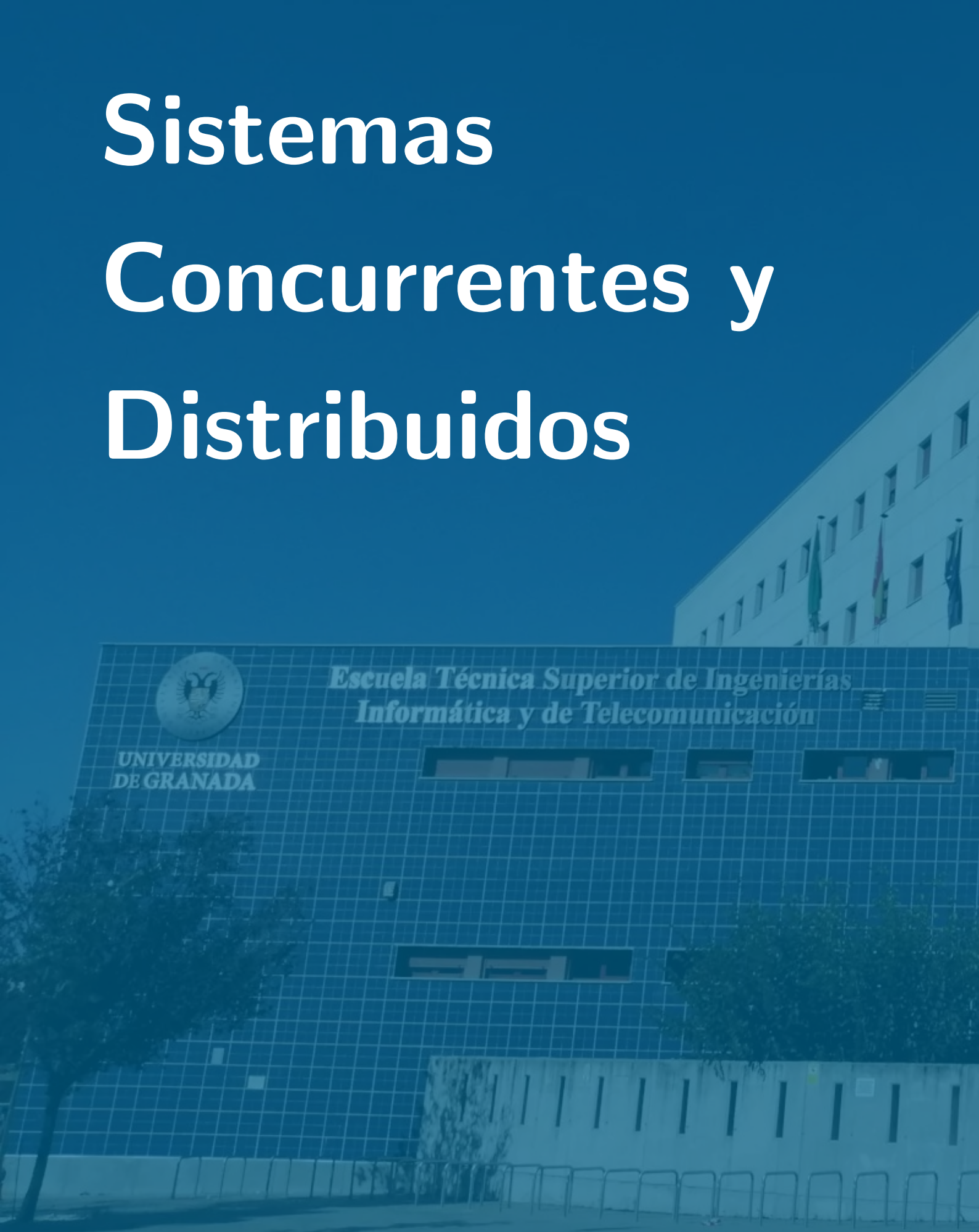
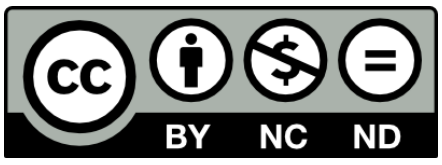
Sistemas Concurrentes y Distribuidos



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. Sincronización en memoria compartida. Monitores

Suponiendo que existe una memoria común para los distintos procesos que ejecutan un programa concurrente, este Capítulo trata sobre la sincronización de los mismos usando para ello instrucciones que usan dicha memoria compartida. Nos centraremos en el uso de los monitores, construcciones de alto nivel que nos ofrecen mayor libertad que los semáforos.

El concepto de semáforo se desarrolló previamente en el Seminario 1 de prácticas[1]. Los semáforos presentan dos grandes limitaciones:

1. Están basados en variables compartidas del programa, por lo que no fomentan la modularidad de los programas, impidiendo su reutilización.

2. Las operaciones de los semáforos (`sem_wait` y `sem_signal`) se encuentran dispersas a lo largo del código del programa concurrente. Además, estas instrucciones no solo afectan al bloque de código en el que se encuentran, sino a cualquier otro bloque que use el mismo semáforo.

En definitiva, los semáforos no son un buen mecanismo de programación concurrente, y además la verificación de programas que usan semáforos es muy complicada.

Era necesario encontrar un nuevo mecanismo de programación concurrente que permitiera la encapsulación de la información y de la sincronización entre procesos, así como programar las operaciones de sincronización (como `wait` o `signal`) dentro de bloques o procedimientos que se ejecuten con instrucciones atómicas, para que las instrucciones de sincronización no se encuentren desperdigadas por el programa. Fue Charles Antony Richard Hoare quien inventó los monitores, concepto en el que ahondaremos a lo largo de este Capítulo.

1.1. Definición de un monitor

La idea básica de monitor es un módulo que contiene un conjunto de variables a las que llamaremos *variables permanentes*[2], de forma que dichas variables solo podrán ser alteradas dentro de los procedimientos del módulo monitor. Garantizaremos que la ejecución de cada uno de esos procedimientos se ejecute la mayor parte del tiempo como una única instrucción atómica, salvo que se produzca algo por lo

[1] Por lo que el lector debería estar familiarizado con ellos.

[2] A pesar de su nombre, no serán constantes, sino que podremos modificar su valor.

que interrumpir la ejecución del procedimiento.

Podemos pensar en un monitor como en un tipo de dato abstracto que define tipos y variables permanentes propias del monitor, así como un conjunto de procedimientos dentro de dicho módulo. No debemos pensar en los monitores como en una clase, ya que no pueden hacer lo mismo que ellas (no se pueden instanciar y tampoco existe polimorfismo o ligadura dinámica).

Ventajas

A continuación, los programas concurrentes estarán formados tanto por procesos que se ejecutarán de forma concurrente, como por monitores, los cuales velarán por la sincronización y acceso a variables compartidas de dichos procesos, de forma que no se produzcan condiciones de carrera o comportamientos indeseados. Podremos modelar tantas relaciones de interacción entre los procesos de un programa concurrente como queramos. De esta forma, el uso de los monitores o de procedimientos asociados a monitores no restringen las posibilidades del modelado de un sistema concurrente.

Los procesos de un programa concurrente no tendrán que llamar a operaciones de sincronización, sino que llamarán a procedimientos del monitor, los cuales realizarán la funcionalidad deseada sobre las variables compartidas garantizando la sincronización entre los procesos.

Además, los monitores nos permiten una alta reusabilidad de código, ya que podremos reutilizar un monitor ya creado para resolver problemas similares. Sin embargo, la reutilización de código no es similar a la usada en programación orientada a objetos mediante instancias de una misma clase, sino que se hará por copias parametrizables: tendremos una definición de un monitor basada en parámetros, y cuando necesitemos usar un monitor, crearemos una copia de dicha definición parametrizándola (pasándole los parámetros que necesitemos para resolver nuestro problema). De esta forma, no es reutilización por instanciación, sino por *parametrización*.

Los procesos que usemos en los progrmas concurrentes no verán el acceso a las variables compartidas, sino que será realizado por los procedimientos del monitor, garantizando que se hacen como deben hacerse, evitando condiciones de carrera. De esta forma, los monitores garantizan la ocultación de las variables compartidas, haciéndolas transparentes a los procesos del sistema concurrente.

Finalmente, existen unos axiomas que nos permiten verificar los programas concurrentes que usen monitores de forma sencilla. Dichas demostraciones estarán basadas en el uso de los invariables globales. Ahondaremos en la verificación de programas concurrentes que utilicen monitores más adelante.

1.1.1. Concepto de monitor

A modo de resumen para comenzar a definir lo que es un monitor, podemos decir que:

- Es un módulo con un conjunto de variables permanentes que solo pueden ser modificadas por los procedimientos del monitor.
- Cada uno de los procedimientos[3] de un monitor se ejecutan en exclusión mutua (garantizando el acceso a las variables compartidas sin condiciones de carrera). Sin embargo, estos no tienen por qué ejecutarse completamente, sino que pueden interrumpirse y en algún momento futuro seguir ejecutándose en exclusión mutua.
- La ejecución de los procedimientos de un monitor modifican el estado interno del mismo (esto es, el conjunto de las variables permanentes asociadas al monitor).
- El estado inicial del monitor (de sus variables permanentes) se establece mediante la ejecución de un procedimiento especial, al que llamaremos *código de inicialización*. Este se ejecuta tras la declaración de una variable de tipo monitor y da valores iniciales a las variables permanentes.

De esta forma, un monitor puede visualizarse como en la tabla 1.1, como un conjunto que englobla:

- Un conjunto de variables, llamadas *variables permanentes*, que no son accesibles desde fuera del monitor.
- Un conjunto de procedimientos que el monitor proporciona como servicio a los procesos de un programa concurrente (para por ejemplo, acceder a las variables permanentes que serán las variables que compartan dichos procesos), llamados *procedimientos exportados* o *exportables*.
- Un procedimiento especial llamado *código de inicialización*, que permite inicializar las variable permanentes.

Variables permanentes
Procedimientos exportados
Código de inicialización

Tabla 1.1: Esquema de un monitor.

Ejemplo. Aunque todavía no entendemos muy bien qué es un monitor, daremos a continuación un ejemplo de uso del mismo, para ilustrar la definición que queremos dar de monitor, pese a que algunas cosas del ejemplo no podamos entenderlas todavía y deberemos dejarlas para más adelante[4].

[3] Podemos pensar en ellos como en los "métodos" de una clase, haciendo hincapié en que los monitores **no son** clases.

[4] Como el tipo de dato `cond`.

En este ejemplo, queremos solventar un problema mediante el paradigma productor/consumidor. Tendremos dos procesos, un productor y un consumidor, de forma que el productor escribirá en un buffer (o vector) que usaremos como cola cíclica (esto es, que si nos pasamos de la posición final, volvemos al inicio y con planificación FIFO), mientras que el consumidor irá leyendo los datos de dicho buffer. Siendo `Buf` una variable de tipo monitor que luego definiremos en este ejemplo, el código del productor y del consumidor será el siguiente (pensando en que tenemos que usar procedimientos del monitor para el acceso a las variables compartidas, en este caso el buffer):

```
Proceso Prod1::
  var d : tipo_dato;

  while true do begin
    d = producir();
    Buf.insertar(d); {mete d en el buffer}
  end do
```

```
Proceso Cons1::
  var x : tipo_dato;

  while true do begin
    Buf.retirar(x); {retira del buffer en x}
    consumir(x);
  end do
```

El código del monitor será el siguiente en pseudo-pascal (hemos omitido el código de inicialización):

```
Monitor Buf
  var
    -elementos_ocupados : int;
    -frente, atras: 0..N-1;
    -no_vacio, no_lleno : cond;

  +insertar(d : tipo_dato);
  +retirar(var x : tipo_dato);
```

Donde vemos 5 variables permanentes: `elementos_ocupados`, que mide la cantidad de posiciones ocupadas del buffer, `frente`, que marca la casilla en la que el productor insertará el próximo dato (por tanto, ha de estar siempre vacía), `atras`, que marca la casilla de la que leerá el consumidor, `no_vacio` y `no_lleno`, variables de tipo `cond`, las cuales aprenderemos lo que hacen más adelante.

Contamos además con dos procedimientos: `insertar`, que inserta un dato en el buffer en caso de que haya hueco (si no hay hueco, se bloquea hasta que el consumidor lea un dato y deje un hueco libre):

```
procedure insertar(d : tipo_dato) begin
  if((frente + 1) mod N = frente) then no_lleno.wait();
  introducir(buf, frente, d);  {inserta d en la posicion frente en el buffer}
```

```
  elementos_ocupados += 1;
  frente = (frente + 1) mod N;
  no_vacio.signal();
end
```

Y con el procedimiento `retir`, que retira un dato del buffer y lo devuelve como resultado del procedimiento, siempre que esto sea posible (es decir, si no hay ningún dato que leer en el buffer, se bloquea esperando a que el productor ponga algún dato):

```
procedure retirar(var x : tipo_dato) begin
  if(frente = atras) then no_vacio.wait();
  eliminar(buf, atras, x);  {inserta buf[atras] en x y lo borra del buffer}
  elementos_ocupados -= 1;
  atras = atras mod N + 1;
  no_lleno.signal();
end
```

Como hemos ya comentado mientras mostrábamos los pseudocódigos del ejemplo, hay que establecer condiciones que identifiquen las dos condiciones inseguras del ejemplo: que el buffer esté lleno o que el buffer esté vacío:

- Si `frente = atras`, entonces el último dato que se ha de consumir está en una casilla vacía en la que el productor escribirá. Se trata de la situación en la que el buffer está vacío. Debemos por tanto, evitar que el consumidor lea un dato del buffer.

- Si `(frente + 1) mod N = atras`, entonces el siguiente dato a introducir en el buffer está justo delante del dato a consumir. Se trata de la situación en la que el buffer está lleno. Debemos por tanto, evitar que el productor introduzca un dato en el buffer[5].

Los procesos del programa llaman a los procedimientos del monitor, y no tienen acceso directo al buffer, por lo que no pueden saber cuándo este está lleno o vacío. De esta forma, lo que sucederá es que los procedimientos internos del monitor realizarán una sincronización interna mediante el uso de llamadas bloqueantes:

- Si el buffer está lleno y el productor se dispone a escribir un dato, quedará el proceso bloqueado hasta que un consumidor lea un dato. Este señalará (`signal`) al productor, desbloqueándolo.

- Si el buffer está vacío y el consumidor se disopne a leer un dato, quedará bloqueado el proceso que ejecute el procedimiento del monitor. Cuando el productor escriba un dato, enviará una señal al consumidor, desbloqueándolo.

Esta funcionalidad se consigue mediante las variables de tipo `cond`. Se verán a continuación, pero para entenderlas por ahora digamos que necesitamos tener una variable

[5]Definimos anteriormente que `frente` siempre apunta a una casilla vacía, por lo que como máximo el buffer tendrá ocupados $N - 1$ elementos.

de tipo `cond` por cada razón por la que queremos bloquear un proceso[6].

