^{i}_{2*i+1} \equiv \{2 * i + 1 \leqslant 20\} \equiv \{2 * i \leqslant 19\} \equiv \left\{i \leqslant \frac{19}{2} = 9{,}5\right\}$$

   y llegamos a que el triple

$$\{i \leqslant 9{,}5\}\ i = 2 * i + 1;\ \{i \leqslant 20\}$$

   es cierto. Como $\{i < 9\} \rightarrow \{i \leqslant 9{,}5\}$, tenemos que nuestro triple es cierto, por la segunda regla de la consecuencia.

4. $\{a > 0\} \quad a = a - 7; \quad \{a > -6\}$
   Tratamos de aplicar el axioma de asignación:

$$\{a > 1\} \equiv \{a - 7 > -6\} \equiv \{a > -6\}^{a}_{a-7}\ a = a - 7\ \{a > -6\}$$

   Como $\{a > 0\} \not\rightarrow \{a > 1\}$, nuestro triple es falso. Como contraejemplo, basta considerar:

$$\begin{aligned} a &= 1 \\ a &> 0 \\ a' = a - 7 = 1 - 7 &= -6 \not> -6 \end{aligned}$$

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. $\{P\}C\{Q \vee P\}$
   Es demostrable, ya que $\{Q\} \rightarrow \{Q \vee P\}$ y por la primera regla de la consecuencia, tomando $R = Q \vee P$:

$$\frac{\{P\}C\{Q\}, \{Q\} \rightarrow \{R\}}{\{P\}C\{R\}}$$

2. $\{P \wedge D\}C\{Q\}$
   No podemos demostrarlo, ya que $\{P\} \not\rightarrow \{P \wedge D\}$.

3. $\{P \vee D\}C\{Q\}$
   Es demostrable, ya que $\{P\} \rightarrow \{P \vee D\}$ y por la segunda regla de la consecuencia, tomando $R = P \vee D$:

$$\frac{\{P\} \rightarrow \{R\}, \{R\}C\{Q\}}{\{P\}C\{Q\}}$$

4. $\{P\}C\{Q \vee D\}$
   Al igual que hemos hecho en el apartado 1, es demostrable ya que $\{Q\} \rightarrow \{Q \vee D\}$ y usando la primera regla de la consecuencia.

5. $\{P\}C\{Q \wedge P\}$
   No podemos demostrarlo, ya que $\{Q\} \not\rightarrow \{Q \wedge P\}$.

Ejercicio 2.1.13. Si el triple $\{P\}C\{Q\}$ es demostrable, ¿cuál de los siguientes triples no se puede demostrar?

1. $\{P \wedge D\}C\{Q\}$
   No puede demostrarse, ya que $\{P\} \not\rightarrow \{P \wedge D\}$.

2. $\{P \vee D\}C\{Q\}$
   Puede demostrarse mediante la segunda regla de la consecuencia, ya que $\{P\} \rightarrow \{P \vee D\}$.

3. $\{P\}C\{Q \vee D\}$
   Puede demostrarse mediante la primera regla de la consecuencia, ya que $\{Q\} \rightarrow \{Q \vee D\}$.

4. $\{P\}C\{Q \vee P\}$
   Puede demostrarse mediante la primera regla de la consecuencia, ya que $\{Q\} \rightarrow \{Q \vee P\}$.

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:

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

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 2.1.16. 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.