El código de los procedimientos es ejecutado por los propios procesos que ejecutan cada proceso (productor o consumidor, en este caso) del programa concurrente. Por tanto, si el productor ejecuta un procedimiento del monitor con un `wait`, dicho proceso se bloqueará y no podrá ejecutar código hasta desbloquearse.

Para que el código que hemos visto funcione adecuadamente, nos falta introducir un último concepto en los monitores, y es que mientras se ejecuta un procedimiento de un monitor, no se puede ejecutar ningún otro, sino que han de ejecutarse en **exclusión mutua**.

1.1.2. Características de programación con monitores

Una vez ilustrado el uso de la herramienta que estamos construyendo en este Capítulo mediante el ejemplo anterior, vamos ahora a introducir la noción de que sólo puede ejecutarse a la vez un único procedimiento de un monitor.

Como ya hemos visto, los procedimientos de los monitores no tienen por qué ejecutarse de principio a fin, sino que un proceso puede comenzar a ejecutar un procedimiento, bloquearse (dejando por tanto libre al monitor) y que otro proceso comience a ejecutar un procedimiento de dicho monitor, sucediéndose un entrelazamiento de las trazas de ejecución de los procedimientos.

Cuando un proceso se encuentra ejecutando un procedimiento del monitor, decimos que el monitor está *ocupado*. En caso contrario, diremos que este está *libre*. Notemos que si un proceso se bloquea mientras ejecuta un procedimiento del monitor, el monitor tiene que quedar libre, ya que si no no habría forma de volver a despertar a dicho proceso (tenemos que ejecutar un `signal` sobre la misma variable `cond` que bloqueó al proceso[7]). La situación de bloquear a un proceso y dejar que entre otro al monitor es delicada y debe hacerse con cuidado, para garantizar que solo haya un único proceso ejecutando un procedimiento del monitor al mismo tiempo.

Los monitores son objetos *pasivos*. Esto es, no tienen una hebra dentro que ejecute su código, sino que simplemente proporciona código (sus procedimientos) a otros procesos para que sean ellos quien ejecuten el código del monitor.

Para implementar una librería con monitores en un lenguaje de programación base, este debe tener la propiedad de ser *reentrante*.

Definición 1.1. Un lenguaje de programación tiene la propiedad de ser reentrante si, siempre que tengamos un proceso ejecutando una función y este se bloquea, sea capaz de conservar la siguiente instrucción a ejecutar y el valor de sus variables

[6]En el caso de productor/consumidor, queremos bloquear un proceso si sucede alguno de los dos puntos superiores, condiciones inseguras, luego nos harán falta dos variables de tipo `cond`. En otros problemas, el número de variables de tipo `cond` podría ser otro.

[7]Se explicará más adelante.

locales tras desbloquearse. Es decir, el proceso no debe enterarse localmente de que nada haya cambiado mientras estaba bloqueado.

Notemos que debemos tener esta propiedad en el lenguaje de programación con el que trabajemos para poder hacer uso de funciones bloqueantes (como `wait`) dentro de los procedimientos de un monitor, algo básico en el funcionamiento de este. Afortunadamente, actualmente todos los lenguajes de programación que encontramos en el mercado son reentrantes.

Instanciación de monitores

El siguiente ejemplo nos ilustra cómo funciona la instanciación de un monitor:

Ejemplo. Aunque los monitores están pensado para programas concurrentes (ya que no tiene sentido su uso en programas secuenciales), usaremos en este ejemplo un monitor en un programa secuencial, ya que sólo nos interesa la forma en la que los monitores se inicializan[8].

Tenemos un programa en el que necesitamos dos variables, las cuales queremos consultar e incrementar mediante un incremento previamente fijado que no cambiará. Para ello, creamos un monitor de acceso a una variable, con parámetros de entrada, para luego poder crear dos copias parametrizadas del mismo. El código del monitor será algo parecido a:

```
1  class monitor VariableProtegida(inicio, incremento : integer);
     var x, inc : integer;

     procedure incremento();
5    begin
       x = x + inc;
     end

     procedure valor(var v : integer);
10   begin
       v = x;
     end

     begin
15     x = inicio; inc = incremento;
     end
```

De esta forma, podemos usar dos copias del monitor de la forma:

```
1  var mv1 : VariableProtegida(0,1);   {empieza en 0 e incrementa en 1}
       mv2 : VariableProtegida(10,4);  {empieza en 10 e incrementa en 4}
       a, b : integer;
   begin
5    mv1.incremento();  {+=1}
```

[8] Además, no hemos terminado de desarrollar cómo es que solo puede ejecutarse a la vez un único procedimiento del monitor, por lo que no entendemos hasta ahora cómo es que sirven para sincronizar programas concurrentes.

```
  mv1.valor(a);       {a=1}
  mv2.incremento();   {+=4}
  mv2.valor(b);       {b=14}
end
```

Exclusión mutua en los procedimientos de un monitor

Si tenemso varios procesos del programa concurrente que quieren hacer uso de procedimientos del monitor a la vez, sólo podremos dejar pasar a un proceso al monitor (suponiendo que este se encuentre libre). Para los otros procesos, almacenamos su llamada al procedimiento.

Para ello, todos los monitores tienen implementada una cola con planificación FIFO, llamada *cola de entrada al monitor*. Si tenemos dos procesos que quieren acceder a un procedimiento de un monitor libre, sólo podrá hacerlo un proceso. La llamada al procedimiento del monitor del otro proceso quedará almacenada en la cola de entrada al monitor, y este pasará a ejecutar el procedimiento deseado una vez el proceso anterior haya dejado libre el monitor.

En esta asignatura, supondremos que la cola de entrada al monitor es suficientemente larga como para albergar a todos los procesos que necesiten esperar a que el monitor quede libre.

Podemos representar la vida de un proceso de un programa concurrente que hace uso de monitores para sincronizar a sus procesos con el siguiente diagrama:

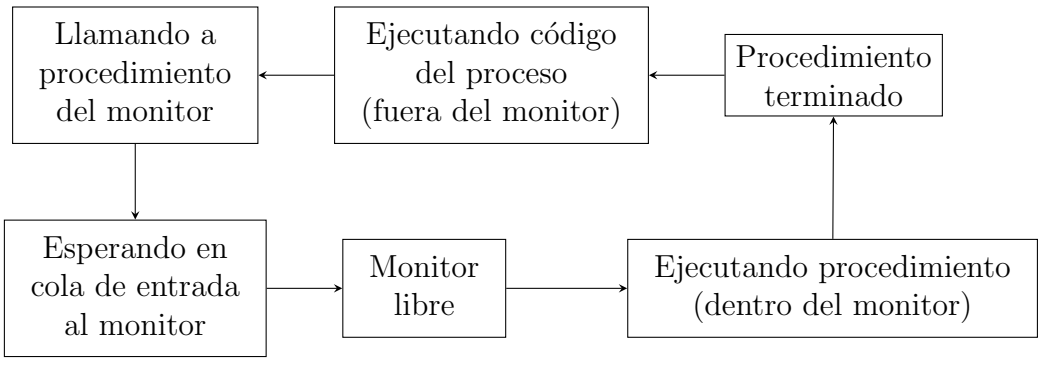


Figura 1.1: Vida de un proceso en un programa concurrente con monitores.

De esta forma, podemos ahora reescribir la descripción gráfica de monitor que hicimos en la tabla 1.1, incluyendo ahora la cola de entrada al monitor, tal y como vemos en la tabla 1.2

Cola del monitor
Variables permanentes
Procedimientos exportados
Código de inicialización

Tabla 1.2: Esquema de un monitor incluyendo la cola de entrada.

1.1.3. Operaciones de sincronización

Las operaciones de sincronización entre los procesos de un programa concurrente se programan, como ya hemos visto, dentro de los procedimientos del monitor. Son instrucciones que permiten detener la ejecución de un procedimiento de un monitor y bloquear en una cola al proceso que ha hecho la llamada del procedimiento del monitor. Tenemos para realizar esta acción dos operaciones principales: `wait` y `signal`.

Sin embargo, las operaciones `wait` y `signal` que manejamos en monitores no se parecen a las que usábamos en los semáforos:

- En los semáforos, la ejecución de `wait` ofrecía la posibilidad de bloquear al proceso, ya que no lo hacía si el entero de dentro del semáforo era mayor estricto que 0. Por contra, en monitores la llamada `wait` siempre será bloqueante.

- Las operaciones `wait` y `signal` eran relativas a un semáforo: hacía falta usar un semáforo por cada razón que tuviéramos dentro de un programa concurrente para bloquear a uno o varios procesos (en el caso del productor/consumidor, usar dos semáforos). Sin embargo, con un solo monitor podremos bloquear procesos por tantas razones como queramos, usando un nuevo tipo de dato.

Tipo de dato `cond`

En los monitores, para poder usar las operaciones de `wait` y `signal`, será necesario utilizar una variable de tipo de dato condición, o `cond`.

Las variables tipo `cond` sólo se encuentran junto con las variables permanentes de un monitor. Estas no se inicializan a ningún valor.

En nuestro monitor, tendremos varias razones por las que querremos bloquear a los procesos concurrentes de nuestro programa por alguna determinada razón hasta que se cumpla una condición determinada. Por ejemplo, en el problema del productor/consumidor:

- Queremos bloquear a cualquier productor que intente escribir si la estructura de datos intermedia que usamos está llena. Desbloquearemos a un proceso productor cuando se vacíe un hueco en dicha estructura.

- Además, queremos bloquear a cualquier consumidor que intente leer de la estructura de datos intermedia cuando esta esté vacía. Desbloquearemos a un consumidor cuando algún productor haya escrito algún dato.

Por cada razón o condición distinta por la que queramos bloquear a los procesos de un programa concurrente en relación a una misma variable compartida (para evitar estados inseguros), crearemos una variable de tipo `cond`. Es decir, una variable por cada una de las razones por las que queramos que esperen los procesos. En el ejemplo del productor/consumidor, son necesarias únicamente dos variables de tipo `cond`.

Las variables de tipo `cond` admiten 4 métodos (aunque sólo recomendamos usar los dos primeros):

wait Bloquea al proceso que ejecuta este método. Dicho proceso pasa a una cola asociada a la variable condición correspondiente con planificación FIFO.

signal En caso de haber algún proceso bloqueado en la cola asociada a la variable condición correspondiente, lo desbloquea. Si esta cola está vacía, es equivalente a una operación nula[9].

queue Devuelve un booleano que indica (`true`) si la cola asociada a la variable condición contiene al menos un proceso bloqueado.

signal_all Desbloquea de una sola vez a todos los procesos bloqueados en la cola asociada a la variable condición. El orden de dicha cola no se mantiene para realizar la petición de acceso al monitor, por lo que se produce competencia entre los procesos para entrar al monitor, incumpliendo la propiedad de equidad entre procesos. Depende de la semántica de las señales del lenguaje[10]. Se recomienda **no usarla**.

De esta forma, la representación gráfica final de un monitor es la que se muestra en la tabla 1.3:

Cola del monitor
Variables permanentes
Variables condición y colas de procesos bloqueados
Procedimientos exportados
Código de inicialización

Tabla 1.3: Esquema de un monitor incluyendo las variables condición.

[9] Esto es, equivalente a la instrucción `;`.
[10] Se explicará más adelante qué es esto.

Semántica desplazante

Como hemos comentado ya, los monitores solo permiten que un único proceso se encuentre ejecutando un procedimiento del mismo. En este caso, decíamos que el monitor está ocupado. En caso de que un proceso que estaba ejecutando el procedimiento ejecute un `wait` (o salga del procedimiento), hay que dejar el monitor libre para dejar pasar a otro. Se trata de un momento muy delicado, ya que se pueden producir condiciones de carrera entre los procesos que quieran conseguir el monitor. Esta situación la hemos solucionado ya con la cola de entrada al monitor, ya que con la planificación FIFO, solo podrá entrar un único proceso al monitor.

Si ahora el proceso nuevo que ejecuta el monitor ejecuta un `signal`, desbloqueará al anterior proceso de la cola de la variable condición correspondiente. Si en dicho momento este proceso sale del procedimiento, dejará el monitor abierto, por lo que podría suceder que un proceso de la cola de entrada al monitor entre antes que el proceso recién desbloqueado, sucediéndose un *robo de señal*, y llegaríamos a una situación de falta de equidad.

Para solucionar este segundo problema, algunos lenguajes implementan una *semántica desplazante* en las señales: el proceso que ejecuta el `signal` le pasa el monitor al proceso que recibió la señal (el primero en la cola de bloqueados de la variable condición correspondiente), sin liberar en ningún momento el monitor, de forma que el proceso señalado tiene prioridad. Se dice que la señal usada con la operación `signal` tiene *semántica desplazante*.

Cabe destacar que **no todos los lenguajes con monitores tienen señales con semántica desplazante**, por lo que en dichos lenguajes pueden sucederse robos de señales. De hecho, para demostrar luego la corrección de nuestros programas concurrentes que usan monitores, supondremos que estamos usando un `signal` que envía una señal con semántica desplazante.

Como comentario final a la descripción de un monitor y para motivar la siguiente sección:

- Se presupone que el programador de monitores es un programador experto, de forma que el compilador en ningún momento se dedicará a comprobar si hemos programado de forma correcta un monitor o un procedimiento de él, más allá de la sintaxis del código.

- No deben programarse operaciones `wait` indebidas ni omitirse operaciones `signal` innecesarias. Para comprobar esto, usaremos nuestro sistema de verificación formal.

1.2. Verificación de programas con monitores

En la verificación de los programas concurrentes que hemos manejado hasta ahora, hemos primero demostrado la corrección secuencial de cada proceso que forma

parte de un programa secuencial, para luego demostrar la no interferencia entre los mismos.

Sin embargo, ahora que introducimos los monitores, esto no podrá ser nunca más así, ya que un programador nunca puede conocer a priori la traza que genera un proceso que forma parte de un programa concurrente con monitores, ya que al ejecutar procedimientos de monitores, estos pueden quedar bloqueados y se ejecutarían en medio instrucciones de otros procesos que podrían alterar las variables compartidas del programa, falseando alguna precondición o poscondición del proceso bloqueado, por lo que tras desbloquearse, no podemos esperar nada de dicho proceso.

Es por tanto que ahora la estrategia a seguir en las demostraciones es mediante un Invariante de Monitor.

1.2.1. Invariante de monitor

Definición 1.2 (Invariante de Monitor)**.** Un Invariante de Monitor (IM) es una relación entre las variables permanentes de un monitor que debe ser cierta en cualquier estado del programa concurrente, excepto cuando un proceso esté ejecutando código de un procedimiento del monitor.

De esta forma, un IM puede no ser cierto durante la ejecución de un procedimiento por parte de un proceso, pero este ha de cumplirse antes y después de la ejecución de dicho procedimiento.

Si conseguimos probar la existencia de un IM en un programa concurrente, entonces bastará con probar cada una de las secciones de código secuenciales entre llamadas a procedimientos del monitor. Para probar finalmente la corrección de los procesos, usaremos que los IM se mantienen antes y después de las llamadas a procedimientos, para conseguir probar finalmente la corrección de cada uno de los procesos. Si nuestro IM estaba relacionado con la solución al problema, como el acceso a variables compartidas estará controlado por los monitores, al final del programa todos los IMs demostrados se seguirán compliendo, por lo que tendremos probada la corrección de nuestro programa concurrente.

Es decir, primero demostraremos que por cada monitor que usamos se verifica un IM, y luego pasaremos a probar la corrección de cada proceso que interviene en el programa concurrente, usando para ello dichos IMs. Finamente, tendremos probado el programa concurrente.

Esquema de demostración

Suponiendo que hemos encontrado una relación matemática entre las variables permanentes de un monitor y queremos probar que se trata de un IM[11], lo primero será probar que IM se cumple en el estado inicial del monitor, esto es, justo después de la inicialización de las variables permanentes, por lo que tendremos que probar que se verifica el **triple de inicialización de variables**:

$$\{V\} \text{ código de inicialización } \{IM\}$$

[11] A continuación, llamaremos a dicha condición `IM`, pese a no haber demostrado que se trate de verdad de un IM.

Posteriormente, deberemos probar que IM se mantiene antes y después de la llamada a cada procedimiento. Es decir, notando por IN a las precondiciones que tenemos antes de la ejecución de un procedimiento y por OUT a las poscondiciones que deseamos tener tras dicho procedimiento, debemos demostrar los **triples de procedimientos del monitor**, es decir, demostrar un triple

$$\{IM \wedge IN\} \text{ procedimiento } \{IM \wedge OUT\}$$

por cada procedimiento que tenga nuestro monitor.

Terminaremos de ver esto más adelante, pero es necesario darnos cuenta de un detalle, y es que si un procedimiento modifica el valor de alguna variable compartida que se usa en otro proceso, debemos demostrar la no interferencia entre dichas instrucciones. Ilustramos esto con el siguiente ejemplo.

Ejemplo. Si tenemos un monitor llamado `Buf` con un procedimientos `retirar(x)`, de forma que modifica el valor del parámetro que le pasamos, ante el siguiente código (si `x` es una variable compartida):

```
1 cobegin y = x; || Buf.retirar(x); coend
```

Tenemos que probar que al cambiar el valor de `x` con el procedimiento `retirar`, no hay interferencia con la instrucción de la izquierda. Es decir, tenemos que probar:

$$NI(pre(y = x), Buf.retirar(x))$$
$$NI(pos(y = x), Buf.retirar(x))$$

Sin embargo, en caso de ejecutar el siguiente código:

```
1 z = x;
cobegin y=z; || Buf.retirar(x); coend
```

No tendríamos que hacerlo, ya que el uso de variables disjuntas nos garantiza la no interferencia entre dichas instrucciones.

1.2.2. Axiomas para operaciones de sincronización con semántica desplazante

Sabemos ya demostrar toda la corrección de un programa secuencial que usa monitores, salvo por un detalle, y es que no sabemos nada sobre cómo demostrar los triples:

$$\{P\}\ c.wait();\ \{Q\}$$
$$\{P\}\ c.signal();\ \{Q\}$$

para cualesquiera asertos P y Q.

En esta subsección, trataremos de dar axiomas para la comprobación de dichos triples, razonándolos de forma intuitiva y mediante el uso de Invariantes de Monitores.

Axioma de operación `wait`

Comenzaremos primero con el triple $\{P\}\ c.wait();\ \{Q\}$. Para necesitar ejecutar una instrucción `wait` en un procedimiento de un monitor, lo que sucede es que estamos cerca de un estado inseguro del programa (intuitivamente, que IM está a punto de incumplirse), pero no llegamos a él, porque para ello ejecutamos esta operación, para impedir que el proceso ejecute una instrucción que falsee el IM. Por tanto, el proceso se bloquea, dejando libre el monitor, por lo que entra otro proceso a ejecutar otro procedimiento.

Solo podremos desbloquear al proceso cuando nos alejemos de dicho estado inseguro, por lo que además de cumplirse el IM, deberá cumplirse una condición un tanto más estricta que el IM (que nos indique que estamos lejos de aquel estado inseguro por el cual se bloqueó el proceso). Dicha condición recibe el nombre de *condición de sincronización*, y la notaremos por C[12].

Resumiendo:

- Antes de ejecutar la operación `wait`, hemos de estar en un estado seguro del programa, por lo que ha de cumplirse el IM.
- Tras ejecutar la operación `wait` (es decir, después de que el proceso haya sido desbloqueado), ha de cumplirse la condición de sincronización C.

Teniendo en cuenta que además se puede cumplir un invariante local al que llamamos L (esto es, relaciones entre variables permanentes del monitor que se cumplen en un determinado momento) antes y después de dicha instrucción `wait`.

De esta forma, acabamos de razonar de forma intuitiva el **Axioma de la operación wait**:

$$\{IM \wedge L\}\ c.wait();\ \{C \wedge L\}$$

Axioma de operación `signal`

Si nos disponemos a ejecutar una instrucción `signal` en nuestro código, es porque el estado del programa se ha alejado de la condición insegura de la que hablábamos en la subsección anterior, que falsearía el valor de verdad de IM. Por tanto, el programa ha llegado a un punto en el que se cumple la condición de sincronización C, y ya puede desbloquear al proceso que anteriormente bloqueó. Tras su desbloqueo, este proceso podría ejecutar una instrucción que volviera a acercarnos a un estado inseguro, pero sin llegar a él (ya que C era suficientemente restrictiva), por lo que como poscondición de la instrucción `signal` no podremos garantizar C, sino sólo podremos asegurar que se sigue cumpliendo IM.

Añadiendo la posibilidad de tener un invariante local L y que si la cola de la variable condición está vacía, la operación `signal` es una instrucción nula, llegamos al **Axioma de la operación signal**:

$$\{\neg vacio(c) \wedge C \wedge L\}\ c.signal();\ \{IM \wedge L\}$$

o equivalentemente:

$$\{c.queue() \wedge C \wedge L\}\ c.signal();\ \{IM \wedge L\}$$

[12]Notemos que según hemos definido C, ha de verificarse que $IM \rightarrow C$.

En caso de cumplirse que $c.queue() = false$, entonces negaría la precondición del triple, haciéndolo la regla cierta por un razonamiento por vacuidad.

Observación. Notemos que el axioma de la operación signal funciona porque hemos supuesto que **tenemos semántica desplazante**, y es que IM se cumple inmediatamente de desbloquear al proceso que tenemos bloqueado, ya que tras ejecutar `signal` hemos cedido el monitor al proceso anteriormente bloqueado, en vez de liberar el monitor y dejar paso a otro proceso cualquiera, donde nada nos garantizaría que C se siguiera cumpliendo tras la ejecución del procedimiento de dicho proceso, pudiendo ahora ejecutar el proceso que se bloqueó bajo una precondición que no es C, lo que podría llevar al programa a adoptar un estado inseguro.

Una vez vistos ya todos los axiomas sobre verificación de operaciones de sincronización de semáforos, estamos listos para desmotrar la corrección de un IM. Lo haremos en el siguiente ejemplo.

Ejemplo. En este ejemplo, queremos programar un monitor que simule el funcionamiento de un semáforo. Para ello, se nos ha ocurrido el siguiente código:

```
1  Monitor Semaforo;
     var s : integer;
         c : cond;

5    procedure P;
     begin
       if s=0 then
         c.wait;
       else
10       null;
       end if
       s = s - 1;
     end

15   procedure V;
     begin
       s = s + 1;
       c.signal;
     end
20

     begin   {código de inicialización}
       s = 0;
     end
```

Donde hemos llamado `P` a la función `sem_wait` del semáforo y por `V` a la función `sem_signal`.

Procedemos a realizar la demostración de que existe un Invariante de Monitor que se mantiene tras la inicialización de las variables permanentes de nuestro monitor y antes y después de cada procedimiento, con la finalidad de poder usar dicho IM en las demostraciones de cualquier programa concurrente que use el semáforo que acabamos de implementar mediante un monitor.

Demostración. Tratamos de demostrar que este monitor tiene como IM el aserto

$$IM \equiv \{s \geqslant 0\}$$

1. Primero, tenemos que demostrar el triple de inicialización de variables:

   $$\{V\}\ s = 0;\ \{s \geqslant 0\}$$

   Como el triple $\{V\}\ s = 0;\ \{s = 0\}$ es cierto por el axioma de asignación y tenemos que $\{s = 0\} \rightarrow \{s \geqslant 0\}$, usando la primera regla de la consecuencia tenemos demostrado el triple.

2. Posteriormente, demostraremos el triple de procedimiento del monitor para el procedimiento `P`: $\{IM\}\ P\ \{IM\}$. Para ello, primero tendremos que probar el triple

   $$\{IM\}\ \texttt{if}\ s = 0\ \texttt{then}\ c.wait;\ \texttt{else}\ null;\ \texttt{end if}\ \{s > 0\}$$

   Luego usaremos la regla del `if`, por lo que será suficiente con probar los triples:

   $$\{IM \wedge s = 0\}\ c.wait;\ \{s > 0\}$$
   $$\{IM \wedge s > 0\}\ null;\ \{s > 0\}$$

   a) Comenzamos por el segundo, por ser más sencillo. Como

   $$\{IM \wedge s > 0\} \equiv \{s \geqslant 0 \wedge s > 0\} \equiv \{s > 0\}$$

   basta probar el triple $\{s > 0\}\ null;\ \{s > 0\}$, que es cierto por el axioma de la sentencia nula.

   b) Para el primer triple, buscamos aplicar el axioma de la operación `wait`, por lo que tenemos que buscar la condición de sincronización. Para ello, buscamos la precondición del `signal` asociado a la misma variable condición, que se encuentra en el procedimiento `V`. Para hallar la precondición de la instrucción `c.signal`, tendremos que demostrar alguna instrucción de dicho procedimiento, con el fin de hallar la precondición.

   Sobre el código de `V`, vemos que antes de `c.signal` se ejecuta una primera instrucción `s=s+1;`. Suponemos que `V` tiene como precondición IM, por lo que buscamos una poscondición para `s=s+1;`:

   $$\{IM\} \equiv \{s \geqslant 0\}\ s = s + 1;$$

   Puede comprobarse con el axioma de asignación que la poscondición buscada es $\{s > 0\}$. Por tanto, esta será la condición de sincronización de la variable condición `c`:

   $$C \equiv \{s > 0\}$$

   Como $\{IM \wedge s = 0\} \equiv \{s = 0\}$, acabamos de probar el primer triple usando el axioma de la operación `wait`:

   $$\{s = 0\}\ c.wait;\ \{s > 0\}$$

Una vez demostrados los dos triples, tenemos probado el triple del `if`, por lo que sólo faltará probar el triple

$$\{s > 0\}\ s = s - 1;\ \{IM\}$$

Para tener probado el triple del procedimiento `P`.

Como $\{IM\} \equiv \{s \geqslant 0\}$, basta aplicar el axioma de asignación, para obtener $\{s > 0\}\ s = s - 1;\ \{s \geqslant 0\}$.

Aplicando finalmente la regla de composición sobre el triple del `if` y este último triple, tenemos ya probado $\{IM\}\ P\ \{IM\}$.

3. Finalmete, hemos de probar el triple $\{IM\}\ V\ \{IM\}$ para garantizar al fin que IM es un IM. Para ello, hemos de probar el triple

$$\{IM\}\ s = s + 1; c.signal;\ \{IM\}$$

Basta con probar los triples

$$\{IM\}\ s = s + 1;\ \{s > 0\}$$
$$\{s > 0\}\ c.signal;\ \{IM\}$$

y aplicar la regla de composición. El primer triple ya lo demostramos en la demostración del triple del procedimiento `P`, luego bastará probar el segundo, el cual es cierto gracias al axioma de la operación `signal`.

Acabamos de probar que $\{IM\}\ V\ \{IM\}$, que era el último procedimiento del monitor, luego IM es un IM.

□

Ejercicio 1.2.1. Se pide demostrar que el siguiente monitor funciona como un semáforo de Habermann.

En un semáforo de Habermann, queremos llevar la cuenta de:

- El número de recursos que han estado disponibles en algún momento, `nv`.
- El número de procesos que han podido hacer uso de un recurso, `np`.
- El número de procesos que han solicitado hacer uso de un recurso, `na`.

```
1  Monitor Semaforo;
     var na, np, nv : int;
         c : cond;

5    procedure P;
     begin
       na = na + 1;
       if(na > nv) then c.wait();
       np = np + 1;
10   end

     procedure V;
```

```
  begin
    nv = nv + 1;
15  if(na > np) then c.signal();
  end

  begin
    na = 0; np = 0; nv = 0;
20 end
```

- Como para poder hacer uso de un recurso hay que haber solicitado acceso a él previamente, siempre tendremos que $na \geqslant np$.
- Como un proceso solo puede hacer uso de un recurso a la vez, el número de recursos que en algún momento han estado disponibles debe ser menor o igual al número de recursos que en algún momento han sido utilizados por algún proceso: $nv \geqslant np$.
- Ahora, destacamos dos casos:
  - Si el número de peticiones para un recurso es mayor que el número de recursos que en algún momento han estado libre, $na \geqslant nv$, entonces es que no se han podido cumplir todas las peticiones, por lo que tienen que haber más procesos que han podido hacer uso de un recurso que recursos disponibles, es decir, $np \geqslant nv$.
  - Por otra parte, si el número de peticiones es menor al número de recursos que en algún momento han estado disponibles, $na \leqslant nv$, entonces todas las peticiones se han podido completar, por lo que $np \geqslant na$.

  Combinando estos dos puntos finales, deducimos que $np \geqslant \text{mín}(na, nv)$.

1.2.3. Regla de la concurrencia para la verificación de programas con monitores

Dado un programa concurrente en el que tenemos n procesos ejecutándose que podemos representar como triples ciertos de Hoare ciertos $\{P_i\}\ S_i\ \{Q_i\}$ con $i \in \{1, \ldots, n\}$ de forma que ninguna variable en P_i o en Q_i es modificada por ningún S_j con $i \neq j$. Si en dicho código tenemos m monitores de forma que para cada uno hemos conseguido probar un IM IM_k con $1 \leqslant k \leqslant m$, entonces podemos aplicar la **regla de concurrencia para programas con monitores**:

$$\frac{\{P_i\}\ S_i\ \{Q_i\} \quad 1 \leqslant i \leqslant n}{\begin{array}{c}\{MI_1 \wedge \ldots \wedge MI_m \wedge P_1 \wedge \ldots \wedge P_n\} \\ cobegin\ S_1\ ||\ S_2\ ||\ \ldots\ ||\ S_n\ coend \\ \{MI_1 \wedge \ldots \wedge MI_m \wedge Q_1 \wedge \ldots \wedge Q_n\}\end{array}}$$

Obteniendo así la verificación de nuestro programa concurrente.

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 (incrementos) en `x`.

- Como cada proceso aumenta dos veces el valor de `x`, el valor de `x` ha de ser, como mínimo, 2.
- Como en total se hacen 4 incrementos, el valor de `x` ha de ser 4 como máximo.

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:

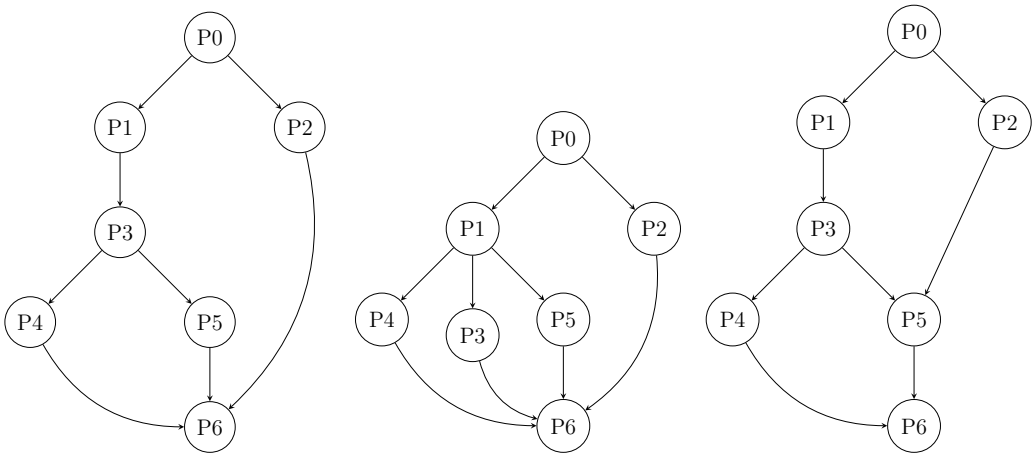
- Uno que lea un ítem del fichero `f` y lo escriba en una variable compartida.
- Otro que lea dicha variable compartida y escriba el ítem en el fichero `g`.

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 segunda 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
```



(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.

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.

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.

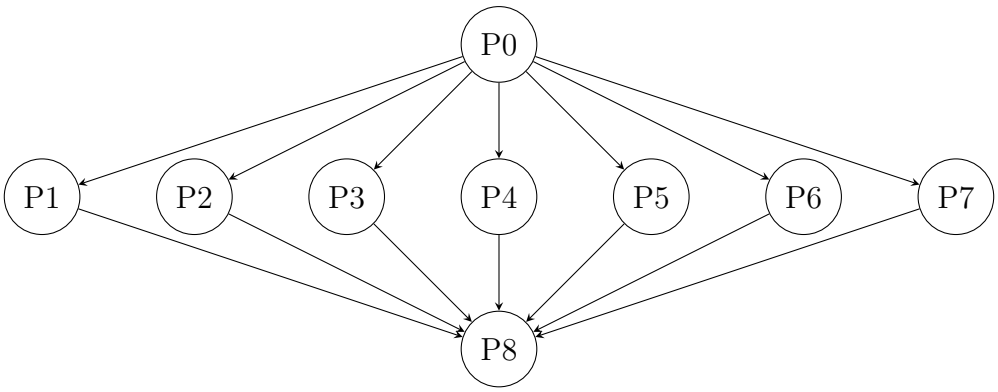


Figura 2.5: DAG para la figura 2.4a.

2. Programa de la figura 2.4b.

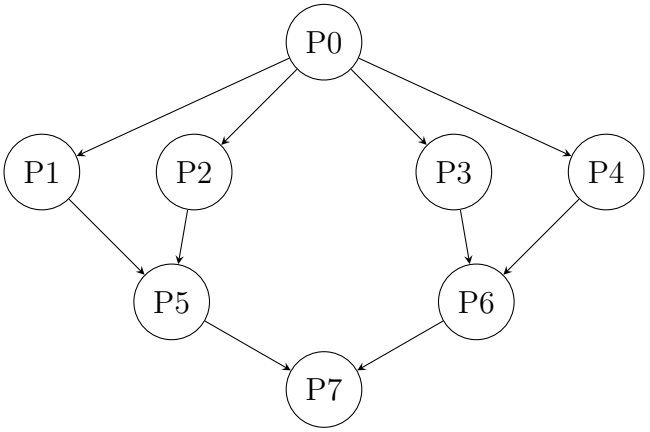


Figura 2.6: DAG para la figura 2.4b.

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).

En primer lugar, notemos que la intrucción 2 siempre se ejecutará antes que la instrucción 3, ya que están ambas en el mismo proceso (el escritor). Por tanto, las secuencia de instrucciones que se pueden dar son las intercalaciones de la instrucción 1 con las otras dos instrucciones; es decir: $A)$ $1-2-3$, $B)$ $2-1-3$ y $C)$ $2-3-1$. El análisis de cada una de ellas está en la Tabla 2.1. Notemos que tanto la opción A como la C son válidas, ya que en ambas se contabiliza el impulso. No obstante, difieren entre sí, puesto que en la opción A se contabiliza el impulso en la hora que termina (imprimiéndose entonces), mientras que en la opción C se contabiliza el impulso en la hora siguiente (no imprimiéndose entonces en esta salida). No obstante, la opción B es incorrecta, ya que no se contabiliza el impulso en la hora que termina ni en la siguiente.

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

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

Opción A			Opción B			Opción C		
Operación	`n`	Salida	Operación	`n`	Salida	Operación	`n`	Salida
–	k	–	–	k	–	–	k	–
`1→ n:=n+1`	$k+1$	–	`2→ write(n)`	k	k	`2→ write(n)`	k	k
`2→ write(n)`	$k+1$	$k+1$	`1→ n:=n+1`	$k+1$	k	`3→ n:=0`	0	k
`3→ n:=0`	0	$k+1$	`3→ n:=0`	0	k	`1→ n:=n+1`	1	k

Tabla 2.1: Tabla de opciones 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.

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:

```
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 2.1.6.

```
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 2.1.6.

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

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

Con estas definiciones, calcular el tiempo que tardará la versión paralela, en los dos siguientes casos. 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 $T_m(p) = p$. Escribe también una comparación cualitativa de los tres tiempos (S, P_1 y P_2).

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

   En este caso, tenemos que hay ganancia de tiempo, ya que las dos instancias de `Sort` no pueden ejecutarse simultáneamente. Por tanto, tenemos que:

   $$P_1 = 2 \cdot T_s(n) + T_m(2n) = 2 \cdot \left(\frac{1}{2} \cdot n^2 - \frac{1}{2} \cdot n \right) + 2n = n^2 - n + 2n = n^2 + n$$

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

   Al poder ejecutarse ahora de forma simultánea, tenemos que:

   $$P_2 = \text{máx}\{T_s(n), T_s(n)\} + T_m(2n) = \left(\frac{1}{2} \cdot n^2 - \frac{1}{2} \cdot n \right) + 2n = \frac{1}{2} \cdot n^2 + \frac{3}{2} \cdot n$$

Como podemos ver, en los tres casos la eficiencia es del orden cuadrático. No obstante, el coeficiente de n^2 es distinto en cada caso, siendo el mayor en la versión secuencial. Deducimos por tanto que las versiones concurrentes son más eficientes que la secuencial, y estas mejoras son significativas para valores de n grandes.

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
                       c[i,j] := c[i,j] + a[i,k]*b[k,j] ;
10             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

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

(a) Procesos concurrentes para calcular por filas.

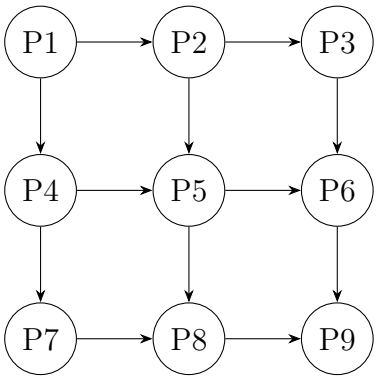
```
var a, b, c : array[1..3,1..3] of
real ;
process CalcularColumna[j : 1..3] ;
    var i, k : integer ;
    begin
        for i := 1 to 3 do begin
            c[i,j] := 0 ;
            for k := 1 to 3 do
                c[i,j] := c[i,j] +
                a[i,k]*b[k,j] ;
        end
    end
end
```

(b) Procesos concurrentes para calcular por columnas.

Figura 2.7: Códigos para el ejercicio 2.1.7.

```
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 2.8: Figuras del ejercicio 2.1.8.

como que elementos distintos de `c` pueden escribirse simultáneamente.

Esta solución puede llevarse a cabo de dos formas (tal y como vimos en la asigantura de AC). Crearemos tres procesos, y cada uno puede calcular una fila de la matriz `c` (Código Fuente 2.7a) o bien cada uno puede calcular una columna de la matriz `c` (Código Fuente 2.7b). No obstante, en la asignatura de AC se vio que, por norma general, es más eficiente calcular por filas que por columnas, ya que en el primer caso se accede a la matriz `a` de forma secuencial, aprovechando así la localidad espacial y provocando un número menor de fallos de caché.

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 2.8a), pero que requieren sincronizarse según determinado grafo (ver la figura 2.8b).

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.

Usaremos para ello un vector de variables booleanas `finalizado`, donde `finalizado[i]` indica si la rutina i ha finalizado o no. Inicialmente estará inicializado a `false`, puesto que ningún proceso ha finalizado. Es decir:

```
var finalizado : array[1..9] of boolean := (false,...,false) ;
```

El procedimiento `EsperarPor` se implementa de la siguiente forma:

```
procedure EsperarPor( i : integer ) ;
begin
    while not finalizado[i] do
    begin
        { no hacer nada }
    end;
end.
```

Respecto a la función `Acabar`, podríamos pensar que tan solo se necesita cambiar el valor de la variable `finalizado[i]` a `true`. No obstante, hemos de tener en cuenta que este programa de ejecuta de forma repetida, por lo que si no se reinicia el vector `finalizado` al final de cada ejecución, en la siguiente ejecución ya no habrá la sincronización necesaria. Por tanto, tras el fin del proceso `P9`, se reinicia el vector `finalizado` a `false`.

```
procedure Acabar( i : integer ) ;
begin
    finalizado[i] := true ;
    if i >= 9 then
        finalizado := (false,...,false) ;
end
```

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
```

En primer lugar, hemos de especificar, en función de n, a qué procesos ha de esperar cada uno, lo cual se hará mediante una matriz compartida. Tenemos que:

```
var espera : array[1..9,1..2] of integer := (
    (-1, -1),    { P1 }
    (1, -1),     { P2 }
    (2, -1),     { P3 }
    (1, -1),     { P4 }
    (2, 4),      { P5 }
    (3, 5),      { P6 }
    (4, -1),     { P7 }
    (5, 7),      { P8 }
    (6, 8)       { P9 }
) ;
Process P[ n : 1..9 ]
begin
    for i := 1 to 2 do
        if espera[n,i] <> -1 then { != -1 }
            EsperarPor( espera[n,i] ) ;

    S_n ;
    Acabar( n ) ;
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 poscondición 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 poscondición del triple es: $i < 21$.

   Pasamos ahora a demostrar el triple siguiente:

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

Usando el axioma de asignación, tenemos que:

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

No obstante, de la definición de Sustitución Textual, tenemos que:

$$\{i < 21\}^{i}_{2*i+1} \equiv \{2 * i + 1 < 21\} \equiv \{i < 10\}$$

Uniendo ambas ecuaciones, obtenemos que el triple es cierto.

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

2. $\{i > 0\} \quad i = i - 1; \quad \{\}$
Obtenemos la poscondición de este triple razonando matemáticamente:

$$i > 0$$
$$i' = i - 1 > -1$$

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

Por tanto, la poscondición del triple es: $i > -1$.
Pasamos ahora a demostrar el triple siguiente:

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

Usando el axioma de asignación, tenemos que:

$$\{i > -1\}^{i}_{i-1} \quad i = i - 1 \quad \{i > -1\}$$

No obstante, de la definición de Sustitución Textual, tenemos que:

$$\{i > -1\}^{i}_{i-1} \equiv \{i - 1 > -1\} \equiv \{i > 0\}$$

Uniendo ambas ecuaciones, obtenemos que el triple es cierto.

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

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:

$$i > j$$
$$i' = i + 1 > j + 1 = j'$$
$$i' > j'$$

Por tanto, la poscondición del triple es: $i > j$. Pasamos a demostrar el triple:

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

Usando la regla de la composición, tenemos que:

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

Identificando Q con $i > j + 1$, tenemos que bastará con probar los triples:

a) $\{i > j\}\ i = i + 1\ \{i > j + 1\}$
Mediante el axioma de asignación, tenemos que:

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

b) $\{i > j + 1\}\ j = j + 1\ \{i > j\}$
Mediante el axioma de asignación, tenemos que:

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

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 poscondición podemos poner cualquier estado del programa, es decir, {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 poscondición:

$$i = 3$$
$$j = 2 * i = 2 * 3 = 6$$

Pasamos a demostrar el triple

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

Usando la regla de composición, 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:

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

No obstante, de la definición de Sustitución Textual, tenemos que:

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

Uniendo ambas ecuaciones, obtenemos que el triple es cierto.

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, tenemos que:

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

   Tratamos por tanto de probar el siguiente 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? (Considerando que $i, x, a \in \mathbb{Z}$)

1. $\{i > 0\} \quad i = i - 1; \quad \{i \geqslant 0\}$
   El siguiente triple sabemos que es cierto:

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

   $\{i > -1\} \to \{i \geqslant 0\}$, luego es cierto por la primera regla de la consecuencia.

2. $\{x \geqslant 7\} \quad x = x + 3; \quad \{x \geqslant 9\}$
   El siguiente triple sabemos que es cierto:

$$\{x \geqslant 7\}\ x = x + 3\ \{x \geqslant 10\}$$

   $\{x \geqslant 10\} \to \{x \geqslant 9\}$, luego es cierto por la primera regla de la consecuencia.

3. $\{i < 9\} \quad i = 2 * i + 1; \quad \{i \leqslant 20\}$
   El siguiente triple sabemos que es cierto:

$$\{i < 9\}\ i = 2 * i + 1\ \{i < 19\}$$

   $\{i < 19\} \to \{i \leqslant 20\}$, luego es cierto por la primera regla de la consecuencia.

4. $\{a > 0\} \quad a = a - 7; \quad \{a > -6\}$

$$\{a > 0\}\ a = a - 7\ \{a > -7\}$$

Pero $\{a > -7\} \not\to \{a > -6\}$, luego este triple no es demostrable.

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 $\{Q\} \to \{Q \vee P\}$ y por la primera regla de la consecuencia, tomando $R = Q \vee P$:

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

Tenemos que se debilita la poscondición.

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

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

Tenemos que se fortalece la precondición.

3. $\{P \vee D\}C\{Q\}$
No es demostrable, porque se debilita la precondición.

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

5. $\{P\}C\{Q \wedge P\}$
No podemos demostrarlo, ya que se fortalece la poscondición.

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

1. $\{P \wedge D\}C\{Q\}$
Sabemos que $\{P \wedge D\} \to \{P\}$, luego puede demostrarse por la segunda regla de la consecuencia (se fortalece la precondición).

2. $\{P \vee D\}C\{Q\}$
No puede demostrarse, porque se debilita la precondición.

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

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

Ejercicio 2.1.14. 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. Tenemos dos posibles trazas de ejecución:
   a) Primero se ejecuta la primera instrucción, por lo que obtendríamos $x = 7$ y $y = 14$.
   b) Primero se ejecuta la segunda instrucción, por lo que obtendríamos $x = 15$ y $y = 10$.
2. Valores finales de x e y si quitamos los símbolos `< >` de instrucción atómica.
   Encontramos cada uno de los dos estados anteriores, además de $x = 7$ y $y = 10$.

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

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

Es decir, queremos comprobar si $R \equiv < x = x-2 >$ con $pre(R) = \{x \geqslant 2\}$ interfiere con los triples siguientes:

1. $\{x \geqslant 0\} \quad < x = x + 3 > \quad \{x \geqslant 3\}$
   Comprobamos en primer lugar su interferencia con la precondición:

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

   Este triple es correcto por la segunda regla de la consecuencia, luego no interfiere con la precondición.

   Comprobemos ahora su interferencia con la poscondición:

   $$\{x \geqslant 3 \wedge x \geqslant 2\} \ < x = x - 2 > \ \{x \geqslant 1\}$$

   En este caso, $\{x \geqslant 1\} \not\to \{x \geqslant 3\}$, luego este triple no es demostrable y R interfiere con la poscondición del triple en cuestión.
2. $\{x \geqslant 0\} \quad < x = x + 3 > \quad \{x \geqslant 0\}$
   Comprobamos en primer lugar su interferencia con la precondición:

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

   Este triple es correcto por la segunda regla de la consecuencia, luego no interfiere con la precondición. Además, como la precondición y la poscondición son iguales, R tampoco interfiere con la poscondición, luego no interfiere con este triple.

3. $\{x \geqslant 7\} \quad < x = x + 3 > \quad \{x \geqslant 10\}$
   Comprobamos en primer lugar su interferencia con la precondición:

   $$\{x \geqslant 7 \wedge x \geqslant 2\} \ < x = x - 2 > \ \{x \geqslant 5\}$$

   No obstante, como $\{x \geqslant 5\} \not\rightarrow \{x \geqslant 7\}$, R interfiere con la precondición de este triple.

4. $\{y \geqslant 0\} \quad < y = y + 3 > \quad \{y \geqslant 3\}$
   R no interfiere con este triple, ya que son variables disjuntas.

5. $\{x \text{ es impar}\} \quad < y = x + 1 > \quad \{y \text{ es par}\}$
   Comprobamos en primer lugar su interferencia con la precondición:

   $$\{x \text{ es impar } \wedge x \geqslant 2\} \ < x = x - 2 > \ \{x \text{ es impar } \wedge x \geqslant 0\}$$

   Por la 2^a regla de la consecuencia, como $\{x \text{ es impar } \wedge x \geqslant 0\} \rightarrow \{x \text{ es impar}\}$, tenemos que es correcto y R no interfiere con la precondición.

   Comprobamos ahora su interferencia con la poscondición:

   $$\{y \text{ es par } \wedge x \geqslant 0\} \ < x = x - 2 > \ \{y \text{ es par } \wedge x \geqslant -2\}$$

   Por la 2^a regla de la consecuencia, como $\{y \text{ es par } \wedge x \geqslant -2\} \rightarrow \{y \text{ es par}\}$, tenemos que es correcto y R no interfiere con la poscondición. Por tanto, R no interfiere con este triple.

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.

Inicialmente, demostraremos los 3 siguientes triples, uno por cada instrucción atómica. Hemos de notar que, en cada uno de ellos, como no sabemos en qué orden se ejecutan, tenemos que incluir en las precondiciones y las poscondiciones todas las posibilidades.

1. El correspondiente a la primera instrucción atómica:

   $$\{x = 0 \vee x = b \vee x = c \vee x = b + c\} \ < x = x + a >$$
   $$\{x = a \vee x = a + b \vee x = a + c \vee x = a + b + c\}$$

Mediante el axioma de asignación, tenemos que:

$$\{x = a \vee x = a + b \vee x = a + c \vee x = a + b + c\}^{a}_{x+a} < x = x + a >$$
$$\{x = a \vee x = a + b \vee x = a + c \vee x = a + b + c\}$$

No obstante, de la definición de Sustitución Textual, tenemos:

$$\{x = a \vee x = a + b \vee x = a + c \vee x = a + b + c\}^{a}_{x+a} \equiv$$
$$\equiv \{x + a = a \vee x + a = a + b \vee x + a = a + c \vee x + a = a + b + c\} \equiv$$
$$\equiv \{x = 0 \vee x = b \vee x = c \vee x = b + c\}$$

Por tanto, el triple en cuestión es cierto.

2. El correspondiente a la segunda instrucción atómica:

$$\{x = 0 \vee x = a \vee x = c \vee x = a + c\} \ < x = x + b >$$
$$\{x = b \vee x = a + b \vee x = b + c \vee x = a + b + c\}$$

Es cierto, y su demostración es análoga al primer caso.

3. El correspondiente a la tercera instrucción atómica:

$$\{x = 0 \vee x = b \vee x = a \vee x = a + b\} \ < x = x + c >$$
$$\{x = c \vee x = a + c \vee x = b + c \vee x = a + b + c\}$$

Es cierto, y su demostración es análoga al primer caso.

Seguidamente, tenemos que ver que dichos 3 triples están libres de interferencias. Para ello, hemos de probar 12 triples, ya que hay 3 instrucciones atómicas, cada una de ellas con 2 asertos, por lo que por cada instrucción hemos de comprobar 4 asertos:

$$NI(x = 0 \vee x = a \vee x = c \vee x = a + c, < x = x + a >)$$
$$NI(x = b \vee x = a + b \vee x = b + c \vee x = a + b + c, < x = x + a >)$$
$$NI(x = 0 \vee x = b \vee x = a \vee x = a + b, < x = x + a >)$$
$$NI(x = c \vee x = a + c \vee x = b + c \vee x = a + b + c, < x = x + a >)$$

$$NI(x = 0 \vee x = b \vee x = c \vee x = b + c, < x = x + b >)$$
$$NI(x = a \vee x = a + b \vee x = a + c \vee x = a + b + c, < x = x + b >)$$
$$NI(x = 0 \vee x = b \vee x = a \vee x = a + b, < x = x + b >)$$
$$NI(x = c \vee x = a + c \vee x = b + c \vee x = a + b + c, < x = x + b >)$$

$$NI(x = 0 \vee x = b \vee x = c \vee x = b + c, < x = x + c >)$$
$$NI(x = a \vee x = a + b \vee x = a + c \vee x = a + b + c, < x = x + c >)$$
$$NI(x = 0 \vee x = a \vee x = c \vee x = a + c, < x = x + c >)$$
$$NI(x = b \vee x = a + b \vee x = b + c \vee x = a + b + c, < x = x + c >)$$

Demostremos ahora el primero, ya que el resto son idénticos.

$$\begin{aligned} &NI(x = 0 \vee x = a \vee x = c \vee x = a + c, < x = x + a >) \equiv \\ &\quad \equiv \{(x = 0 \vee x = a \vee x = c \vee x = a + c) \wedge (x = 0 \vee x = b \vee x = c \vee x = b + c)\} \\ &\qquad < x = x + a > \{x = 0 \vee x = a \vee x = c \vee x = a + c\} \equiv \\ &\quad \equiv \{x = 0 \vee x = c\} < x = x + a > \{x = 0 \vee x = a \vee x = c \vee x = a + c\} \end{aligned}$$

Este triple efectivamente es cierto, lo cual se puede demostrar empleando en primer lugar el Axioma de Sustitución y, posteriormente, la primera regla de la consecuencia.

Por tanto, y tras aplicar la Regla de la Composición Concurrente, tenemos de forma directa que:

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

Ejercicio 2.1.17. El siguiente triple:

$$\begin{gathered} \{x = 0 \wedge y = 0 \wedge z = 0\} \\ < x = z + a > \; || \; < y = x + b > \\ \{(x = a) \wedge (y = b \vee y = a + b) \wedge z = 0\} \end{gathered}$$

(a) Es indemostrable salvo que se cumpla siempre que $a = 0$.

(b) El triple anterior es demostrable para cualquier valor de las variables a o b.

(c) Es indemostrable salvo que se cumpla siempre que $b = 0$.

(d) Es indemostrable salvo que se cumpla siempre que $a = 0 \wedge b = 0$.

Veamos si podemos demostrarlo. Para ello, notamos cada instrucción atómica de la siguiente forma:

$$\begin{aligned} S_1 &=< x = z + a > \\ S_2 &=< y = x + b > \end{aligned}$$

Veamos cuál ha de ser la precondición de cada instrucción atómica:

$$\begin{aligned} P_1 &= x = 0 \wedge (y = 0 \vee y = b) \wedge z = 0 \\ P_2 &= (x = 0 \vee x = a) \wedge y = 0 \wedge z = 0 \end{aligned}$$

Veamos cuál ha de ser la poscondición de cada instrucción atómica:

$$\begin{aligned} Q_1 &= x = a \wedge (y = 0 \vee y = b) \wedge z = 0 \\ Q_2 &= [(x = 0 \wedge y = b) \vee (x = a \wedge y = a + b)] \wedge z = 0 \end{aligned}$$

Vemos por tanto que ambos triples son ciertos:

1. $\{P_1\}S_1\{Q_1\}$

   De la definición de Sustitución Textual, tenemos que:

   $$\{x = a \wedge (y = 0 \vee y = b) \wedge z = 0\}^{x}_{z+a} \equiv \{z + a = a \wedge (y = 0 \vee y = b) \wedge z = 0\} \equiv$$
   $$\equiv \{(y = 0 \vee y = b) \wedge z = 0\}$$

   Por tanto, del Axioma de Asignación, tenemos que:

   $$\{(y = 0 \vee y = b) \wedge z = 0\} < x = z + a > \{x = a \wedge (y = 0 \vee y = b) \wedge z = 0\}$$

   Finalmente, usando la segunda regla de la consecuencia, como se tiene que $\{P_1\} \rightarrow \{(y = 0 \vee y = b) \wedge z = 0\}$, tenemos que el triple es cierto.

2. $\{P_2\}S_2\{Q_2\}$

   De la definición de Sustitución Textual, tenemos que:

   $$\{[(x = 0 \wedge y = b) \vee (x = a \wedge y = a + b)] \wedge z = 0\}^{y}_{x+b} \equiv$$
   $$\equiv \{[(x = 0 \wedge x + b = b) \vee (x = a \wedge x + b = a + b)] \wedge z = 0\} \equiv$$
   $$\equiv \{(x = 0 \vee x = a) \wedge z = 0\}$$

   Por tanto, del Axioma de Asignación, tenemos que:

   $$\{(x = 0 \vee x = a) \wedge z = 0\} < y = x + b >$$
   $$\{[(x = 0 \wedge y = b) \vee (x = a \wedge y = a + b)] \wedge z = 0\}$$

   Por tanto, usando la segunda regla de la consecuencia, como se tiene que $\{P_2\} \rightarrow \{(x = 0 \vee x = a) \wedge z = 0\}$, tenemos que el triple es cierto.

Veamos ahora que no interfieren entre sí. Como tenemos dos instrucciones atómicas, cada una con dos asertos, hemos de comprobar 4 asertos:

$$NI(P_2, S_1) \equiv \{P_1 \wedge P_2\}S_1\{P_2\}$$
$$NI(Q_2, S_1) \equiv \{P_1 \wedge Q_2\}S_1\{Q_2\}$$
$$NI(P_1, S_2) \equiv \{P_2 \wedge P_1\}S_2\{P_1\}$$
$$NI(Q_1, S_2) \equiv \{P_2 \wedge Q_1\}S_2\{Q_1\}$$

Veamos por tanto en qué queda cada uno de ellos:

$$NI(P_2, S_1) \equiv \{x = 0 \wedge y = 0 \wedge z = 0\} < x = z + a > \{(x = 0 \vee x = a) \wedge y = 0 \wedge z = 0\}$$
$$NI(Q_2, S_1) \equiv \{x = 0 \wedge y = b \wedge z = 0\} < x = z + a >$$
$$\{[(x = 0 \wedge y = b) \vee (x = a \wedge y = a + b)] \wedge z = 0\}$$
$$NI(P_1, S_2) \equiv \{x = 0 \wedge y = 0 \wedge z = 0\} < y = x + b > \{x = 0 \wedge (y = 0 \vee y = b) \wedge z = 0\}$$
$$NI(Q_1, S_2) \equiv \{x = a \wedge y = 0 \wedge z = 0\} < y = x + b > \{x = a \wedge (y = 0 \vee y = b) \wedge z = 0\}$$

Intentemos demostrar el segundo. Usando la definición de Sustitución Textual, tenemos que:

$$\{[(x = 0 \wedge y = b) \vee (x = a \wedge y = a + b)] \wedge z = 0\}^{x}_{z+a} \equiv$$
$$\equiv \{[(z + a = 0 \wedge y = b) \vee (z + a = a \wedge y = a + b)] \wedge z = 0\} \equiv$$
$$\equiv \{y = a + b \wedge z = 0\}$$

Por tanto, del Axioma de Asignación, tenemos que:

$$\{y = a+b \wedge z = 0\} < x = z+a > \{[(x = 0 \wedge y = b) \vee (x = a \wedge y = a+b)] \wedge z = 0\}$$

No obstante, de forma general, tenemos que $\{x = 0 \wedge y = b \wedge z = 0\} \not\rightarrow \{y = a + b \wedge z = 0\}$, por lo que $NI(Q_2, S_1)$ no es demostrable. No obstante, si $a = 0$, entonces sí que sería demostrable. Supongamos por tanto a partir de ahora que $\underline{a = 0}$.

Intentemos ahora demostrar el cuarto. Usando la definición de Sustitución Textual, tenemos que:

$$\{x = a \wedge (y = 0 \vee y = b) \wedge z = 0\}^{y}_{x+b} \equiv$$
$$\equiv \{x = a \wedge (x + b = 0 \vee x + b = b) \wedge z = 0\} \equiv$$
$$\equiv \{x = a \wedge (x = -b \vee x = 0) \wedge z = 0\} \equiv \{x = 0 \wedge z = 0\}$$

donde en la última igualdad hemos usado que $\underline{a = 0}$. Por tanto, del Axioma de Asignación, tenemos que:

$$\{x = 0 \wedge z = 0\} < y = x + b > \{x = a \wedge (y = 0 \vee y = b) \wedge z = 0\}$$

Además, como $\underline{a = 0}$, tenemos que $\{x = 0 \wedge z = 0\} \rightarrow \{x = a \wedge y = 0 \wedge z = 0\}$, tenemos que es cierto. Por tanto, $NI(P_1, S_2)$ es demostrable.

Los otros dos triples son análogamente ciertos, por lo que podemos aplicar la regla de la composición concurrente y llegar al siguiente triple:

$$\{x = 0 \wedge y = 0 \wedge z = 0\}$$
$$< x = z + a > \,||\, < y = x + b >$$
$$\{x = a \wedge y = a + b \wedge z = 0\}$$

Este es el triple que queríamos demostrar, suponiendo que $\underline{a = 0}$. Por tanto, la respuesta correcta es la **a)**, ya que no es demostrable salvo que se cumpla siempre que $a = 0$.

Ejercicio 2.1.18. Suponer que $\{suma > 1\}\ suma = suma + 4\ \{suma > 5\}$ es demostrable, entonces: ¿cuál de los siguientes triples es también demostrable? (indicar por qué)

1. $\{suma > 2\}\ suma = suma + 4\ \{suma > 5\}$.
   Es demostrable, ya que $\{suma > 2\} \rightarrow \{suma > 1\}$ y podemos aplicar la segunda regla de la consecuencia.

2. $\{suma \geqslant 1\}\ suma = suma + 4\ \{suma > 5\}$.
   No es demostarble, ya que debilita la precondición.

3. $\{suma > 0\}\ suma = suma + 4\ \{suma > 5\}$.
   No es demostarble, ya que debilita la precondición.

4. $\{suma > 1\}\ suma = suma + 4\ \{suma > 6\}$.
   No es demostrable, ya que fortalece la poscondición.

Ejercicio 2.1.19. Suponer que $\{x < y\}\ C_1\ \{u < v\}$ es demostrable, entonces: ¿cuáles de los siguientes triples son también demostrables? (indicar por qué)

1. $\{x \leqslant y\}\ C_1\ \{u < v\}$.
   No es demostrable, ya que debilita la precondición.

2. $\{x \leqslant y - 2\}\ C_1\ \{u < v\}$.
   Es demostrable, ya que $\{x \leqslant y - 2\} \rightarrow \{x + 2 \leqslant y\} \rightarrow \{x < y\}$, y mediante la segunda regla de la consecuencia se tiene que es cierto.

3. $\{x \leqslant y\}\ C_1\ \{u \leqslant v\}$.
   El triple $\{x < y\}\ C_1\ \{u \leqslant v\}$ sí que es demostrable ya que relaja la poscondición, pero el triple que se nos dice no es demostrable, ya que también relaja la precondición. Como además no tenemos relación entre x y u ni entre y y v, no podemos inferir nada.

4. $\{x < y\}\ C_1\ \{u < v - 2\}$.
   No es demostrable, ya que fortalecemos la poscondición.

Ejercicio 2.1.20. Seleccionar el valor correcto de las 2 variables (x e y) después de ejecutarse el siguiente programa concurrente:

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

(a) $x = 7$ y $y = 14$.

(b) $x = 5$ y $y = 10$.

(c) $x = -7$ y $y = 14$.

(d) $x = -3$ y $y = 10$.

Numeramos las instrucciones atómicas de la siguiente forma:

1. `<x=x+y>`
2. `<y=x*y>`
3. `<x=x-y>`

Veamos ahora, en función del orden de ejecución, cuál sería el valor de las variables x e y:

- 1, 2, 3: $x = -7$ y $y = 14$.
- 1, 3, 2: $x = 5$ y $y = 10$.
- 3, 1, 2: $x = 5$ y $y = 10$.

- 2, 1, 3: $x = 5$ y $y = 10$.
- 2, 3, 1: $x = 5$ y $y = 10$.
- 3, 2, 1: $x = 9$ y $y = 6$.

Por tanto, las respuestas b y c son correctas.

Ejercicio 2.1.21. El siguiente código concurrente no puede ser demostrado directamente con la lógica de aserciones (pre y poscondiciones). Elegir la respuesta que explica correctamente la razón de que ocurra esto.

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

(a) Porque la poscondición que se propone $\{x = 2 * a\}$ es falsa.

(b) Porque falta incluir la posibilidad de que el valor final de x sea también $\{x = a\}$.

(c) Porque al aplicar directamente la regla de inferencia de la *composición concurrente* utilizo unas condiciones (pre y post-condiciones) demasiado débiles.

(d) Porque tengo que incluir en los asertos el valor del contador de programa de cada procesador.

Notamos cada instrucción atómica de la siguiente forma:

$$S_1 = S_2 =< x = x + a >$$

Veamos cuál ha de ser la precondición de cada instrucción atómica:

$$P_1 = P_2 = x = 0 \vee x = a$$

Veamos cuál ha de ser la poscondición de cada instrucción atómica:

$$Q_1 = Q_2 = x = a \vee x = 2a$$

Veamos ahora que cada triple es cierto. Como son los mismos, hemos de demostrar:

$$\{x = 0 \vee x = a\} < x = x + a > \{x = a \vee x = 2a\}$$

Este se demuestra de forma directa. Además, también hemos de demostrar que no interfieren entre sí. Tenemos que demostrar:

$$NI(P_1, S_2) \equiv \{x = 0 \vee x = a\} < x = x + a > \{x = 0 \vee x = a\}$$
$$NI(Q_1, S_2) \equiv \{x = a\} < x = x + a > \{x = a \vee x = 2a\}$$

Estos también son ciertos, por lo que podemos aplicar la regla de la composición concurrente y llegar al siguiente triple:

$$\{x = 0 \vee x = a\} cobegin < x = x + a >; < x = x + a > coend; \{x = a \vee x = 2a\}$$

Por la primera regla de la consecuencia, podemos debilitar la precondición, llegando al siguiente triple:

$$\{x = 0\} cobegin < x = x + a >; < x = x + a > coend; \{x = a \vee x = 2a\}$$

No obstante, la poscondición no se puede debilitar, por lo que la respuesta correcta es la **b)**, ya que falta incluir en los asertos el valor final de x sea también $\{x = a\}$.

Ejercicio 2.1.22. Estudiar cuáles son los valores finales de las variables `x` e `y` en el siguiente programa. Insertar los asertos adecuados entre llaves, antes y después de cada sentencia, para poder obtener una traza de demostración del programa, que incluya en su último aserto los valores finales de las variables.

```
1 int x = c1;
  int y = c2;
  x = x + y;
  y = x * y;
5 x = x - y;
```

Tenemos que cada triple, por orden, es:

$$\{x = c_1 \wedge y = c_2\}$$
$$x = x + y$$
$$\{x = c_1 + c_2 \wedge y = c_2\}$$
$$y = x * y$$
$$\{x = c_1 + c_2 \wedge y = (c_1 + c_2) \cdot c_2\}$$
$$x = x - y$$
$$\{x = (c_1 + c_2) - (c_1 + c_2) \cdot c_2 = (c_1 + c_2) \cdot (1 - c_2) \wedge y = (c_1 + c_2) \cdot c_2\}$$

Ejercicio 2.1.23. Demostrar que el siguiente triple es cierto:

$$\{x = 0\}$$
$$\texttt{cobegin}$$
$$< x = x + 1 > \,||\, < x = x + 2 > \,||\, < x = x + 4 >$$
$$\texttt{coend}$$
$$\{x = 7\}$$

Notamos cada instrucción atómica de la siguiente forma:

$$S_1 =< x = x + 1 >$$
$$S_2 =< x = x + 2 >$$
$$S_3 =< x = x + 4 >$$

Veamos cuál ha de ser la precondición de cada instrucción atómica:

$$P_1 = x = 0 \vee x = 2 \vee x = 4 \vee x = 6$$
$$P_2 = x = 0 \vee x = 1 \vee x = 4 \vee x = 5$$
$$P_3 = x = 0 \vee x = 1 \vee x = 2 \vee x = 3$$

Veamos cuál ha de ser la poscondición de cada instrucción atómica:

$$Q_1 = x = 1 \vee x = 3 \vee x = 5 \vee x = 7$$
$$Q_2 = x = 2 \vee x = 3 \vee x = 6 \vee x = 7$$
$$Q_3 = x = 4 \vee x = 5 \vee x = 6 \vee x = 7$$

Cada triple es directamente cierto por el Axioma de Asignación. Veamos ahora que no interfieren entre sí. Tenemos que demostrar:

$$NI(P_2, S_1) \equiv \{x = 0 \vee x = 4\} < x = x + 1 > \{x = 0 \vee x = 1 \vee x = 4 \vee x = 5\}$$
$$NI(Q_2, S_1) \equiv \{x = 2 \vee x = 6\} < x = x + 1 > \{x = 2 \vee x = 3 \vee x = 6 \vee x = 7\}$$
$$NI(P_3, S_1) \equiv \{x = 0 \vee x = 2\} < x = x + 1 > \{x = 0 \vee x = 1 \vee x = 2 \vee x = 3\}$$
$$NI(Q_3, S_1) \equiv \{x = 4 \vee x = 6\} < x = x + 1 > \{x = 4 \vee x = 5 \vee x = 6 \vee x = 7\}$$

$$NI(P_1, S_2) \equiv \{x = 0 \vee x = 4\} < x = x + 2 > \{x = 0 \vee x = 2 \vee x = 4 \vee x = 6\}$$
$$NI(Q_1, S_2) \equiv \{x = 1 \vee x = 5\} < x = x + 2 > \{x = 1 \vee x = 3 \vee x = 5 \vee x = 7\}$$
$$NI(P_3, S_2) \equiv \{x = 0 \vee x = 1\} < x = x + 2 > \{x = 0 \vee x = 1 \vee x = 2 \vee x = 3\}$$
$$NI(Q_3, S_2) \equiv \{x = 4 \vee x = 5\} < x = x + 2 > \{x = 4 \vee x = 5 \vee x = 6 \vee x = 7\}$$

$$NI(P_1, S_3) \equiv \{x = 0 \vee x = 2\} < x = x + 4 > \{x = 0 \vee x = 2 \vee x = 4 \vee x = 6\}$$
$$NI(Q_1, S_3) \equiv \{x = 1 \vee x = 3\} < x = x + 4 > \{x = 1 \vee x = 3 \vee x = 5 \vee x = 7\}$$
$$NI(P_2, S_3) \equiv \{x = 0 \vee x = 1\} < x = x + 4 > \{x = 0 \vee x = 1 \vee x = 4 \vee x = 5\}$$
$$NI(Q_2, S_3) \equiv \{x = 2 \vee x = 3\} < x = x + 4 > \{x = 2 \vee x = 3 \vee x = 6 \vee x = 7\}$$

Todos estos son ciertos, por lo que podemos aplicar la regla de la composición concurrente y llegar al siguiente triple:

$$\{x = 0\}$$
$$\texttt{cobegin}$$
$$< x = x + 1 > || < x = x + 2 > || < x = x + 4 >$$
$$\texttt{coend}$$
$$\{x = 7\}$$

Ejercicio 2.1.24. Dada la siguiente construcción de composición concurrente `P`:

$$\texttt{cobegin}$$
$$< x = x - 1 >; < x = x + 1 >; || < y = y - 1 >; < y = y + 1 >;$$
$$\texttt{coend}$$

demostrar que se cumple la invarianza de $\{x = y\}$, es decir, que $\{x = y\}\ P\ \{x = y\}$ es un triple cierto.

Para ello, comenzamos demostrando los siguientes triples (se ha de mantener la invarianza en el código secuencial)

$$\{x = y\}\ x = x - 1; x = x + 1;\ \{x = y\}$$
$$\{x = y\}\ y = y - 1; y = y + 1;\ \{x = y\}$$

1. Respecto al primero, tenemos que los siguientes triples son ciertos:

$$\{x = y\}\ x = x - 1\ \{x + 1 = y\}$$
$$\{x + 1 = y\}\ x = x + 1\ \{x = y\}$$

   Usando la regla de la composición, se tiene.

2. Respecto al segundo, tenemos que los siguientes triples son ciertos:

$$\{y = x\}\ y = y - 1\ \{y + 1 = x\}$$
$$\{y + 1 = x\}\ y = y + 1\ \{y = x\}$$

   Usando la regla de la composición, se tiene.

Ahora, los triples son libres de interferencia por tener variables disjuntas. Podemos aplicar por tanto la regla de la composición concurrente, llegando a lo que queríamos probar:

$$\{x = y\}$$
$$\texttt{cobegin}$$
$$< x = x - 1 >; < x = x + 1 >; || < y = y - 1 >; < y = y + 1 >;$$
$$\texttt{coend}$$
$$\{x = y\}$$

Ejercicio 2.1.25. Usando la regla de la conjunción, demostrar que

$$\{i > 2\}\ i = 2 * i\ \{i > 4\}$$

Ejercicio 2.1.26. Se dan los siguientes triples de Hoare:

$$\{j > 1\}\ i = i + 2;\ j = j + 3;\ \{j > 4\}$$
$$\{i > 2\}\ i = i + 2;\ j = j + 3;\ \{i > 4\}$$

Demostrar que estos triples implican que

$$\{j > 1 \wedge i > 2\}\ i = i + 2;\ j = j + 3\ \{j > 4 \wedge i > 4\}$$

¿Qué regla se debe utilizar para la demostración?

Ejercicio 2.1.27. Sean A y B los valores iniciales de a y b respectivamente. Escribir un fragmento de código que tenga $\{a = A + B \wedge b = A - B\}$ como poscondición y demostrar que el código es correcto.

En este caso, nos piden un código C que cumpla:

$$\{a = A \wedge b = B\}\ C\ \{a = A + B \wedge b = A - B\}$$

Sea $C =< a = a + b; b = a - 2b >$. Buscamnos entonces demostrar los siguientes triples:

$$\{a = A \wedge b = B\}\ a = a + b\ \{a = A + B \wedge b = B\}$$
$$\{a = A + B \wedge b = B\}\ b = a - 2b\ \{a = A + B \wedge b = A - B\}$$

Demostramos cada uno por separado:

1. Usando el axioma de asignación:

$$\{a = A + B \wedge b = B\}^a_{a+b} \equiv \{a + b = A + B \wedge b = B\} \equiv$$
$$\equiv \{a = A \wedge b = B\}\ a = a + b\ \{a = A + B \wedge b = B\}$$

2. Usando el axioma de asignación:

$$\{a = A + B \wedge b = A - B\}^b_{a-2b} \equiv \{a = A + B \wedge a - 2b = A - B\} \equiv$$
$$\equiv \{a = A + B \wedge A + B - 2b = A - B\} \equiv$$
$$\equiv \{a = A + B \wedge b = B\}\ b = a - 2b\ \{a = A + B \wedge b = A - B\}$$

Usando la regla de la composición, tenemos que el código es correcto.

Ejercicio 2.1.28. Demostrar que la siguiente sentencia tiene la poscondición $\{x \geqslant 0, x^2 \geqslant a^2\}$. `if a > 0 then x = a else x = -a`. Es decir, probar el triple:

$$\{V\}\ \texttt{if a > 0 then x = a else x = -a}\ \{x \geqslant 0, x^2 \geqslant a^2\}$$

Para ello, tenemos que usar la regla del `if`:

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

Luego bastará con probar los triples

$$\{a > 0\} \equiv \{V \wedge a > 0\}\ x = a\ \{x \geqslant 0 \wedge x^2 \geqslant a^2\}$$
$$\{a \leqslant 0\} \equiv \{V \wedge a \leqslant 0\}\ x = -a\ \{x \geqslant 0 \wedge x^2 \geqslant a^2\}$$

1. Usando el axioma de asignación:

$$\{a \geqslant 0\} \equiv \{a \geqslant 0 \wedge a^2 \geqslant a^2\} \equiv \{x \geqslant 0 \wedge x^2 \geqslant a^2\}^x_a\ x = a\ \{x \geqslant 0 \wedge x^2 \geqslant a^2\}$$

   Como $\{a > 0\} \rightarrow \{a \geqslant 0\}$, usamos la segunda regla de la consecuencia y tenemos el primer triple demostrado.

2. Usando el axioma de asignación:

$$\{a \leqslant 0\} \equiv \{-a \geqslant 0 \wedge a^2 \geqslant a^2\} \equiv \{x \geqslant 0 \wedge x^2 \geqslant a^2\}^x_{-a}\ x = -a\ \{x \geqslant 0 \wedge x^2 \geqslant a^2\}$$

Y acabamos de probar el triple que nos pedía el ejercicio.

Ejercicio 2.1.29. El siguiente fragmento de código tiene $\{P\} \equiv \left\{sum = \frac{j(j-1)}{2}\right\}$ como precondición y poscondición. Demostrar que es verdadero:

$$\{P\}\ sum = sum + j;\ j = j + 1;\ \{P\}$$

Queremos demostrar el triple:

$$\left\{sum = \frac{j(j-1)}{2}\right\}\ sum = sum + j;\ j = j + 1;\ \left\{sum = \frac{j(j-1)}{2}\right\}$$

Para ello, será suficiente con demostrar los triples

$$\left\{sum = \frac{j(j-1)}{2}\right\}\ sum = sum + j;\ \left\{sum = \frac{(j+1)j}{2}\right\}$$
$$\left\{sum = \frac{(j+1)j}{2}\right\}\ j = j + 1;\ \left\{sum = \frac{j(j-1)}{2}\right\}$$

y aplicar la regla de composición.

1. Para demostrar el primer triple, usamos el axioma de asignación:

$$\left\{sum = \frac{(j+1)j}{2}\right\}^{sum}_{sum+j}\ sum = sum + j;\ \left\{sum = \frac{(j+1)j}{2}\right\}$$

   Usando la definición de Sustitución Textual, tenemos que:

$$\left\{sum = \frac{(j+1)j}{2}\right\}^{sum}_{sum+j} \equiv \left\{sum + j = \frac{(j+1)j}{2}\right\} \equiv$$
$$\equiv \left\{sum = \frac{(j+1)j}{2} - j\right\} \equiv \left\{sum = \frac{j(j-1)}{2}\right\}$$

2. Para el segundo, usamos también el axioma de asignación:

$$\left\{sum = \frac{j(j-1)}{2}\right\}^{j}_{j+1} j = j + 1;\ \left\{sum = \frac{j(j-1)}{2}\right\}$$

   Usando de nuevo la definición de Sustitución Textual, tenemos que:

$$\left\{sum = \frac{j(j-1)}{2}\right\}^{j}_{j+1} \equiv \left\{sum = \frac{(j+1)(j+1-1)}{2}\right\} \equiv \left\{sum = \frac{(j+1)j}{2}\right\}$$

Por lo que el triple del enunciado es cierto.

Ejercicio 2.1.30. Demostrar que

$$\{i * j + 2 * j + 3 * i = 0\}\ j = j + 3;\ i = i + 2;\ \{i * j = 6\}$$

Vamos buscando aplicar la regla de la composición. Para ello, y como desconocemos el estado intermedio por el que debemos pasar, usamos directamente la Sustitución Textual al final, para así obtener la precondición del segundo triple.

$$\{i * j = 6\}^{i}_{i+2} \equiv \{(i+2) * j = 6\} \equiv \{i * j + 2 * j = 6\} \equiv \{j * (i+2) = 6\}$$

Usando esa precondición, el segundo libre se demuestra directamente con el axioma de asignación. Demostramos ahora el primer triple:

$$\{i * j + 2 * j + 3 * i = 0\}\ j = j + 3;\ \{j * (i+2) = 6\}$$

Usando la sustitución textual, tenemos que:

$$\{j * (i+2) = 6\}^{j}_{j+3} \equiv \{(j+3) * (i+2) = 6\} \equiv \{j * (i+2) + 3 * (i+2) = 6\} \equiv$$
$$\equiv \{i * j + 2 * j + 3 * i = 0\}$$

Por lo que, tras usar la regla de la composición, vemos que el triple del enunciado es cierto.

Ejercicio 2.1.31. ¿Por qué en la regla del `while B`, la condición `B` debe ser verdadera al comienzo del bucle?

Ejercicio 2.1.32. Considerar una función con dos argumentos que se usa en un programa. Explicar por qué el uso de alias puede ser un problema en este caso.

Ejercicio 2.1.33. Demostrar la corrección parcial del siguiente fragmento de programa:

```
1  sum:= 0; j:= 1;
   while j <> c do begin {<> es !=}
       sum:= sum+j;
       j:= j+1;
5  end
   {sum = c*(c-1)/2}
```

Para ello, tenemos que hacer uso de la regla de la iteración:

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

Identificando términos, sean:

$$I \equiv sum = \frac{j(j-1)}{2}$$
$$B \equiv j \neq c$$
$$S = sum = sum + j;\ j = j + 1$$

Luego tendremos que probar que se cumple el triple

$$\left\{ sum = \frac{j(j-1)}{2} \wedge j \neq c \right\}\ sum = sum + j; j = j + 1;\ \left\{ sum = \frac{j(j-1)}{2} \right\}$$

Para ello, será suficiente con demostrar los triples

$$\left\{sum = \frac{j(j-1)}{2} \wedge j \neq c\right\}\ sum = sum + j;\ \left\{sum = \frac{(j+1)j}{2} \wedge j \neq c\right\}$$
$$\left\{sum = \frac{(j+1)j}{2} \wedge j \neq c\right\}\ j = j + 1;\ \left\{sum = \frac{j(j-1)}{2}\right\}$$

y aplicar la regla de composición.

1. Para demostrar el primer triple, usamos el axioma de asignación:

$$\left\{sum = \frac{(j+1)j}{2} \wedge j \neq c\right\}^{sum}_{sum+j} sum = sum+j;\ \left\{sum = \frac{(j+1)j}{2} \wedge j \neq c\right\}$$

$$\left\{sum = \frac{(j+1)j}{2} \wedge j \neq c\right\}^{sum}_{sum+j} \equiv \left\{sum + j = \frac{(j+1)j}{2} \wedge j \neq c\right\} \equiv$$
$$\equiv \left\{sum = \frac{(j+1)j}{2} - j \wedge j \neq c\right\} \equiv \left\{sum = \frac{j(j-1)}{2} \wedge j \neq c\right\}$$

2. Para el segundo, usamos también el axioma de asignación:

$$\left\{sum = \frac{j(j-1)}{2}\right\}^{j}_{j+1} j = j + 1;\ \left\{sum = \frac{j(j-1)}{2}\right\}$$

$$\left\{sum = \frac{j(j-1)}{2}\right\}^{j}_{j+1} \equiv \left\{sum = \frac{(j+1)(j+1-1)}{2}\right\} \equiv \left\{sum = \frac{(j+1)j}{2}\right\}$$

Además, tenemos que se tiene:

$$\left\{sum = \frac{(j+1)j}{2} \wedge j \neq c\right\} \rightarrow \left\{sum = \frac{(j+1)j}{2}\right\}$$

Por tanto, usando la segunda regla de la consecuencia, tenemos que el segundo triple es cierto.

Por tanto, mediante la regla de la iteración, tenemos que:

$$\left\{sum = \frac{j(j-1)}{2}\right\}\ \texttt{while j <>c do begin sum:= sum+j; j:= j+1 end}\ \left\{sum = \frac{c(c-1)}{2}\right\}$$

Como inicialmente $j = 1$ y $sum = 0$, tenemos que este triple coincide con el enunciado del ejercicio.

Ejercicio 2.1.34. Demostrar la corrección del siguiente triple:

$$\{a[i] \geqslant 0\}\ a[i] = a[i] + a[j];\ \{a[i] \geqslant a[j]\}$$

Para ello, basta aplicar el axioma de la asignación:

$$\{a[i] \geqslant a[j]\}^{a[i]}_{a[i]+a[j]}\ a[i] = a[i] + a[j];\ \{a[i] \geqslant a[j]\}$$

$$\{a[i] \geqslant a[j]\}^{a[i]}_{a[i]+a[j]}\ \equiv \{a[i] + a[j] \geqslant a[j]\} \equiv \{a[i] \geqslant 0\}$$

Ejercicio 2.1.35. Verificar el siguiente segmento de programa:

$$\{n \geqslant 0\}$$
$$i = 1;$$
$$\texttt{while } i \leqslant n \texttt{ do begin}$$
$$a[i] = b[i];$$
$$i = i + 1;$$
$$\texttt{end}$$
$$\left\{\bigwedge_{i=1}^{n}(a[i] = b[i])\right\}$$

Para ello, como tenemos que demostrar la corrección de un bucle, hemos de buscar un invariante global que nos lleve a la poscondición indicada. Queremos demostrar que el programa copia el vector `b` en el `a`, por lo que un invariante que puede servirnos es

$$\{I\} \equiv \left\{\bigwedge_{j=1}^{i-1}(a[j] = b[j])\right\}$$

Primero, comprobamos que el invariante es cierto antes de entrar al bucle, es decir:

$$\{n \geqslant 0\}\ i = 1;\ \{I\}$$

- Usando el axioma de asignación, tenemos que:

$$\{n \geqslant 0\}\ i = 1;\ \{n \geqslant 0 \wedge i = 1\}$$

- Usando la regla de la consecuencia tenemos que es cierto, ya que:

$$\{n \geqslant 0 \wedge i = 1\} \rightarrow \{i = 1\} \equiv \left\{\bigwedge_{j=1}^{i-1}(a[j] = b[j]) \wedge i = 1\right\} \rightarrow \{I\}$$

Posteriormente, hemos de demostrar que $\{I \wedge B\}\ S\ \{I\}$ con $\{B\} \equiv \{i \leqslant n\}$ y S el cuerpo del bucle para poder aplicar la regla de la iteración. Esto lo hacemos empleando la regla de la composición y la regla de la consecuencia:

$$\{I \wedge B\} \equiv \left\{\bigwedge_{j=1}^{i-1}(a[j] = b[j]) \wedge i \leqslant n\right\}$$
$$a[i] = b[i];$$
$$\left\{\bigwedge_{j=1}^{i}(a[j] = b[j]) \wedge i \leqslant n\right\}$$
$$i = i + 1;$$
$$\left\{\bigwedge_{j=1}^{i-1}(a[j] = b[j]) \wedge i \leqslant n + 1\right\} \rightarrow \{I\}$$

Habiendo demostrado que dicho triple es cierto, podemos aplicar la regla de iteración. Usando esta y la regla de la consecuencia, tenemos que:

$$\{I\} \equiv \left\{ \bigwedge_{j=1}^{i-1} (a[j] = b[j]) \right\}$$

$$\texttt{while } i \leqslant n \texttt{ do begin}$$

$$a[i] = b[i];$$

$$i = i + 1;$$

$$\texttt{end}$$

$$\{I \wedge \neg B\} \equiv \left\{ \bigwedge_{j=1}^{i-1} (a[j] = b[j]) \wedge i > n \right\} \equiv \left\{ \bigwedge_{j=1}^{i-1} (a[j] = b[j]) \wedge i - 1 \geqslant n \right\} \rightarrow \left\{ \bigwedge_{j=1}^{n} (a[j] = b[j]) \right\}$$

Uniendo por tanto los triples, mediante la regla de la composición tenemos que:

$$\{n \geqslant 0\}$$

$$i = 1;$$

$$\{I\}$$

$$\texttt{while } i \leqslant n \texttt{ do begin}$$

$$a[i] = b[i];$$

$$i = i + 1;$$

$$\texttt{end}$$

$$\left\{ \bigwedge_{i=1}^{n} (a[i] = b[i]) \right\}$$

Esto es por tanto cierto, y hemos demostrado la corrección del programa.

Ejercicio 2.1.36. El siguiente fragmento de programa calcula $\sum_{i=1}^{n} i!$. Demostrar que es correcto.

```
1  i = 1; sum = 0; f = 1;
   while i <> n+1 do begin     {<> es !=}
     sum = sum + f;
     i = i + 1;
5    f = f * i;
   end
```

Para ello, usaremos la regla de iteración:

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

Por lo que tenemos que buscar un invariante global I que nos permita concluir al final que el programa calcula $\sum_{i=1}^{n} i!$.

Observando el código, podemos ver que en `sum` va almacenando dicho número, mientras incrementa `i` en cada iteración y va calculando en `f` el factorial de `i`. Planteamos por tanto el siguiente invariante I:

$$\{I\} \equiv \left\{ sum = \sum_{j=1}^{i-1} j! \wedge f = i! \right\}$$

En primer lugar, demostramos el triple para comprobar que el invariante es cierto al inicio del programa:

$$\{V\}\ i = 1;\ sum = 0;\ f = 1;\ \{I\}$$

Este es directamente cierto usando el axioma de asignación:

$$\{V\}\ i = 1;\ sum = 0;\ f = 1;\ \{i = 1 \wedge sum = 0 \wedge f = 1\} \equiv \left\{ i = 1 \wedge f = 1! \wedge sum = \sum_{j=1}^{0} j! = 0 \right\}$$

A continuación, trataremos de probar el triple $\{I \wedge B\}\ S\ \{I\}$, para $B \equiv \{i \neq n+1\}$ y S el cuerpo del bucle:

$$\{I \wedge B\} \equiv \left\{ sum = \sum_{j=1}^{i-1} j! \wedge f = i! \wedge i \neq n+1 \right\}$$

$$sum = sum + f;$$

$$\left\{ sum = \left(\sum_{j=1}^{i-1} j! \right) + i! \wedge f = i! \wedge i \neq n+1 \right\} \equiv \left\{ sum = \sum_{j=1}^{i} j! \wedge f = i! \wedge i \neq n+1 \right\}$$

$$i = i + 1;$$

$$\left\{ sum = \sum_{j=1}^{i-1} j! \wedge f = (i-1)! \wedge i \neq n+2 \right\}$$

$$f = f * i;$$

$$\left\{ sum = \sum_{j=1}^{i-1} j! \wedge f = i \cdot (i-1)! \wedge i \neq n+2 \right\} \equiv \left\{ sum = \sum_{j=1}^{i-1} j! \wedge f = i! \wedge i \neq n+2 \right\} \rightarrow \{I\}$$

Luego podemos aplicar la regla de iteración, para obtener finalmente que:

$$\{I\} \equiv \left\{ sum = \sum_{j=1}^{i-1} j! \wedge f = i! \right\}$$

$$\texttt{while } i \neq n+1 \texttt{ do begin}$$

$$sum = sum + f;$$

$$i = i + 1;$$

$$f = f * i;$$

$$\texttt{end}$$

$$\{I \wedge \neg B\} \equiv \left\{ sum = \sum_{j=1}^{i-1} j! \wedge f = i! \wedge i = n+1 \right\} \rightarrow \left\{ sum = \sum_{j=1}^{n} j! \wedge f = (n+1)! \right\}$$

Si tomamos `sum` como la salida del programa, tenemos probado lo que queríamos.

Ejercicio 2.1.37. Hallar la precondición $\{P\}$ que hace que el siguiente triple sea correcto:

$$\{P\}\ a[i] = 2 * b;\ \{j \leqslant i\ \wedge\ k < i\ \wedge\ a[i] + a[j-1] + a[k] > b\}$$

Para ello, basta aplicar el axioma de asignación:

$$\{j \leqslant i\ \wedge\ k < i\ \wedge\ a[i] + a[j-1] + a[k] > b\}^{a[i]}_{2 \cdot b} a[i] = 2 * b;$$
$$\{j \leqslant i\ \wedge\ k < i\ \wedge\ a[i] + a[j-1] + a[k] > b\}$$

Usando la definición de Sustitución Textual, tenemos que:

$$\{j \leqslant i\ \wedge\ k < i\ \wedge\ a[i] + a[j-1] + a[k] > b\}^{a[i]}_{2 \cdot b} \equiv$$
$$\equiv \{j \leqslant i\ \wedge\ k < i\ \wedge\ 2 \cdot b + a[j-1] + a[k] > b\}\ \equiv$$
$$\equiv \{j \leqslant i\ \wedge\ k < i\ \wedge\ b + a[j-1] + a[k] > 0\}$$

Luego estamos buscando la precondición:

$$\{P\} \equiv \{j \leqslant i\ \wedge\ k < i\ \wedge\ b + a[j-1] + a[k] > 0\}$$

Ejercicio 2.1.38. Demostrar que para $n > 0$ el siguiente fragmento de programa termina.

```
1  i = 1; f = 1;
   while i <> n do begin
     i = i + 1;
     f = f * r;
5  end
```

Ejercicio 2.1.39. Hallar la precondición de la terna:

$$\{P\}\ a[i] = b;\ \{a[j] = 2 * a[i]\}$$

Para ello, simplemente aplicamos el axioma de asignación:

$$\{a[j] = 2 * b\} \equiv \{a[j] = 2 * a[i]\}^{a[i]}_{b}\ a[i] = b;\ \{a[j] = 2 * a[i]\}$$

Ejercicio 2.1.40. Para cada uno de los siguientes fragmentos de código, obtener la poscondición apropiada:

1. $\{i < 10\}\ i = 2 * i + 1;$.
   La poscondición es $\{i < 21\}$:

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

   que puede demostrarse aplicando el axioma de asignación.

2. $\{i > 0\}\ i = i - 1;$.
   La poscondición es $\{i > -1\}$:

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

   que puede demostrarse aplicando el axioma de asignación.

3. $\{i > j\}\ i = i + 1;\ j = j + 1;$.
   La poscondición es $\{i > j\}$:

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

   Ambos pueden demostrarse aplicando el axioma de asignación y finalmente tenemos que:

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

   aplicando la regla de composición.

4. $\{V\}\ i = 3;\ j = 2 * i.$
   La poscondición es $\{i = 3 \wedge j = 6\}$:

$$\{V\}\ i = 3;\ \{i = 3\}$$
$$\{i = 3\}\ j = 2 * i;\ \{i = 3 \wedge j = 6\}$$

Ejercicio 2.1.41. Para cada uno de los siguientes fragmentos de código, obtener las precondiciones apropiadas.

1. $i = 3 * k;\ \{i > 6\}$.
   Aplicando el axioma de asignación:

$$\{k > 2\} \equiv \{3 \cdot k > 6\} \equiv \{i > 6\}^{i}_{3 \cdot k}\ i = 3 * k;\ \{i > 6\}$$

   obtenemos que la precondición es $\{k > 2\}$.

2. $a = b * c;\ \{a = 1\}$.
   Aplicando el axioma de asignación:

$$\{b = c^{-1}\} \equiv \{b \cdot c = 1\} \equiv \{a = 1\}^{a}_{b \cdot c}\ a = b * c;\ \{a = 1\}$$

   La precondición es $\{b = c^{-1}\}$.

3. $b = c - 2;\ a = a/b;$[1]

Ejercicio 2.1.42. Verificar el siguiente código. Indicar todas las reglas usadas.

$$\{y > 0\}\ xa = x + y;\ xb = x - y;$$

Ejercicio 2.1.43. Verificar el siguiente código, indicando todas las reglas usadas.

$$\{V\}\ if\ x < 0\ then\ x = -x\ \{x \geqslant 0\}$$

Para comenzar, probamos que $\{x < 0\}\ x = -x;\ \{x \geqslant 0\}$ usando el axioma de asignación:

$$\{x < 0\} \equiv \{-x \geqslant 0\} \equiv \{x \geqslant 0\}^{x}_{-x}\ x = -x;\ \{x \geqslant 0\}$$

Posteriormente, como sabemos que $\{V \wedge (x < 0)\} \equiv \{x < 0\}$ y que $\{x \geqslant 0\}\ null\ \{x \geqslant 0\}$ por el axioma de la sentencia nula, podemos aplicar la regla del `if`:

$$\frac{\{V \wedge (x < 0)\}\ x = -x;\ \{x \geqslant 0\}, \{V \wedge x \geqslant 0\}\ null\ \{x \geqslant 0\}}{\{V\}\ \texttt{if}\ x < 0\ \texttt{then}\ x = -x\ \texttt{else null}\ \{x \geqslant 0\}}$$

[1] Esto es lo que pone en la relación, falta la poscondición para poder hacerse.

Ejercicio 2.1.44. Verificar el siguiente segmento de programa:

$$max = a[1]; i = 1;$$
$$\texttt{while } i <> n+1 \texttt{ do begin}$$
$$\texttt{if } a[i] \geqslant max \texttt{ then } max = a[i];$$
$$i = i + 1;$$
$$\texttt{end}$$
$$\left\{ \bigwedge_{i=1}^{n} (max \geqslant a[i]) \right\}$$

Es decir, tenemos que probar que el código anterior calcula el máximo del vector `a` de longitud `n` (sin tener en cuenta el valor `a[0]`), que se almacena en `max`. Al tratarse de un bucle, hemos de buscar un invariante global para poder aplicar la regla de iteración. El invariante global que usamos[2] es el siguiente:

$$\{I\} \equiv \left\{ \bigwedge_{j=1}^{i-1} (max \geqslant a[j]) \right\}$$

Para comenzar, hemos de ver que el invariante es cierto al inicio del programa, es decir, que:

$$\{V\}\ max = a[1];\ i = 1;\ \{I\}$$

Lo cual es cierto, ya que:

$$\{V\}\ max = a[1];\ i = 1;\ \{max = a[1] \wedge i = 1\}$$
$$\{max = a[1] \wedge i = 1\} \rightarrow \{max \geqslant a[1] \wedge i = 1\} \rightarrow \left\{ \bigwedge_{j=1}^{i-1} (max \geqslant a[j]) \wedge i = 1 \right\} \rightarrow \{I\}$$

Posteriormente, pasaremos a demostrar el triple $\{I \wedge B\}\ S\ \{I\}$ con $\{B\} \equiv \{i \neq n+1\}$ y S el cuerpo del bucle para poder aplicar la regla de iteración:

$$\{I \wedge B\} \equiv \left\{ \bigwedge_{j=1}^{i-1} (max \geqslant a[j]) \wedge i \neq n+1 \right\}$$
$$\texttt{if } a[i] \geqslant max \texttt{ then } max = a[i];$$
$$\left\{ \bigwedge_{j=1}^{i} (max \geqslant a[j]) \wedge i \neq n+1 \right\}$$
$$i = i + 1;$$
$$\left\{ \bigwedge_{j=1}^{i-1} (max \geqslant a[j]) \wedge i \neq n+2 \right\} \rightarrow \{I\}$$

Donde hemos usado que

[2]Para buscarlo, hemos de pensar en una regla que se mantenga iteración tras iteración.

$$\left\{\bigwedge_{j=1}^{i-1}(max \geqslant a[j]) \wedge i \neq n+1\right\} \texttt{ if } a[i] \geqslant max \texttt{ then } max = a[i]; \left\{\bigwedge_{j=1}^{i}(max \geqslant a[j]) \wedge i \neq n+1\right\}$$

Para comprobar su veracidad, hemos de ver que:

$$\left\{\bigwedge_{j=1}^{i-1}(max \geqslant a[j]) \wedge i \neq n+1 \wedge a[i] \geqslant max\right\} \; max = a[i]; \; \left\{\bigwedge_{j=1}^{i}(max \geqslant a[j]) \wedge i \neq n+1\right\}$$

$$\left\{\bigwedge_{j=1}^{i-1}(max \geqslant a[j]) \wedge i \neq n+1 \wedge max > a[i]\right\} \; null; \; \left\{\bigwedge_{j=1}^{i}(max \geqslant a[j]) \wedge i \neq n+1\right\}$$

Y ambos se verifican. Podemos ya aplicar la regla de iteración para llegar a que:

$$\{I\} \equiv \left\{\bigwedge_{j=1}^{i-1}(max \geqslant a[j])\right\}$$

$$\texttt{while } i <> n+1 \texttt{ do begin}$$

$$\texttt{if } a[i] \geqslant max \texttt{ then } max = a[i];$$

$$i = i + 1;$$

$$\texttt{end}$$

$$\{I \wedge \neg B\} \equiv \left\{\bigwedge_{j=1}^{i-1}(max \geqslant a[j]) \wedge i = n+1\right\} \equiv \left\{\bigwedge_{j=1}^{n}(max \geqslant a[j])\right\}$$

Que era lo que queríamos probar.

Ejercicio 2.1.45. Demostrar la corrección parcial del siguiente código:

$$max = a[1]; i = 1;$$

$$\texttt{while } i < n \texttt{ do begin}$$

$$i = i + 1;$$

$$\texttt{if } a[i] \geqslant max \texttt{ then } max = a[i];$$

$$\texttt{end}$$

$$\left\{\bigwedge_{i=1}^{n}(max \geqslant a[i]), \bigvee_{j=1}^{n}(max = a[i])\right\}$$

Ejercicio 2.1.46. Demostrar la corrección parcial del siguiente código:

$$i = 0; j = n;$$
$$\texttt{while } i < n \texttt{ do begin}$$
$$i = i + 1;$$
$$j = j - 1;$$
$$a[i] = b[j]$$
$$\texttt{end}$$
$$\left\{ \bigwedge_{i=1}^{n} (a[i] = b[n-i]) \right\}$$

Como se trata de un bucle, hemos de usar la regla de iteración, por lo que buscamos un invariante globl que nos sirva. Observando el código, vemos que lo que hace es almacenar en el vector `a` el vector simétrico a `b` (es decir, invertir el vector `b`). A partir de esta premisa, pensamos que el invariante que nos sirve puede ser:

$$\{I\} \equiv \left\{ \bigwedge_{k=1}^{i} (a[k] = b[n-k]) \wedge j = n - i \right\}$$

En primer lugar, hemos de ver que el invariante se verifica al inicio del programa:

$$\{V\}\ i = 0;\ j = n;\ \{I\}$$

Lo cual es cierto, ya que

$$\{V\}\ i = 0;\ j = n;\ \{i = 0 \wedge j = n\}$$
$$\{i = 0 \wedge j = n\} \rightarrow \{I\}$$

Posteriormente, y con vistas a aplicar la regla de iteración, hemos de ver que se cumple el triple $\{I \wedge B\}\ S\ \{I\}$, con $\{B\} \equiv \{i < n\}$ y S el cuerpo del bucle:

$$\{I \wedge B\} \equiv \left\{ \bigwedge_{k=1}^{i} (a[k] = b[n-k]) \wedge j = n - i \wedge i < n \right\}$$
$$i = i + 1;$$
$$\left\{ \bigwedge_{k=1}^{i-1} (a[k] = b[n-k]) \wedge j = n - i + 1 \wedge i < n + 1 \right\}$$
$$j = j - 1;$$
$$\left\{ \bigwedge_{k=1}^{i-1} (a[k] = b[n-k]) \wedge j = n - i \wedge i < n + 1 \right\}$$
$$a[i] = b[j];$$
$$\left\{ \bigwedge_{k=1}^{i-1} (a[k] = b[n-k]) \wedge a[i] = b[n-i] \wedge j = n - i \wedge i < n + 1 \right\} \equiv$$
$$\equiv \left\{ \bigwedge_{k=1}^{i} (a[k] = b[n-k]) \wedge j = n - i \wedge i < n + 1 \right\} \rightarrow \{I\}$$

donde hemos usado que

$$\{j = n - i\}\ i = i + 1;\ j = j - 1;\ \{j = n - i\}$$

que puede probarse mediante composición de los triples

$$\{j = n - i\}\ i = i + 1;\ \{j = n - i + 1\}$$
$$\{j = n - i + 1\}\ j = j - 1;\ \{j = n - i\}$$

que sabemos que son ciertos por el axioma de asignación.

Podemos finalmente aplicar la regla de iteración, llegando a que:

$$\{I\} \equiv \left\{ \bigwedge_{k=1}^{i} (a[k] = b[n-k]) \wedge j = n - i \right\}$$
$$\texttt{while } i < n \texttt{ do begin}$$
$$i = i + 1;$$
$$j = j - 1;$$
$$a[i] = b[j]$$
$$\texttt{end}$$
$$\{I \wedge \neg B\} \equiv \left\{ \bigwedge_{k=1}^{i} (a[k] = b[n-k]) \wedge i \geqslant n \right\} \rightarrow \left\{ \bigwedge_{k=1}^{n} (a[k] = b[n-k]) \right\}$$

Que era lo que queríamos probar.

Ejercicio 2.1.47. Demostrar la corrección parcial del siguiente código:

```
1  i = 0;
   s = 0;
   while i <= n do begin
     s = s + a[i];
5    a[i] = s;
     i = i + 1;
   end
```

Ejercicio 2.1.48. Dados $n \geqslant 0$, $i \leqslant n$, demostrar que el siguiente segmento de progama evalúa

$$\frac{n!}{i!(n-i)!}$$

```
1  k = 0; fact = 1;
   while k <> n do begin
     k = k + 1;
     fact = fact * k;
5    if k <= i then afact = fact;
     if k <= n-i then bfact = fact;
   end
   bcof = fact/(afact*bfact);
```

Ejercicio 2.1.49. Demostrar la terminación del fragmento de programa dado en el problema 2.1.44 ¿Qué condición se debe imponer para realizar la demostración?